Adaptive Caching for Demand Prepaging

Scott F. Kaplan
Department of Mathematics
and Computer Science
Amherst College
Amherst, MA 01002-5000
sfkaplan@cs.amherst.edu

Lyle A. McGeoch
Department of Mathematics
and Computer Science
Amherst College
Amherst, MA 01002-5000
lam@cs.amherst.edu

Megan F. Cole
AC #961
Amherst College
Amherst, MA 01002-5000
mfcole@amherst.edu

ABSTRACT

Demand prepaging was long ago proposed as a method for taking advantage of high disk bandwidths and avoiding long disk latencies by fetching, at each page fault, not only the demanded page but also other pages predicted to be used soon. Studies performed more than twenty years ago found that demand prepaging would not be generally beneficial. Those studies failed to examine thoroughly the interaction between prepaging and main memory caching. It is unclear how many main memory page frames should be allocated to cache pages that were prepaged but have not yet been referenced. This issue is critical to the efficacy of any demand prepaging policy.

In this paper, we examine *prepaged allocation* and its interaction with two other important demand prepaging parameters: the *degree*, which is the number of extra pages that may be fetched at each page fault, and the *predictor* that selects which pages to prepage. The choices for these two parameters, the reference behavior of the workload, and the main memory size all substantially affect the appropriate choice of prepaged allocation. In some situations, demand prepaging cannot provide benefit, as any allocation to prepaged pages will increase page faults, while in other situations, a good choice of allocation will yield a substantial reduction in page faults. We will present a mechanism that dynamically adapts the prepaged allocation on-line, as well as experimental results that show that this mechanism typically reduces page faults by 10 to 40% and sometimes by more than 50%. In those cases where demand prepaging should not be used, the mechanism correctly allocates no space for prepaged pages and thus does not increase the number of page faults. Finally, we will show that prepaging offers substantial benefits over the simpler solution of using larger pages, which can substantially increase page faults.

1. INTRODUCTION AND MOTIVATION

The gap between main memory and disk access times has been large for decades and continues to grow. Because of this gap, it is critical that virtual memory systems reduce the number of *page faults*—references to pages that are on disk. A *demand paging* virtual memory system responds to each page fault by fetching from disk only the referenced demanded page. In contrast, *demand prepaging* was proposed as a method for reducing the number of page faults by allowing extra pages to be fetched along with the demanded page. If those extra pages are referenced soon after they are fetched, then the virtual memory system avoids the separate disk accesses that would otherwise have been needed to fetch each one.

The vast majority of the delay for fetching a single page is due to the *latency* of a disk access, which is typically somewhere from 1 to 10 milliseconds. The *transfer delay*, which is the time required to read and transmit the page from the backing store into main memory, is much smaller—on the order of tenths of milliseconds. Consequently, the additional delay caused by "piggy-backing" the transfer of extra pages along with the demanded page can be negligible[1]. It is more important that a virtual memory system reduce the total number of page faults than reduce the number of pages transferred between main memory and disk, as disk bandwidths have been improving at a greater rate (approximately 20% per year) than disk latencies (approximately 5% per year).

In spite of the recent decrease in the cost of RAM that has made it possible for most systems to contain large main memories, paging has not ceased to be a problem. Firstly, the memory requirements of operating systems and software packages have also grown, expanding to fill the available main memory space. Secondly, RAM is not always a low-cost addition to a system. While 256 MB currently costs as little as US$50 for a desktop or server system, the same memory addition to a notebook computer costs US$270[2]. These costs assume that the system owner is capable of installing the RAM herself—an invalid assumption for a large fraction of computer users, who instead would have to incur additional labor costs to have the RAM installed. Such users tend not to upgrade any part of their systems, instead buying whole new systems when performance becomes intolerable. We must also consider that some systems, particularly notebook machines, have limited RAM capacity: few notebooks current allow for more than 1 GB of RAM, while some lightweight and slim ones can accept at most

[1]The ability to fetch these extra pages within the same disk access requires that the pages be *clustered*. We will address past work on this problem in Section 2.

[2]These prices were taken from `<http://www.mcglen.com>`.

320 MB of RAM. For such machines, the low cost of RAM is irrelevant.

Paging remains a relevant topic because most systems do page *sometimes*, even if typical systems do not page under most circumstances. When a system does need to page, a modest amount of paging can cause a dramatic decrease in performance. Improvements in virtual memory management should seek to make the consequences of paging less drastic. We seek a *graceful degradation* in performance as main memory resources become scarce relative to the needs presented by the workload. If a workload requires only modestly more memory than is available in RAM, then the virtual memory system should allow that workload to execute with tolerable overhead. Demand prepaging can help to make it possible to execute workloads with memory needs in modest excess of main memory availability, in an acceptable amount of time, and without requiring the user to buy more RAM or find another machine—two solutions that are themselves time-consuming.

The need for a new evaluation. There have been a number of studies of demand prepaging. Some of these studies required that programmers or compilers enable applications to provide hints to the system about future references [14, 18, 19, 20]. Others assumed no such hints, performing prepaging based only on reference history [2, 8, 9, 17]. Because the methods requiring hints are applicable only to numeric and scientific applications whose reference patterns are exceedingly regular, and because many applications do not provide such hints, we will only consider demand prepaging without hints.

The results from the studies of demand prepaging without hints have been ambiguous. Joseph [9] and Smith [17] concluded that demand prepaging was only of benefit only for main memories large enough to cache most of footprint, or for systems that use exceedingly small page sizes (32 to 64 bytes). Since little paging occurs when main memory can nearly cache the entire process footprint, and since such small page sizes were not practical for virtual memory management, they believed that demand paging did little to reduce page faults and could often increase them under other circumstances. Although Baer and Sager [2] found some reduction in page faults with demand prepaging, they also found that variations in the prepaging mechanisms had little effect on the outcome. Horspool and Huberman [8] found that varying the rapidity of eviction for prepaged but unused pages had a negligible effect on prepaging results.

In contrast, we have found that demand prepaging without hints can be effective for a wide range of memory sizes on page sizes typical for virtual memory systems. We have also found that variations in prepaging policy, particularly the amount of time for which prepaged but not-yet-used pages are cached, can have a dramatic effect on performance. Because prepaging seems to provide benefit for a variety of applications, and because the parameters that are critical to a prepaging policy interact in ways that were not thoroughly evaluated in these past works, we believe that an analysis of prepaging and its interacting factors is needed. We seek to provide that analysis by highlighting three critical, interacting factors:

- **Prepaged allocation:** The number of main memory page frames dedicated to the caching of *prepaged pages*—extra pages fetched during a fault on some other page but not used since that fault.

- **Degree:** The number of extra pages that may be fetched at a page fault.

- **Prediction:** The policy that determines which extra pages to prepage.

Previous studies have examined different predictors. However, higher degrees have not received much attention, and the most critical component, prepaged allocation, has been addressed superficially. We will show that good choices for prepaged allocation vary with the application and total memory size. Furthermore, we will show that the choices of degree and predictor also strongly influence the best choice of prepaged allocation.

Our experiments will show that simple predictors yield good prepaging performance. Two simple predictors—one based on spatial locality, the other on temporal locality—performed well for a variety of applications and memory sizes. We will also see that a small degree of 2 to 4 pages is all that is required for a substantial reduction in page faults. Finally, we will show that a good choice of prepaged allocation can yield a useful reduction in page faults, but a poor choice can substantially increase page faults.

Real kernels may be using prepaging badly. A real VM system either prepages or it does not. Those systems that do not employ prepaging are missing its potential benefits. Systems that *do* prepage, if not designed to adapt to program reference behavior, will either obtain only part of the possible benefit, or worse, will suffer an increase in page faults relative to a non-prepaging system. Previous work on the clustering of pages on a backing store [3] describes the inclusion of prepaging in OSF/1 UNIX in order to match another UNIX implementation that also prepaged. Recent versions of the Linux kernel also read all pages on disk that are located in the same sector as the demanded page. The cost of reading such pages is so low that many kernels are likely to employ some kind of prepaging.

Because of the lack of recent research on demand prepaging, there is little to guide kernel implementors in choosing prepaged allocation, degree, and predictor. Our experiments show that no single choice of prepaged allocation is "safe," as it will yield an increase in page faults for some combination of input application, memory size, degree, and predictor. Only a virtual memory system that dynamically adapts the prepaged allocation can obtain the benefits of prepaging while avoiding its potential harm. We seek to provide the analysis that kernel implementors need to use prepaging well.

Dynamic, on-line adaptation of prepaging parameters. Since prepaged allocation should be dynamically adapted on-line, we will present an approach for adapting allocation that, in simulated experiments, yields a reduction in page faults that typically matches or exceeds the best possible fixed allocation. Just as importantly, we will see that this adaptive mechanism successfully avoids allocating *any* main memory space to prepaged pages if that allocation would increase the fault rate.

Road map. Section 2 will provide a more detailed background on demand prepaging, including a discussion of related work. Section 3 will show the results of exploratory experiments that reveal the possible benefits and costs that prepaging can yield with different configurations. We will use those results in Section 4 to describe a mechanism for dynamically adapting the prepaging configuration of a system. Section 5 will describe our trace-driven experiments with different forms of prepaging and provide the results and discussion of those experiments. Section 6 will summarize the results of our work and examine some future research directions.

2. BACKGROUND AND RELATED WORK

We begin by establishing a few terms and concepts. We will establish the circumstances for which prepaging may be either harmful or beneficial to a virtual memory system.

2.1 When is prepaging beneficial?

Without prepaging, every page frame in main memory is used to cache pages that have been referenced since being fetched. We will refer to such pages as *used pages.* With prepaging, some number of page frames are used to hold pages that were speculatively fetched but have not yet been referenced during their residency. These pages we will call *prepaged pages.* Thus, prepaging introduces a contention for main memory space between used and prepaged pages.

Allocation, costs, and benefits. Prepaging can be beneficial only if the prepaged pages are referenced before they are evicted. Prepaging can be harmful only if the prepaged pages displace used pages that, under a non-prepaging system, would have been re-referenced before being evicted[3]. In other words, the *benefit* of prepaging is the number of references to prepaged pages that would not have been resident under a non-prepaging policy, thus avoiding a page fault; the *cost* of prepaging is the number of references to pages that are not resident under the given prepaging policy, but would have been resident under a non-prepaging policy. If some partitioning of main memory yields greater benefit than cost, then prepaging can be used to reduce the number of faults. If no such partitioning exists, then the use of prepaging would pollute main memory.

Two notions of prepaged allocation. One simple approach to prepaging could be to dedicate some number of main memory page frames to caching prepaged pages, where those page frames would never cache a used page. Such an assumption is undesirable, as any reference may turn a prepaged page into a used page. With a rigidly fixed partitioning, a reference to a prepaged page could cause the eviction of a used page even though prepaged page frames remain unoccupied.

[3]Smith [17] incorrectly characterizes this cost. He states that a prepaging will be detrimental if the displaced, used page is referenced before the prepaged page. This description is an oversimplification in two ways: firstly, the prepaged page may still be referenced before it is evicted, implying no net change in page faults, as one fault has been saved to create another one. Secondly, if the used page *would have been evicted before being referenced irrespective of the prepaging*, then the prepaging has not raised the number of page faults.

To avoid this undesirable situation, we must allow the number of page frames holding prepaged pages to shrink between page faults as prepaged pages are referenced. When a prepaged page becomes a used page, we simply allow the partitioning of main memory to change correspondingly. However, we may still wish to establish an upper bound on the allocation to prepaged pages. We introduce two terms to differentiate between this upper bound on page frames for prepaged pages and the dynamically fluctuating number of page frames holding prepaged pages:

- The **consumed allocation** is the number of page frames that are being used to cache prepaged pages at a particular moment.
- The **target allocation** is the maximum number of page frames that may be used for caching prepaged pages.

Note that the consumed allocation can increase only at a page fault and decrease only when references to prepaged pages occur. Thus, the consumed allocation cannot directly be chosen—only the target allocation may be selected as a parameter to a prepaging policy. Also note that it is the consumed allocation at each moment that determines whether or not a reference will contribute to the cost, to the benefit, or to neither.

How degree and predictor affect cost and benefit. The number of hits to prepaged pages is dependent not only on the allocation of page frames to prepaged pages, but also on the *choice* of prepaged pages. Both *degree* and *predictor* determine which pages are speculatively fetched and therefore have direct impact on the choice of target allocation.

A good predictor will select pages that the process will use very soon. For such a predictor, a page will be a prepaged page for a short time before it becomes a used page. Therefore, a small target allocation will yield a large number of hits to the prepaged page partition of main memory, thus displacing very few used pages.

A poor predictor will select pages that the process may not reference for some time. Therefore, a larger target allocation must be chosen to yield hits to the prepaged page partition. This larger allocation implies a smaller allocation to used pages and thus a larger potential cost for prepaging. The net benefit of using some predictor depends therefore on the choice of target allocation.

Degree relates similarly to target allocation. A small degree implies that few pages are being fetched into the prepaged page partition at each fault. A smaller target allocation can hold pages that were prepaged during some number of preceding faults, and the turnover of pages in the prepaged page partition is slow. With a larger degree, more pages are fetched at each fault. To cache the pages that were prepaged for the same number of preceding faults, a larger target allocation may be required.

Note the relationship, however, between predictor and degree. For a good predictor, a higher degree may be desirable, as each fault allows the system to fetch more pages that are likely to be referenced soon, thanks to a high hit rate for the prepaged pages. If a smaller degree were used, more page faults would need to occur in order to fetch the same pages, thus limiting the benefit of the prepaging mechanism.

2.2 Previous prepaging studies

Previous studies not only examined demand prepaging without hints, but also page clustering on the backing store, filesystem prefetching and caching, and page replacement.

OBL. Perhaps the earliest work on prepaging was Joseph's paper on the *One Block Lookahead (OBL)* policy [9]. Under this policy, whenever a reference to some page[4] number i causes a page fault, then not only should page i be fetched, but also page $i + 1$ if it is on disk. This simple policy relied on the common property of *spatial locality*; that is, if page i is referenced, then it is likely that pages near i in the address space will also be referenced soon.

Joseph recognizes the potential for main memory pollution due to prepaging. To avoid the eviction of used pages in favor of these speculatively prepaged pages, he recommends that prepaged pages be placed at the end of the LRU queue of resident pages. Under this scheme, a prepaged page must be used before the next page fault or it will be evicted. A target allocation of a single page frame to hold one prepaged page is likely to be insufficient in many cases. It is therefore not surprising that experiments with OBL yielded little benefit.

Smith [17] also performed experiments with OBL and a different allocation scheme. Specifically, when pages i and $i + 1$ are fetched, they are respectively inserted as the first and second elements in the LRU queue. A target allocation of 50% of main memory is implied by this mechanism. Smith found, in his experiments, that prepaging often increased the number of misses. This result is not surprising given a target allocation that may have been too large for many applications and memory sizes. Smith also found that prepaging was only of benefit at memory sizes large enough that little faulting occurred, and thus where a decrease in allocation to used pages is less likely to increase misses than with a smaller memory size.

OBL/k. Horspool and Huberman [8] proposed *OBL/k*. This policy differed from both Joseph's and Smith's versions of OBL in one aspect that is critical to our study: it took a parameter k that specified the rate at which prepaged pages were evicted. Higher values of k implied that pages moved more quickly toward eviction. Consequently, k dictates the target allocation, as the two values are inversely related.

Horspool and Huberman found that, in general, OBL/k was able to reduce modestly the number of page faults. Oddly, they found that the value of k seemed to have little effect on their results. We believe that the test suite on which they performed their experiments was too limited to show the consequences of prepaged allocation, as they had few reference traces, and those traces may not have exhibited interesting reference behavior at the memory sizes examined.

OSF/1 Clustered Paging. Some operating system kernels already contain virtual memory systems that use demand prepaging. Black, et al. [3] present developments in the OSF/1 UNIX virtual memory system, including the addition of *clustered paging*. Although the system described in this paper is relatively old, it is one of the few published descriptions of a VM system's prepaging structure. It also represents a simple and likely approach to prepaging. The researchers discuss their implementation of *page clusters*, which are groups of pages that are adjacent in the virtual address space and are stored contiguously on disk. Each cluster holds as many as 8 pages, and all of the non-resident pages of a cluster are fetched when one of them is demanded.

This study does not consider the problem of prepaged allocation. However, the authors do state that the implementation of clustered paging required little modification to the kernel's virtual memory functions, which "already support multiple pages." The unverified implication is that all fetched pages are treated just as the referenced page is. If that interpretation is correct, this implementation may be allowing as much as $\frac{c-1}{c}$ of main memory, where c is the size of the cluster, to be allocated to prepaged pages. We will see that such a large allocation can be detrimental to page faults rates for some applications.

The Linux 2.4 VM system. The Linux 2.4 kernel is more recent than OSF/1, but has a VM system that is less well documented and studied. The documentation project for this VM [11] indicates that Linux systems prepage in a manner than may be harmful for some workloads. Specifically, Linux will prepage all pages located in the same sector as the demanded page, thereby reading multiple pages with a single seek operation. There is some effort made by the page cleaning mechanism to cluster pages on disk by their addresses, thus implying prediction based on spatial locality.

Critically, prepaged pages are categorized as *active*—a designation that places them among the most recently used pages. While Linux does not keep a strict LRU ordering of resident pages, its approximation of LRU implies that some time will have to pass before a prepaged page is evicted. Consequently, the prepaged allocation has the potential to be substantial. Linux therefore is likely to be displacing used pages in favor of prepaged pages in circumstances where the caching of used pages would be more valuable. Furthermore, because Linux does not dynamically adapt the prepaged allocation, it is also likely that in some cases, prepaged pages are being evicted prematurely.

Filesystem prefetching studies. There have been recent studies that examine the interaction of *prefetching*—the speculative fetching of disk blocks that may be requested soon—and the filesystem cache, which holds filesystem blocks in main memory. Both Cao, et al. [4], and Patterson, et al. [15] evaluated methods of partitioning the filesystem cache between requested blocks and prefetched blocks. The on-line policies evaluated in these studies required accurate hints from the application.

Patterson, et al. [15] also presented a more thorough analysis of the costs and benefits of caching prefetched blocks. They propose an approximate, on-line valuation of used blocks by keeping a histogram of the hits to the different positions of the LRU queue that orders those used blocks. From this histogram, a function can be computed that indicates the number of misses suffered at each possible allocation for used blocks. This approach to tracking potential costs and benefits is similar to the one we will present in Section 4 as part of a mechanism to dynamically adapt the target allocation.

Some of the work in filesystem prefetching has concen-

[4]We assume *page* and *block* to be synonyms in the present context.

trated on the development of more complex predictive models [7, 13]. These studies focus on the ability to detect sequences of block use, much as some data compression models detect sequences of bytes. The researchers have demonstrated that these more complex models yield better prefetching predictions. A somewhat similar, sequentially-based predictor for VM prepaging systems was proposed by Cho and Cho [6]. While improved predictors are certainly of interest to any prefetching or prepaging system, we seek to focus also on the factors of allocation and degree and to show their interaction with different predictors. We therefore will not consider more complex predictors in this paper.

Other relevant studies. While the heuristic that we will present in Section 4 is somewhat similar to the one presented by Patterson, et al. [15], it bears greater similarity to adaptive mechanisms presented in other virtual memory studies. Specifically, [16] presents EELRU, a page replacement policy that performs a cost/benefit analysis for different choices of eviction strategies, while [10, 22] address the partitioning of main memory into compressed and uncompressed partitions, with the sizes of the partitions determined by cost/benefit comparisons. Both of these mechanisms order pages within an LRU queue and keep histograms of hits to each LRU queue position so that they may calculate costs and benefits. The EELRU and compressed caching mechanisms also decay the contents of the histogram so that the calculations reflect recent reference behavior. We will address the adaptive mechanism in more detail in Section 4.

3. EXPLORATORY EXPERIMENTS

In this section, we will present the results of exploratory experiments in which we simulated a demand prepaging virtual memory system. In these experiments, we varied the predictor, degree, and target allocation in order to see the relationships between them and the regularities of their effects on prepaging performance. For these exploratory experiments, no dynamic adaptation was used—the values assigned to the three parameters were fixed for the duration of each simulation. The reference traces used as inputs to the simulations were gathered from executing applications. The results of the simulations revealed the need for dynamic adaptation in prepaging implementations.

3.1 Experimental design

3.1.1 The fixed-parameter policies

Below, we describe these prepaging policies for which the parameters are fixed for the lifetime of the simulations. We also specify the values and ranges used for the three parameters.

Common elements. The basic structure of our prepaging policy involved two LRU page queues: one for prepaged pages, known as the *prepaged queue*, and one for used pages, known as the *used queue*. Given a total main memory size of k pages, the sum of the sizes of both queues may not exceed k. Initially, both queues are empty. As the first k pages are referenced, the used queue will grow to hold those k pages, and the prepaged queue will remain empty.

When the $k + 1^{st}$ page has been referenced, the least recently used page will be evicted from the used queue. Once pages have been evicted, prepaging becomes possible. If a page fault is caused by a reference to a page on the backing store, the referenced page is fetched. Additionally, the predictor selects d other pages, where d is the degree, as candidates to be prepaged. Whichever of those d candidates are on the backing store will be fetched along with the referenced page. The demanded page is inserted at the front of the used queue, and the prepaged pages are inserted at the front of the prepaged queue.

For each page fetched, another page must be evicted. Assume that a total of f pages are to be prepaged, where $f \leq d$, thus implying that a total of $f + 1$ pages to be fetched. The number of prepaged pages to be evicted is the number by which the prepaged queue would exceed its target allocation after adding those f pages. In other words, pages are evicted from the prepaged queue only if it would otherwise exceed its target allocation. Any remaining evictions are performed on pages from the used queue.

When a used page is referenced, it is brought to the front of the used queue. When a prepaged page is referenced, it is removed from the prepaged queue and inserted at the head of the used queue. As mentioned in Section 2.1, this movement of pages allows the sizes of the prepaged and used queues to vary dynamically as references occur.

Main memory and target allocation ranges. For each reference trace, we simulated several possible main memory sizes. The upper bound of this range was the footprint of the traced process—that is, the smallest memory size for which every page used by the process is cached. In determining a lower bound, we observe that it is undesirable to simulate an arbitrarily small memory, as such small memory sizes are unlikely ever to be used due to the overwhelmingly large number of misses that they would cause with *any* memory management policy.

To concentrate our analysis on a range of "interesting" memory sizes, we chose as the smallest memory size the number of pages for which paging causes the process to run approximately one thousand times slower than if no paging had occurred. This estimate is conservative in that there are few situations, if any, for which a user would wait so long for a program that paged so much. In order to calculate this smallest memory size, we had to approximate both the run-time of each process represented by the traces as well as the average disk access time for page faults. We assumed a processor that could execute 500 million instructions per second and a disk with a 5 ms access time.

For each memory size, we chose to simulate target allocations between 0% and 75% of that main memory size. We chose 75% as an upper bound because it will be uncommon that such a large target allocation would be beneficial and because these experiments sought only to show the effects of varying target allocations.

Degree ranges. We chose to simulate degrees from one to eight pages. We did not simulate larger degrees because we found that substantial benefit was possible with smaller degrees and because it would be difficult to cluster larger numbers of pages on the backing store.

Predictors. We have simulated three predictors, listed below. The first two were chosen for their simplicity and their resemblance to predictors used in past studies. The third is chosen as a benchmark to aid in interpreting the results of the first two. While other predictors are always possible, we sought to demonstrate that prepaging is viable even with unsophisticated prediction techniques.

- **Address-local:** This predictor assumed that pages nearby in the address space to the one being demanded would be likely to be used soon. More specifically, for a degree d and a referenced page number n that caused the page fault, this predictor would select as prepaging candidates the first d pages in the sequence $n+1$, $n-1$, $n+2$, $n-2$, $\ldots$

- **Recency-local:** This predictor assumed that pages nearby in an LRU ordering to the one being demanded would likely be used soon. For a degree d and a reference to page found at LRU queue position p, this predictor would select as prepaging candidates the first d pages found in the sequence of positions $p-1$, $p+1$, $p-2$, $p+2$, $\ldots$[5]

- **Pessimal:** This predictor always selected d *dummy pages*—pages that were never used by the process. This predictor represents a useful worst-case in two ways. Firstly, it always selects pages that will not be used soon. Secondly, it ensures that the consumed allocation will quickly reach and remain at the target allocation, thus reducing the space for used pages as much as possible. This predictor represents the harm that can be done by prepaging with a poor predictor and a non-zero target allocation.

3.1.2 The traces

We used ten references traces gathered using the `Etch` instrumentation tool and presented in [12]. These traces were gathered on a Windows NT system and included a mix of both batch-style processes (gcc, compress, perl, vortex) and interactive, GUI processes (Acrobat Reader, Go, Netscape Navigator, Photoshop, Powerpoint, and Word). While no set of traces is representative of all applications, nor of the use of those particular applications, we believe this set represents a sufficiently varied set of applications and uses to be illustrative.

It is also important to note that the footprints of the processes captured in these traces are not large when compared to modern main memory capacities. The smallest of the processes requires approximately 1 MB, and the largest requires a few dozen MB. While gathering new traces of processes with larger footprints would be desirable, such a task is a substantial undertaking. More importantly, we believe that by examining each process over the interesting memory size range as described above, we demonstrate the generality of our results for a variety of reference patterns and footprints. What is more critical is observing the behavior of memory management policies over the interesting range of memory sizes, where that range is relative to the given footprint. We will see that our results are consistent across significant variation in memory requirements for the processes represented in the traces used here.

[5]The sequence of pages selected by this predictor begins with *preceding* pages in the LRU queue first because of the arguments presented by Wilson, et al. in [21] regarding Horspool and Huberman's OBL/k policies. To summarize the argument, LRU queues provide a relative order of last reference to pages. Pages that precede the referenced page are pages that were last used *after* the referenced page, and thus may be better choices for prepaging than a page that was last used *before* the referenced page.

3.2 Results and observations

Each simulation yielded two essential results:

- **Misses:** The number of references to pages on the backing store, each of which causes a page fault[6].

- **Transfers:** The number of pages, both demanded and prepaged, that are fetched from the backing store to the main memory.

The number of misses is associated with the paging time lost to disk *latency*, as each page fault requires a separate disk access. The number of transfers is related to the use of disk *bandwidth*, as each fetch requires that another page-sized block of data be transferred from disk to main memory. Demand prepaging tends to increase the number of transfers, as some prepaged pages are evicted before being used, and thus correspond to transfers that a demand paging policy would not have incurred. We examine both values, but we believe that the number of misses is far more important given disk characteristics. Bandwidth has improved more rapidly than latency, and for smaller degrees, the increased number of transfers is not substantial. However, if paging is performed on more heavily loaded disks, or if the backing store device is not a disk, then the increase in bandwidth consumption may be undesirable. For such systems, the results for transfers should be given greater weight.

When prepaging can yield benefit. In Figure 1, we see a typical example of an application where prepaging would reduce the number of page faults. This figure represents **photoshp** (Adobe Photoshop) running within a 796-page main memory. The x-axis is the target allocation as a percentage of the main memory. The y-axis shows the percentage of misses relative to a demand paging, LRU-managed main memory of the same size.

We can see that when no space is allocated for prepaged pages, all of the policies converge to producing 100% of the number of misses for demand paging under LRU. As we look from left to right on the plot, we see that different predictors yield different results for the same target allocation. At this memory size, this process could have benefited under either address-local or recency-local prediction, although recency-local is the better choice in this case. Observe that target allocations need not be large to obtain the best results—here, roughly 15% to 20% of the memory would be best. We can also see that too large a target allocation increases page faults by displacing used pages. Finally, we can see that the range of target allocations for which benefit is obtained is a reasonably wide range, from just a few percent to 25% and 45% for address-local and recency-local, respectively.

Most importantly, we notice that the shape of these curves is a regular "U" shape. This characteristic was present in all of our fixed-allocation results where prepaging was able to provide benefit, implying that a simple search heuristic is likely to find a good target allocation. Finally, we note that

[6]Note that we do not include, in this count, *compulsory misses*, which are the references to pages that have never before been used and thus would be misses for any policy.

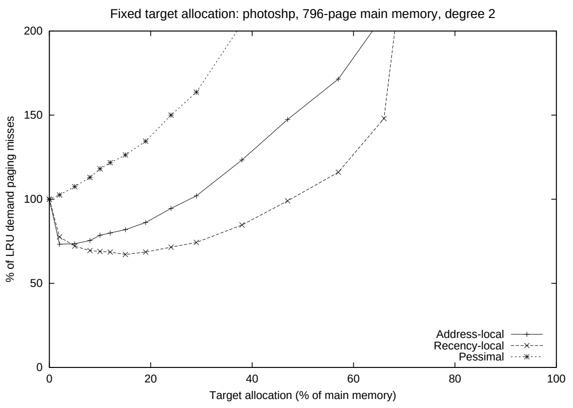

Figure 1: An example where, for each predictor over a range of fixed target allocations, prepaging would reduce the number of page faults.

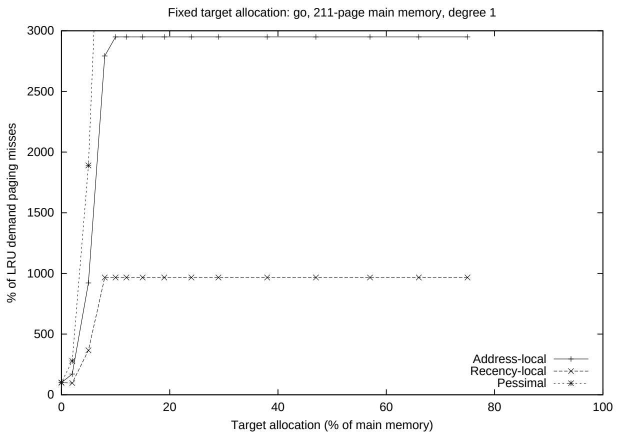

Figure 2: An example where both predictors at any fixed target allocation would increase the number of page faults.

while these two predictors can reduce page faults by approximately 15% and 30% respectively, it is possible, at a degree of 2, for a perfect predictor to obtain a reduction of approximately 66% by eliminating two out of three page faults, given the degree of 2. Although there is unlikely to exist an on-line predictor that exhibits such optimal behavior, there is still substantial room for improved on-line prediction.

When prepaging cannot yield benefit. We see, in Figure 2, the application **go** (a Go player) at a memory size where nearly any reduction in allocation to the used queue causes the number of page faults to rise dramatically. For the pessimal predictor, we can see a rapid increase in misses, indicating the high cost for any target allocation. Notice that even for small target allocations of less than 10% of main memory, both predictors can lead to enormous increases in page faulting: recency-local can cause a ten-fold increase, and address-local a thirty-fold increase. Nonetheless, we still cannot claim that it is not possible for prepaging to have reduced the page faulting, as a perfect predictor could still eliminate 50% of the page faults given the degree of 1. It is merely unlikely, in a case like this one, that an on-line predictor could provide benefit that exceeds the cost.

It is also possible not to obtain benefit if a predictor rarely selects candidates for prepaging that are on the backing store. In this case, the consumed allocation remains low, and prepaging has little effect. Such cases tend to occur only for larger memory sizes where few faults occur, or for pathological access patterns that prevent useful prediction.

Total memory size vs. ideal target allocation. We would like to identify the target allocation for which prepaging proves *most* beneficial for a given application with a given total memory size. Figure 3 shows a plot of the best fixed target allocations for **acroread** (Acrobat Reader) over the interesting range of memory sizes (as described in Section 3). Notice that the ideal target allocation not only varies over memory sizes, but is different for each predictor. Unsurprisingly, the ideal target allocation also varies between applications. Results from the other traces in our test suite revealed that there was no pattern between the predictor, memory size, and ideal target allocation with respect to the best, fixed target allocation.

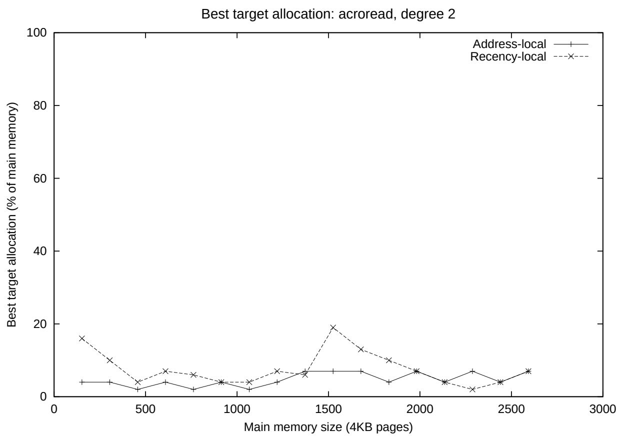

Figure 3: The best, fixed target allocation varies with the main memory size and the predictor.

Choosing between predictors. The performance of a prepaging policy depends on the quality of the predictions. We found that neither predictor dominates the other for all applications. In contrast to what we see in Figure 1, where recency-local is the better predictor, the results for a number of other traces and memory sizes are better for the address-local predictor. For a single application, the preferred predictor may change with the memory size. The efficacy of a predictor depends on the reference behavior for the given application *and* the given memory size.

While previous work has demonstrated that clustering pages on disk according to their spatial locality is possible, no such previous work exists for clustering pages according to their temporal locality. While we believe that such a clustering is possible, the address-local predictor is more immediately applicable because of corresponding known clustering strategies.

Choosing a degree. We found that higher degrees often yielded more benefit. However, small degrees of 2 to 4 pages obtained substantial reductions in page faults, and as the degree grew beyond 4, the additional reductions were diminishing. We will see, in Section 5, these improved reductions with higher degrees; however, we will also see that higher degrees increase transfers, and make dynamic adaptivity less stable.

4. A DYNAMICALLY ADAPTIVE PREPAGING POLICY

We have developed a strategy for adapting on-line the target allocation. In this section, we will describe the mechanism by which adaptivity occurs and consider the feasibility of its implementation in a real system. We do not attempt to adapt the degree because smaller degrees can consistently yield good performance, therefore implying that it is not critical that degree be dynamically adapted. We did attempt successfully to adapt between predictors. However, it is unclear whether or not a clustering mechanism could be realized that would allow for dynamic changing of the predictor. We will therefore not address that mechanism here.

4.1 The Queues and Histograms

The queues used for dynamic adaptation are arranged in a manner similar to the fixed-parameter policies described in Section 3. There are two queues: one used to order used pages, and the other to order prepaged pages. However, unlike the fixed-parameter policy, these queues hold both resident *and* non-resident pages. The rules for the movement of pages within the queues are the same as with the fixed-parameter policies: Used pages are always moved to the head of the used queue, and prepaged pages are always moved to the head of the prepaged queue. A page remains in its queue when it is evicted. As before, a page can only be prepaged if it is not resident. By extending these queues to record the relative order of last use or last prepage among the non-resident pages, we can determine how large an allocation would have been required to have successfully cached each page.

With each queue, we maintain a *hit histogram.* For each position in a given queue, we maintain a count of the number of times a referenced page was found in that position. Thus, if a referenced page is found in position i of the used queue, then we increment entry i in the used queue's histogram. By keeping this information about past references, the policy can determine how many misses would have been incurred for each possible partitioning of main memory between used and prepaged pages. Therefore, the mechanism can calculate the cost and benefit of each possible target allocation, allowing the policy to choose dynamically a good value for that parameter.

4.2 Periodic Decay

The histograms, as described above, will contain the *entire* referencing history of the process, giving all past references equal weight. Real processes tend not to access memory in the same pattern throughout their executions. Most processes exhibit infrequent but drastic changes in reference behavior known as *phase changes.* After a phase change occurs, those references performed in the distant past tend to predict poorly the reference behavior of the near future.

Therefore, if the histograms are to guide the adaptivity of our prepaging policy, then those histograms must reflect the recent history that is likely to be a useful guide for the near future. The histograms must be periodically decayed so that the recent past carries greater weight than the distant past. The period of decay should be tied to the paging rate of the system: more paging activity should imply more frequent decay.

In order to couple decay with virtual memory events, we relied on the concept of *virtual memory time*, which is the timescale at which virtual memory events occur [10]. The use of virtual memory time as the basis for this type of decay has been demonstrated in previous studies [16, 22]. Roughly speaking, a clock keeping virtual memory time "ticks" when a process references a page that has been evicted *or* a page that is near eviction. For a k-page main memory, we chose to perform decay for every $\frac{k}{8}$ virtual memory clock ticks; that is, decay would occur only when the most recently used $\frac{1}{8}^{th}$ of main memory had changed its composition. We chose to apply an exponential decay function to the histogram—that is, we multiplied each histogram entry in both histograms by some value λ, where $0 \leq \lambda \leq 1$. We will see, in Section 5, that the dynamically adaptive prepaging policy was not sensitive to the choice of decay values.

4.3 Cost/Benefit Evaluation

The adaptive prepaging policy can use the histograms to compute the approximate costs and benefits for each possible target allocation. As an example, consider a k-page memory. For some target allocation value l, where $0 \leq l < k$, the cost is the sum of the used histogram entries $k - l$ to k, and the benefit is the the sum of the prepaged histogram entries 1 to l. The net reduction in misses is then the difference between the benefit and the cost. Over all possible values of l, the one that yields the best net reduction is chosen as the new target allocation.

This evaluation can be performed with any frequency. We chose to perform it after every histogram decay. However, it is possible that less frequent evaluations are needed to obtain good results.

4.4 Implementation Issues

Although we have not yet implemented our dynamic adaptation mechanism in a real kernel, we must consider the feasibility of such an implementation. Specifically, we are concerned with the overhead introduced by both the gathering of histogram data, the decay of histogram data, and the cost/benefit calculation of new target allocations.

Maintaining the histograms. We have described the histograms as though one entry is kept for each LRU queue position. Maintaining such a histogram implies that the VM system is able to record *every* memory reference. Such accurate maintenance would require unacceptable overhead.

It is for this same reason that real VM systems do not maintain true LRU page orderings. Instead, real systems use LRU approximations such as CLOCK [5] and *segmented queue* [1]. These policies ignore references to some of the more recently used pages. Therefore, implementations of these policies don't require VM system intervention for references to those pages. Because these pages are so recently used, they are in no danger of eviction, and so the inability to provide a relative order among them is of little consequence. In exchange for this inaccuracy, the vast majority of references to those most recently used pages incurs no overhead.

The same principle can be applied in maintaining the histograms. If references to the l most recently used pages of a k-page memory are not tracked, then the corresponding l histogram entries will be empty. The adaptive mechanism would have no information about the number of hits that have occurred to pages in those top l queue positions. Therefore, we must limit the target allocation for prepaged pages to $k - l$; that is, at least l pages must be guaranteed for the caching of used pages.

This restriction is unlikely to inhibit the prepaging mechanism. If the l most recently used pages are referenced so often that tracking their use is prohibitive, then it would be exceedingly unlikely that evicting some of those l pages in favor of prepaged pages would be beneficial. As we have seen in Section 3, ideal target allocations tend to be small, suggesting that limiting the target allocation to $k - l$ pages will not interfere with the successful use of prepaging so long as l is a reasonably small fraction of k.

Notice that the histogram for the prepaged queue can be perfectly accurate. A reference to a page in the prepaged queue may invoke the VM system, but that reference causes the page to move to the front of the used queue. Subsequent references to that page will therefore incur no overhead. The cost of tracking the first reference to a prepaged page is small, as few such events occur relative to the number of references to the most recently used pages.

Histogram decay. For a histogram with n entries, each entry must be multiplied by the given decay factor. Therefore, a single decay operation requires time linear in the number of histogram entries. Since the period of decay is connected to virtual memory time, decays will occur only when a number of pages near or beyond eviction have been referenced. If virtual memory time is passing quickly, it is likely because the system is paging, and so the cost of performing decay is dwarfed by the cost of the disk accesses. If virtual memory time is passing slowly, then the system is not paging significantly, and few decay operations are performed, thus imposing little overhead.

Cost/benefit calculation. The calculation of cost and benefit is also an operation that is linear in the number of histogram entries. A single traversal of the histograms allows us to accumulate, as increasingly large target allocations are considered, the increase in hits to prepaged pages and the increase in misses to used pages. By storing the maximum difference between benefit and cost, the new target allocation can be selected. As with histogram decay, the period of cost/benefit calculation is driven by virtual memory time, and so the calculation will be performed frequently only if paging is occurring.

5. EXPERIMENTS ON THE DYNAMICALLY ADAPTIVE PREPAGER

We simulated our dynamically adaptive prepaging policies in order to ascertain their performance for a set of applications and for a range of main memory sizes. We will see, in this section, that the target allocation can be adapted, on-line, so as to provide benefit when possible and avoid detriment otherwise.

5.1 Policies

Below are the policies that we simulated in order to ascertain the behavior of the dynamically adaptive prepaging policies.

- **Dynamic allocation w/ address-local:** The target allocation is adapted dynamically while address-local is used for prediction and a fixed degree is provided.
- **Dynamic allocation w/ recency-local:** This policy is identical to the previous one except that the recency-local predictor is used.
- **Best fixed-allocation w/ address-local:** Given a chosen degree and address-local prediction, select *off-line* the best, fixed target allocation for the given trace and main memory size.
- **Best fixed-allocation w/ recency-local:** This policy is identical to the previous one except that the recency-local predictor is used.

5.2 Parameters

We used the same traces described in Section 3 for these experiments. For each trace, we also simulated the same range of main memory sizes that were used for the exploratory experiments, where the smallest memory size corresponds to a one thousand fold slowdown due to paging under a demand paging system, and where the largest memory size is the smallest that would contain the entire footprint of the process. The predictors and target allocations to be used are determined by the policies listed above. We simulated a range of degrees from one to eight pages.

5.3 Results and Discussion

Before we begin with some general observations about the results over all of the applications and memory sizes, we must note that the plots shown in Figures 4 and 5 do not show the results from the entire simulated range. We simulated a memory as large as the footprint of the process. However, at the largest memory sizes, a trivially small number of page faults may occur. Because the results in the plots are normalized, small differences in the number of page faults at large memory sizes are exaggerated. Therefore, we exclude from these plots the memory sizes at which fewer than one hundred faults occur under demand paging. We believe that such a small number of faults is of little interest in evaluating memory management policies. Also note that because of space constraints, we show the results of simulating six of the ten applications used, using a degree of 2. These six applications are representative of the whole set, and we will highlight important features in each plot.

We begin by observing the most obvious features of the plots in Figure 4. For all of the applications at most of the memory sizes, page faults are reduced by prepaging. The results shown for **perl** and **photoshp** are representative of those applications not shown. Both predictors yield benefit, but one tends to dominate the other across all memory sizes. The benefit from prepaging increases as the memory size increases, with a substantial middle range of memory sizes where 20% to 40% of page faults are eliminated. The plot for **compress95** shows the application that received the greatest benefit from prepaging, with a 40% to 55% reduction in page faults. This application exhibits such sequential regularity in its accesses that either predictor performed well.

In contrast, the plot for **go** shows the one application in our test suite for which prepaging offered anywhere from modest reductions to modest increases in page faults. At the various smaller and middling memory sizes, there was an increase in misses of 2% to 5%, and at the largest memory size, approximately 17%. Note, however, that at the larger memory sizes, fewer total faults occur, and so deviations from the baseline appear to be larger. Also note that an increase of more than 2% was not observed for any other application. We are unsure of why the reference behavior of **go** yielded this degradation in performance.

Adapting the target allocation. The plots in Figure 5 show the results for **cc1** using the address-local predictor, and **vortex** using the recency-local predictor. These plots allow us to see how well the adaptive mechanism chooses a target allocation. The mechanism should often rival and sometimes improve upon the best possible choice for a fixed target allocation. For **cc1**, we see a case where adaptivity is able to match or improve upon the best fixed target allocation at nearly every memory size. Such improvement implies that the changes in reference behavior during execution require that the target allocation change in response. For **vortex**, the adaptive mechanism merely selects a value near the best fixed allocation and delivers performance comparable to that value. The results for all applications in our test suite were similar to these, with improvements on the best fixed allocation occurring frequently. The adaptivity never substantially increased the page faults in comparison to the best fixed target allocation.

The effect of degree on misses and transfers. Figure 6 shows the results of varying the degree for **vortex** under address-local prediction and **perl** under recency-local prediction. These two are representative of the effect of degree on page faults. Across the majority of the memory range examined, degrees of 1 and 2 yield a useful reduction in misses. However, a degree of 4 yields a significant additional reduction, while degrees of 6 and 8 provide even greater but diminishing improvements.

The choice of degree strongly influences the number of page transfers required for fetches. We can see, in Figure 7, that the total number of transfers can be substantially increased by prepaging with higher degrees. For **acroread**, the number of transfers increases consistently for each increase in degree. We also see that recency-local causes a larger increase than address-local does; recency-local almost always selects candidates for prepaging that have been evicted to the backing store, whereas address-local may frequently select candidates that are already resident. The increase shown here is similar for all of the applications in the test suite.

Although higher degrees provide greater reductions in misses, the increase in transfers implies that smaller degrees may be more desirable in practice. While disk transfer time is a less significant factor than disk latency, the nearly eight-fold increase in transfers shown in Figure 7 for a degree of 8 under the recency-local predictor may be too large an increase. Also consider that clustering small groups of pages on a backing store is likely to be easier than clustering larger groups, again implying that smaller degrees are desirable.

Choice of decay. We performed simulations with a range of values for λ, the multiplicative factor used to decay periodically the histogram entries. Given a range of $0.1 \leq \lambda \leq 0.9$ for this factor, we found it to have little effect on the adaptive mechanism. This lack of sensitivity is shared by the similar adaptivity mechanisms described in [16] and [22]. For all applications at nearly every memory size, the results for different decay values differed by less than 1%. The largest difference was observed for **compress95**, where, for smaller memory sizes and larger degrees, the choice of decay could change the number of page faults by as much as 20%. Even for this application, however, with smaller degrees the difference was at most about 5%. We found that $\lambda = 0.5$ yielded consistently beneficial results.

Comparison to larger page size. It is tempting to think that the advantages of prepaging can more simply be captured by using a larger page size. We simulated demand paging using the LRU policy for 8 KB pages, which are twice as large as the 4 KB pages that we've used for all other experiments. In a few cases (**compress95** and **perl**), the use of 8 KB pages reduced page faults about as well as the prepaging policies that we examined using a degree of 1. For all other cases, demand paging with 8 KB pages performed not only worse than prepaging, but also worse than demand paging with 4 KB pages, often by more than 50%.

With larger pages, the pollution of main memory may be substantial when only a small portion of each page is actually needed. The use of smaller pages can reduce that pollution by increasing the ratio of needed data to unneeded data in each page. Demand prepaging maintains the benefits of the smaller page size while obtaining the improved disk access characteristics of larger pages. While there are other good reasons to increase the page size (such as limited TLB capacities), page fault rate is not among them. We chose not to consider page sizes *smaller* than 4 KB, as no current CPU architecture supports such page sizes.

6. CONCLUSIONS AND FUTURE WORK

Demand prepaging is a conceptually simple idea in which multiple pages are fetched at each page fault. However, the three interacting parameters of target allocation, degree, and predictor dictate whether or not a prepaging policy is helpful or harmful when compared to demand paging. Previous investigations into prepaging have either constricted the possible benefits of prepaging by allocating too little memory to cache prepaged pages, or have assumed that the application can provide substantial hints about future references to the virtual memory system. We have shown that demand prepaging, without hints, may be beneficial to a typical virtual memory system.

We have examined those three interacting parameters in more detail. The choice of target allocation is critical, and its ideal value depends not only on the predictor and the degree, but also on the main memory size, the application, and the reference behavior of the process. The ideal choice of predictor can also change with the application, main memory size, and phase. There is a sensitivity to the degree, but its possible variation is not large, with small degrees

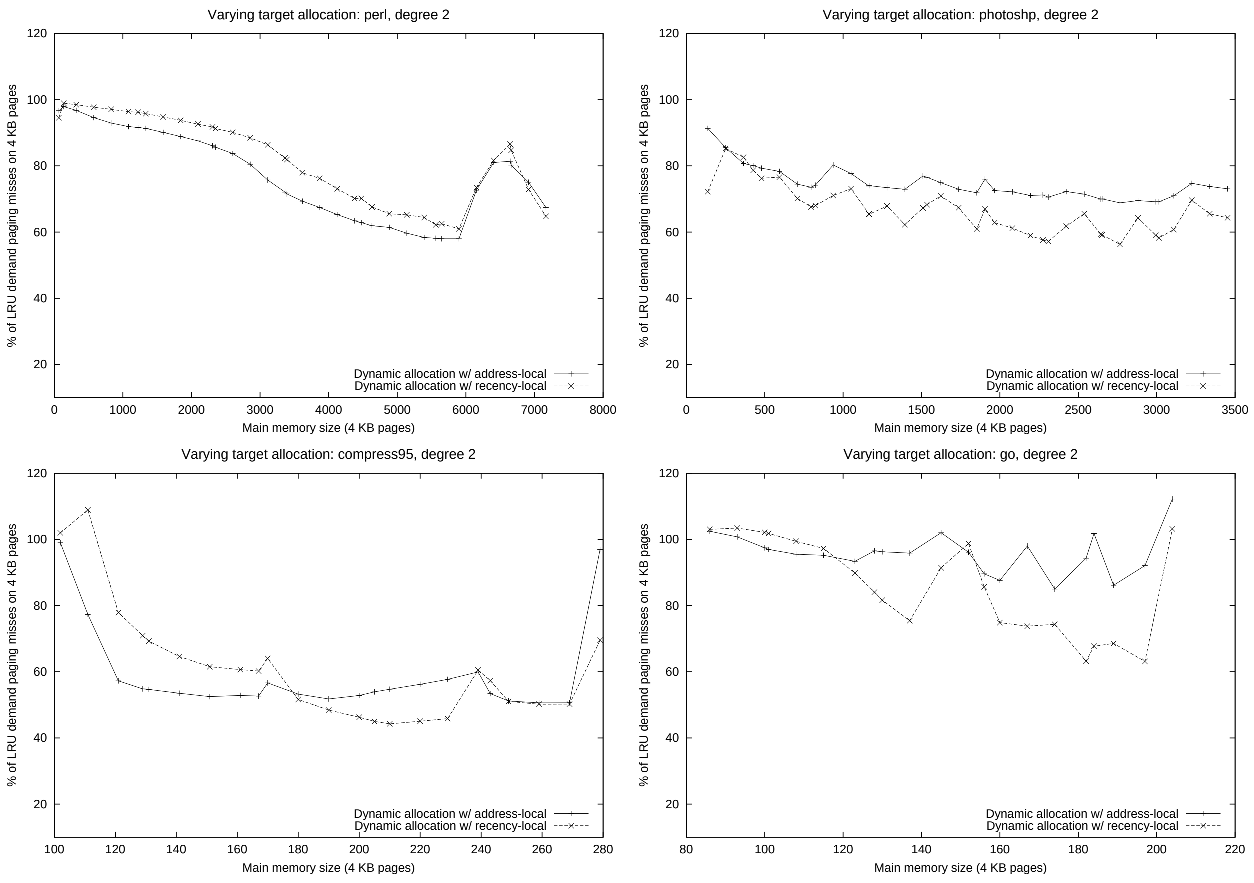

Figure 4: The number of faults under our dynamic prepaging policies relative to demand paging.

(between 2 and 4 pages) yielding good results.

Given a better understanding of these parameters, we have been able to develop a dynamically adaptive mechanism for demand prepaging. This mechanism can adapt the target allocation based on simple cost/benefit calculations derived from the recent reference behavior of the process. The mechanism often yields better results than any single, fixed choice of target allocation could. In other cases, the dynamically adaptive policies nearly matched the best fixed choices of target allocation and predictor.

Based on the results presented here, the use of prepaging without any adaptive mechanism can result in a substantial degradation of performance in some cases. While it would be desirable for real systems to use prepaging to its greatest benefit, it is more critical that such systems do not misuse prepaging and incur an increase in page faults. Use and development of adaptive mechanisms like the one presented here may allow for safe use of prepaging without foregoing its potential benefits.

Future directions. The work presented here suggests a number of future investigations. It is clear that better predictors can be developed, thus improving the performance of prepaging. The adaptivity mechanisms presented could be improved to more effectively avoid harm. Clustering methods should be investigated, using the results here as guidance, either in a real system or in an simulated environment using an accurate disk model. There should be a demonstration that the histograms used for adaptivity could be maintained on-line in a real system.

It will be important to consider prepaging in a multiprogrammed environment. We chose to investigate uniprogrammed simulations first, as it was clear that the parameters and effects of prepaging were not well understood, and examining them in a simplified environment has been revealing. Multiprogrammed workloads introduce substantial complication into VM studies, as the interleaving of CPU and disk scheduling makes the experiments difficult to reproduce and the results difficult to understand. While we expect the adaptive approach used here to be applicable to multiprogrammed workloads, that assumption should be tested, and the referencing characteristics of such workloads should be investigated more thoroughly.

The current results, however, suggest that prepaging with simple predictors and adaptive heuristics can yield a substantial reduction in virtual memory page faults. By amortizing the cost of a disk access across a number of fetched pages, it may be possible for a system to incur a modest amount of paging without suffering a collapse in performance.

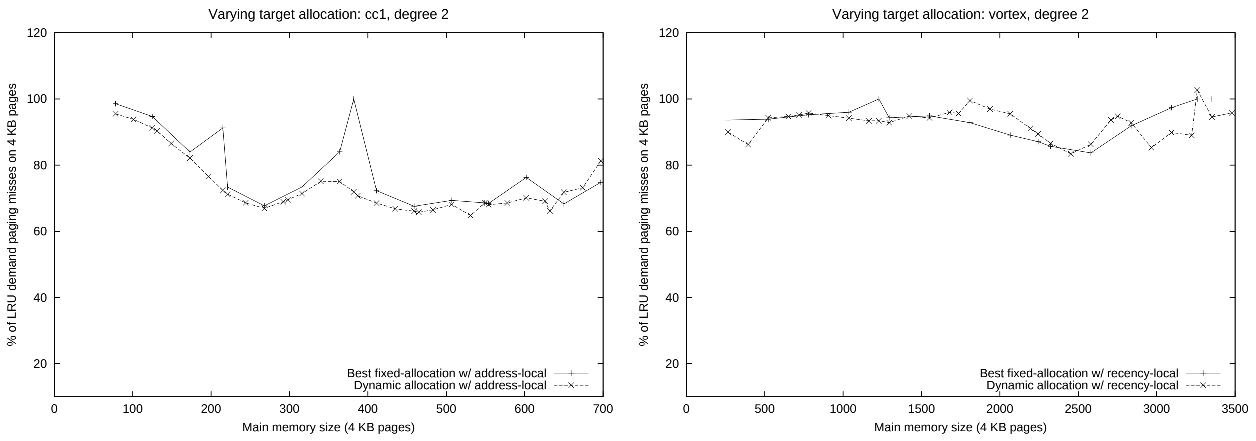

Figure 5: Dynamic adaptation of target allocation sometimes outperforms the selection of the best fixed allocation, and sometimes merely approaches it.

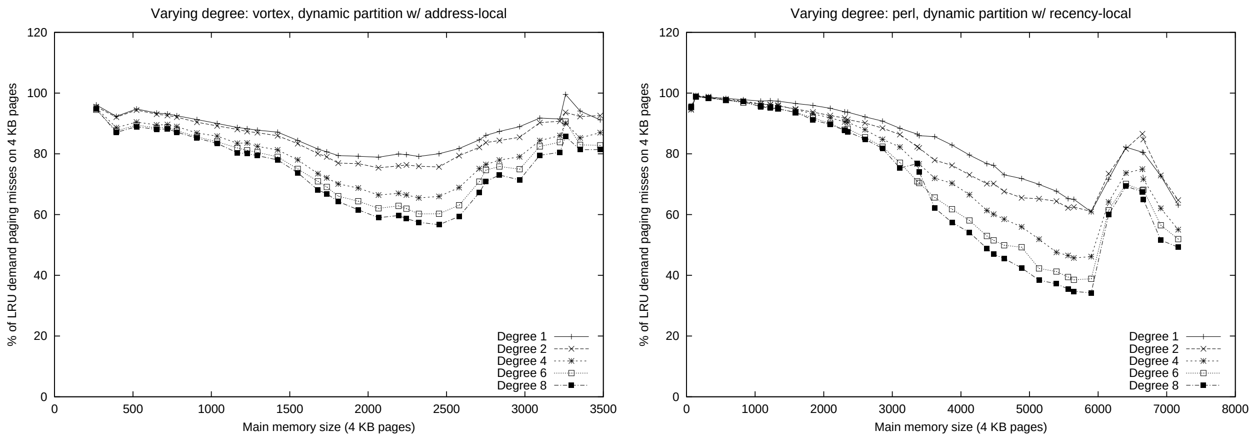

Figure 6: Higher degrees yield greater reductions in page faults, but with diminishing improvements for degrees beyond 4 pages.

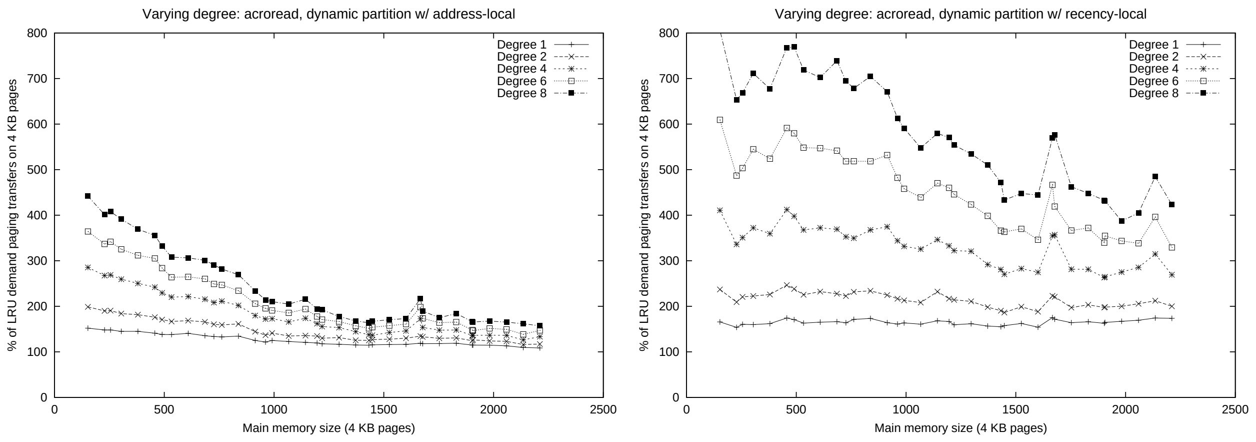

Figure 7: The number of transfers increases substantially with prepaging. The recency-local predictor increases transfer far more than the address-local for this process.

7. REFERENCES

[1] O. Babaoglu and D. Ferrari. Two-level replacement decisions in paging stores. *IEEE Transactions on Computers*, C-32(12):1151–1159, Dec. 1983.

[2] J.-L. Baer and G. R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, SE-2(1):54–62, Mar. 1976.

[3] D. Black, J. Carter, G. Feinberg, R. MacDonald, S. Mangalat, E. Sheinbrood, J. Sciver, and P. Wang. OSF/1 virtual memory improvements. In *Proceedings of the USENIX Mac Symposium*, pages 87–103, November 1991.

[4] P. Cao, E. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *The 1995 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 23, pages 188–197. ACM Press, May 1995.

[5] R. W. Carr. *Virtual Memory Management*. UMI Research Press, Ann Arbor, Mich., 1984.

[6] S. Cho and Y. Cho. Page fault behavior and two prepaging schemes. In *Proceedings of the 1996 IEEE 15th Annual International Phoenix Conference on Computers and Communications*, pages 15–21, March 1996.

[7] K. Curewitz, P. Krishnan, and J. Vitter. Practical prefetching via data compression. In *Proceedings of The 1993 ACM SIGMOD Conference on Management of Data*, pages 257–266, May 1993.

[8] R. N. Horspool and R. M. Huberman. Analysis and development of demand prepaging policies. *Journal of Systems and Software*, 7:183–194, 1987.

[9] M. Joseph. An analysis of paging and program behaviour. *Computer Journal*, 13:48–54, 1970.

[10] S. F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, Aug. 1999.

[11] J. Knapka. Outline of Linux memory management system. `http://home.earthlink.net/~jknapka/linux-mm/vmoutline.html`.

[12] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on windows NT. In *25th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, 1998.

[13] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of The 1997 USENIX Annual Technical Conference*, pages 275–288, Jan. 1997.

[14] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI)*, November 1996.

[15] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, Colorado, Dec. 1995. ACM Press.

[16] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. EELRU: Simple and efficient adaptive page replacement. In *1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 122–133. ACM Press, June 1999.

[17] A. J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, Dec. 1978.

[18] K. S. Trivedi. Prepaging and applications to array algorithms. *IEEE Transactions on Computers*, C-25(9), September 1976.

[19] K. S. Trivedi. Prepaging and applications to the STAR-100 computer. *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*, pages 435–446, April 1977.

[20] K. S. Trivedi. An analysis of prepaging. *Computing*, 22(3):191–210, 1979.

[21] P. R. Wilson, S. V. Kakkad, and S. S. Mukherjee. Anomalies and adaptation in the analysis and development of prepaging policies. *Journal of Systems and Software*, 1994.

[22] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of The 1999 USENIX Annual Technical Conference*, pages 101–116, Monterey, California, June 1999. USENIX Association.