Anomalies and Adaptation in the Analysis and Development of Prepaging Policies *

Paul R. Wilson, Sheetal V. Kakkad, and Shubhendu S. Mukherjee†
Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78751-1182
wilson@cs.utexas.edu

January 27, 1994

Overview

In "Analysis and Development of Demand Prepaging Policies," [1] Horspool and Huberman show that it is possible to design prefetching memory policies that preserve a "stack" inclusion property, much like LRU, allowing them to simulate these policies for all sizes of memory in a single pass through a reference trace. We believe that the details of Horspool and Huberman's algorithms introduce unexpected anomalous properties, however. In particular, their policies are not properly *memory size relative*—events that should only matter to some sizes of memory adversely affect replacement decisions for memories of very different sizes. Slight changes to the algorithms can restore memory size relativity and make them much better-behaved. In addition, we would like to point out that Horspool and Huberman's algorithms actually simulate *adaptive* policies, which may explain some of their unexpectedly positive results. This view suggests that properly size-relative inclusion-preserving policies can be used to systematically evaluate adaptive memory management schemes.

1 Inclusion, Simulation and Memory Size Relativity

The central idea of inclusion-preserving paging policies is to define a "priority" ordering function, which says which pages will be in memory for any given number of pages. At each point in the simulation, this priority ordering determines which pages will be evicted if there is not enough memory. So, for example, an LRU ordering always evicts the Least Recently Used page; for a memory of size n, only the n most recently used pages are kept in memory. The priority ordering of pages is implemented as a list (or "queue") of page numbers, ordered by the recency of the latest touch to each page; a page's position in the queue signifies its priority. Memory references reorder the queue in a way that is independent of any particular memory size, but whose *interpretation* varies depending on what memory sizes are of interest.

1.1 Background—LRU simulation

The goal of trace-driven memory simulation is to simulate the behavior of memories of different sizes, by processing a reference trace (i.e., a list of the memory reads and writes) for an execution of a program. The trace is processed in order, one reference at a time, and state information is maintained to determine the effect of each of these events, i.e., whether it's a hit or a miss in the cache being simulated.

The naive way of doing this would be to simulate each memory size separately. So, for example, if simulating an LRU replacement policy, we could use a variety of *LRU queues*[1] of different lengths. Each LRU queue contains the page numbers of the pages that would be in a given size memory at a each step in the simulation. Each memory reference would move the referred-to page to the head of *every* queue. For each

*This work was supported in part by a gift from Novell, Inc.
†Current address: Dept. of Computer Science, University of Wisconsin at Madison.

[1] Some refer to the priority ranking as a "stack," since elements are "pushed down" to make room for something that is moved to the first position; inclusion-preserving algorithms are therefore often called "stack" algorithms. We prefer to use the term "queue," because it seems more intuitive that pages "cut in line" to skip to the end of a queue, and because it simplifies the explanation of algorithms we will describe later.

queue, the reference counts as a miss if the page isn't present in the queue, and as a hit if it is present. A miss causes the last page in the queue to be "evicted," i.e., pushed out to make room. Pushing a page number off the end of a queue of length n thus represents the eviction of the page from a memory with n pages.

Inclusion-preserving simulation [2] takes advantage of the fact that these LRU queues are very similar, so that a single long LRU queue can be used, rather than a large set of queues managed by the same policy.

The single monolithic LRU queue is, in effect, the union of all finite-length LRU queues. At any point in the simulation, the order specified by an LRU queue of length n is a prefix of the ordering specified by an LRU queue of length $n+1$. (That is, a queue of length n contains exactly the same items as a tail of length n of any larger queue.) When a page is pushed down from position n to $n+1$, that represents its being evicted from a memory of size n—i.e., it reaches the "head" of a queue of that length, and is removed from it.

As shown below, the left end of the queue holds the most recently used page's identifier, in this case B, while the other positions hold successively less-recently-used pages further to the right. (We put the "tail" of the queue at the left and use numbers that increase towards the right, i.e., the LRU direction. This allows us to use a common priority numbering scheme for the tails of all finite-length queues.)

```
page ID:   B  I  H  F  A  G  C  D  J...(LRU)
priority:  1  2  3  4  5  6  7  8  9...
```

Touching A, the 5th-most-recently used page, brings A up from the 5th page in the priority ordering to the first position, implicitly shifting the first four pages down by one position each—page F moves from the 4th position to the 5th position, which was vacated when A was moved, and B, I, and H also shift down one position, so that A can be moved to the head of the "queue." Pages beyond the 5th position (G C D J...) do not have their queue positions changed.

After touching page A:

```
page ID:   A  B  I  H  F  G  C  D  J...(LRU)
priority:  1  2  3  4  5  6  7  8  9...
```

For a memory with at least 5 pages, this is *not* interpreted as a miss, because page A will be in memory. But for a smaller memory, it *is* interpreted as a miss, because moving something from the 20th priority position to the first is interpreted as bringing something into memory—the 5th page will already have been evicted from a memory of size 4 or less.

The important fact here is that when a page is touched at a given priority position, it can be interpreted as a miss for some memory sizes and a hit for others. It is only necessary to count the number of hits at each queue position to find the total misses for *every* queue position. For example, suppose that after processing a (short) reference trace, we have the following counts of hits:

```
queue pos.:    1   2   3   4   5   6   7   8   9  10
# touches:   100  50  30  20  10   5   4   3   3   0...
```

Suppose that there were also 20 touches to pages that were not yet in the queue, i.e., initial touches, and no touches to pages at the 10th position or beyond. For a memory of size 9 or more, the only misses are these 20 initial touches, which fault pages into memory. For a memory of size 8, the three touches to the 9th position would also be misses, for a total of 23. For a memory of size 7, the three touches to the eighth position would also be misses, for a total of 26. For a memory of size 6, we would also interpret the four touches to the 7th position as misses, bringing the total to 30, and so on.

The miss count for a memory of a given size is simply the total number of touches to higher positions (i.e., touches to pages that would have been evicted) plus the number of initial misses. After the initial processing of the trace, we can simply start from the right (least-recently-used) end of the queue position touch distribution, and keep a cumulative sum to compute the misses at each position. For convenience, we can consider the initial misses (touches to pages that are not yet in the queue) to be hits at an "infinite" queue position; no finite queue would contain those pages, so they will always be interpreted as misses.

An LRU replacement policy is really a simple form of adaptation to programs' behavior—when a page is touched, it is cached in fast memory in anticipation that the program will continue to behave the same way for some time, i.e. will continue to touch the page for some time. A single LRU queue satisfactorily addresses behaviors at multiple memory sizes—repeated touches to a few pages over a short period of time make a major difference to replacement policies for very small memories, but much less difference to replacement decisions for larger memories.[2]

[2]This is not to say that an LRU policy for a particular memory completely ignores "higher frequency" behavior. It does remember the order of *last* reference in detail, while ignoring any number of short-duration reorderings between the first and last references to pages.

1.2 Memory-Size Relativity

The key strategy in inclusion-preserving simulation is to abstract away from a particular memory size by using an ordering that is independent of memory size, but to record information that can later be interpreted relative to any particular memory size. With a simple LRU policy, this is easy, because the LRU ordering is already independent of memory size. For some other policies, slight variations must be used to make inclusion-preserving simulation possible.

The LRU policy actually has *two* important properties. One is that the ordering is independent of any particular memory size; that is the inclusion property.

An equally important property of LRU is that the queue reorderings *make sense* for each memory size. Not only is the LRU ordering uniform across memory sizes, allowing efficient simulation, but the effect of this policy is reasonable for any particular memory size. The queue reorderings at a small time scale (near the beginning of the queue) are not detrimental to the effect of the policy for large memories, and in fact are exactly what would be done when simulating only a single size of memory.[3]

We refer to this second property as *memory size relativity*. Where the inclusion (memory size independence) is a property of the simulation itself, memory size relativity is a relationship between the simulation and the actual policy it is intended to simulate.

When simulating policies that are more sophisticated than LRU, it may be necessary to modify the policy to make it to preserve inclusion. That is, if a policy does not have the convenient memory size-independent priority ordering that LRU has, we may be able to devise a good *approximation* of the policy which *does* have the inclusion property. This means that the simulation is not really of exactly the policy one might actually use for a given sized memory, but it is close enough to be a useful approximation and allows simulating all sizes of memory efficiently.

In making the policy inclusion-preserving, it is important to be careful to keep it properly *memory size relative* as well. That is, actions taken to allow the simulation of one size memory should not significantly affect the results for different-sized memories.

The goal of inclusion-preserving simulation is to factor out redundant work that is independent of any particular memory size; in doing so, it is important to ensure that the redundancies are actually there, and not an artifact of an oversimplified model. The changes made to the algorithm should allow the factoring out of redundant work across memory sizes, without introducing spurious behavior for any memory size.

[3]Frequent small-scale reorderings *do* make LRU harder to implement in practice, and most existing implementations are only approximations of LRU. This is rather different from a breakdown of memory size relativity, however. An exact LRU policy isn't doing anything nonsensical for a particular memory size, even if the implementation is doing something unnecessarily expensive.

2 Inclusion-preserving Prefetch Policies

Horspool and Huberman's main innovation is to point out that *any* queue ordering will preserve inclusion if it is defined in a way that is independent of any particular memory size. They show that prefetching policies can work well in this respect. Another innovation is to show that by carefully defining the conditions under which prefetches do occur, *demand* prefetch policies can be simulated—that is, policies which avoid initiating (potentially expensive) data transfers solely for the purpose of prefetching. Such policies only prefetch when there is a *demand miss* to be serviced, i.e., an actual program reference to a page that's not in memory.

2.1 Prefetch-always Sequential Prefetching

Suppose that at some point in a simulation, we have the following queue ordering; we will use alphabetic page identifiers corresponding to pages' address ordering in memory.[4] (We use letters rather than page numbers to avoid confusion with queue position numbers.)

```
page ID:   A  B  I  H  F  C  G  D  J...(LRU)
priority:  1  2  3  4  5  6  7  8  9...
```

Suppose we want to simulate a *one block lookahead* prefetch policy, where a touch to a page not only brings it to the head of the queue, but also brings the next sequential page to the head of the queue. So if we touch page F, we will also bring page G to the head of the queue:

```
page ID:   F  G  A  B  I  H  C  D  J...(LRU)
priority:  1  2  3  4  5  6  7  8  9...
```

This pushes the pages A, B, I and H down to make room for F and G, but pages to the right of F and G

[4]E.g., if page E is page number 1126, page F is page 1127, G is 1128, and so on.

are undisturbed, because the moved pages are "shifted right" to close the gap. Note also that page C, which was *between* the F and G, is moved right only one position. That's because only one page (G) was moved from C's right to C's left. F was already ahead of C, so it doesn't have any effect on C's position.

This queue reordering is inclusion-preserving—when a "prefetched" page is moved to the head of the queue, that *may or may not* actually represent a prefetch for a given size of memory, but the queue reordering itself is independent of memory size. For large memories, it simply represents a reordering of pages that are all in memory already.

Having established that this is an inclusion-preserving policy (and therefore easy to simulate), we must ask if it is properly memory size relative. That is, is the policy we simulate useful in practice, for an actual memory of any particular size?

This policy may introduce some distortion compared to an actual policy for a fixed memory size, e.g., the reordering of in-memory pages may affect the order in which pages are evicted, even though the reorderings are really only for the benefit of smaller memories. It is therefore a breakdown of memory size relativity. A real policy might only bring "prefetched" pages to the head of the queue when a page was *actually* prefetched, which depends on the actual size of memory. The simulated policy brings them to the head of the queue irrespective of which pages are resident in memory.

If we were to use a real implementation of the policy we simulate, a page that is already in memory could be (pseudo-)"prefetched" repeatedly without being touched, and never get evicted despite the fact that it's no longer being used—it is effectively tied to a page that is being used by the policy of "prefetching" the next sequential block. Touches to pages at a small scale may thus affect the eviction ordering—and hence miss rates—for larger memories as well.

As Horspool and Huberman point out, this choice of queue reordering policy may be unorthodox, and is entirely for the purposes of allowing efficient simulation, but it does *not* keep the algorithm from being implementable in practice. We may choose to implement conventional prefetching, rather than the inclusion-preserving variant we devised to make simulation efficient; we can simply accept that the simulation is not exact. Alternatively, we may choose to implement the policy exactly. That is, if the simulations give a positive result, it is certainly possible to implement precisely the same policy in a real system. The simulations thus provide a lower bound on the possible performance of prefetching algorithms.

2.2 Demand Prefetch Policies

It should be noted that we have described a *prefetch always* policy—a page is prefetched any time the previous sequential page is touched. This kind of prefetching may not be practical or desirable for some situations, especially in conventional virtual memories. For the above example, consider a memory of size 6. In that case, the memory is large enough that page F will already be in memory, but small enough that page G, whose priority is lower, will already have been evicted.

Prefetching page G may not be feasible or effective. If page G is prefetched but not touched, a disk seek will be wasted. Even if page G *is* touched after being prefetched, there may not be enough time between the touches to page F and G—the seek will be initiated, but not have time to complete before G is actually touched. Given that main memories are several orders of magnitude faster than disk, it turns out that such nondemand prefetches are often useless. If a process touches F, initiating the prefetch of G, and actually touches a microsecond later, the prefetch won't do much good—it takes milliseconds to seek and load the page, and saving a microsecond doesn't help. There isn't enough *overlap* to perform any useful computation while the disk seek is performed, so prefetching is not helpful.

On the other hand, consider a memory of size 4. In that case, both pages will have been evicted, and touching page F will cause a normal demand fault—the reference will be a simple miss, and a disk seek will be issued to load the page. If page G is prefetched, no second seek is necessary—since it's the next sequential page on disk, it is only necessary to keep reading after page F is loaded. The transfer time is typically much smaller than the seek time, and this kind of prefetch is more likely to be worthwhile.

In this case (*demand prefetching* or *prefetch-on-miss*), the prefetch of page G needn't be overlapped with normal computation to be useful. The second seek is "piggybacked" onto the first one by combining them into one seek—in effect, the second seek is done in parallel with the first, rather than in parallel with normal computation. No real "overlap" (computation time between the issuing of a prefetch and touching the prefetched item) is needed.

Horspool and Huberman therefore wanted an inclusion-preserving prefetch policy that did *not* attempt prefetches that could not be piggybacked onto demand prefetches.

(Note that the inclusion property and the demand prefetching property are mostly independent.)

In virtual memory hierarchies, demand prepaging policies are especially valuable because they do not require extra disk seeks, which might defeat the purposes of prefetching. In other situations, however, non-demand seeks may not be as much of a problem. They do make it difficult to interpret simulation results, however, because it's hard to tell which "successful" prefetches would actually occur far enough in advanced to do any good.

2.3 Demand Prefetching with Inclusion

At first glance, it might seem that demand prefetching would necessarily violate inclusion, because what counts as a miss (on which to piggyback an extra seek) is intrinsically dependent on the size of memory. Horspool and Huberman show that there is not a direct opposition here, though, and show that a policy can preserve both inclusion and the demand-prefetch-only property.

Horspool and Huberman devised demand prefetching algorithms to avoid the problems with interpreting miss rates. Their policy inhibits *any* prefetch that would be a non-demand prefetch for *any* size memory. In the example above, the prefetch of G would not occur, because it is problematic for memories of size 5 or 6. This is a rather conservative policy, because that same "prefetch" would not be a problem for any other memory sizes. For larger-sized memories, both pages would be in memory and no fetches would be necessary at all. For smaller-sized memories, the touch to F would be a demand miss and the fetch of G could be piggybacked on it.

To preserve the demand-prefetch-only property, they simply inhibit any prefetch of a page that is further along the queue than the page actually being touched. This avoids demand prefetches for memory sizes that are between the two queue positions—i.e., for which the touched page is in memory, hence not a miss, but the prefetched page is not in memory.

Note that this policy preserves inclusion at some apparent sacrifice in memory size relativity. Some prefetches are inhibited because they would affect only a few sizes of memory, but the inhibition affects the results for many other sizes of memory. As with the simpler prefetch-always policy, this breakdown of size relativity does not prevent the algorithm from being implementable; if simulations suggest that it will work well, it is possible to implement. As the next section shows, the cost is nonzero and reveals a surprising characteristic of the algorithm.

2.4 Unintended Adaptiveness

Surprisingly, this apparently overly-conservative demand-prefetch-only policy *outperformed* conventional (non-inclusion-preserving) one-block-lookahead (OBL) demand prefetching. Horspool and Huberman's goal had been to modify OBL slightly to make it possible to simulate all memory sizes efficiently, and inadvertently created an algorithm that simply worked better.

We believe that the concept of memory size relativity can help explain why this policy is superior; we must ask what the policy is doing when it inhibits a prefetch, and why it might be a good idea *for the memory sizes it matters to the most.*

Consider the case of a page F and its successor in memory, G. If F is touched and G is prefetched, they are both brought to the head of the queue. Now suppose that G is touched, and then both are not touched thereafter. When G is touched, it is put ahead of F in the queue, because it is touched after F. That is, *the relative order of the two pages reflects the order of their last use.* In this case, G is touched after F. If both of them go unused for a while, they will drift down through the queue, but remain in that order. If F is then touched again, G *will* be prefetched.

Now consider the case in which F is touched, G is prefetched, and only F is touched thereafter—that is, G is prefetched, but not touched. In that case, G will drift down through the queue while F remains active, and the relative order will reflect the fact that G was not used. In this case, Horspool and Huberman's policy will suppress the prefetch of G next time F is touched, in order to preserve the demand prefetch only property.

More importantly, however, this policy *suppresses prefetches if a similar prefetch was unsuccessful last time.* If a page is prefetched and not touched, the policy will inhibit the prefetch next time. In effect, it is an *adaptive* prefetch policy, which uses the queue ordering to record the success or failure of previous prefetches.

When we examine this policy with an eye toward memory size relativity, we realize that a real implementation for a fixed-size memory would have to keep queue ordering information for pages that have already been evicted—if it didn't, it wouldn't know which prefetches to suppress.

This unintended breakdown of timescale relativity turns out to be a properly size-relative approximation of a *different* policy. We refer to this policy as the "fool me once" rule—if you do get tricked into fetching something unnecessarily, then don't fall for that trick next time.

Note that Horspool and Huberman's policy is probably not the *best* inclusion-preserving approximation of the fool-me-once rule. A prefetched page may get touched, but the page that triggered the prefetch may get touched again shortly thereafter, reversing their ordering in the queue and unnecessarily inhibiting future prefetches. A better rule might be to prefetch G if it's *near* F in the queue, independent of their relative ordering. (This would introduce some non-demand prefetches, however.) A simple way to do this would be to only prefetch pages that are to the right in the queue if they're not very far to the right. This would not inhibit as many prefetches, but would still allow bounding of the "error" introduced if non-demand prefetches are not feasible.

3 Reducing Misfetch Costs

Horspool and Huberman also tested several variations on their inclusion-preserving demand prefetching algorithms, intended to limit the cost of bad fetches. Their intuition (which we share) is that most good fetches pay off soon after the prefetch—i.e., prefetched pages are usually touched soon, and ones that aren't touched soon are less likely to be touched at all. If this intuition is correct, it is desirable to evict prefetched pages more quickly than actually-referenced pages; prefetched pages should be kept around "for a while," but then evicted to avoid wasting too much memory on them.

Horspool and Huberman implement this policy by adding a parameter k, which accelerates prefetched pages' eviction from memory. When the priority ordering is altered by bringing a page (or two) to the beginning of the list, that moves other pages down by a page or two. Horspool and Huberman accelerate this movement for prefetched pages, by moving them extra queue positions each time. The intuition is that prefetched pages will be evicted twice as fast as normal pages if k is 2, three times as fast if k is 3, and so on.

4 A Breakdown of Memory Size Relativity

One problem with Horspool and Huberman's formulation is that prefetched pages have their priorities decay at an accelerated rate—intuitively, they move through the prioritizing queue faster than non-prefetched pages. Unfortunately, the rate at which prefetched pages' priorities decay is not properly relative to their position in the queue, and the normal rate of advancement of a page at that position. Consider the normal behavior of the prefetching algorithm without this accelerated motion, and given the following queue ordering:

```
A B D E C H
```

Suppose we touch page B, and prefetch page C, bringing both of them to the head of the queue, thus:

```
B C A D E H
```

Pages A will effectively move down the queue *two* positions, because both page B and page C had been to its right and had moved to its left. Pages D and E will only be moved down one position, because A was *already* to their left beforehand. Page H will not be affected at all, because both pages were already to its left, and their motion does not change its position. The normal motion of a page depends on the page's queue position *relative to the touched and prefetched pages*. The accelerated motion should therefore be relative to this. For an acceleration factor k, pages to the left (of both the touched and the prefetched page) should move $2k$, pages in the middle should move $1k$ positions, and pages to the right should move $0k$ positions, i.e., not at all.

Conclusions

Horspool and Huberman have shown that it is possible to devise inclusion-preserving policies that allow the simulation of sophisticated prefetching algorithms. We believe that this is a potentially important advance in the study of memory hierarchies, and should receive broader attention. Slight modifications to their algorithms can restore memory size relativity, and allow the simulation of more interesting replacement policies.

We hope that we have clarified some important issues in the design of inclusion-preserving memory simulators. In particular, the notion of memory size relativity can be be helpful in designing policies that are both realistic and usefully adaptive. We would also like to point out that these techniques could be generalized to study adaptive replacement as well as adaptive prefetching. (A page's passage through the priority list could be sped up through certain segments, then delayed before entering the next. This could simulate the eviction of pages from some sizes of memory without affecting larger sizes of memory.)

These seem to be promising techniques for efficiently and exhaustively analyzing the performance of many adaptive memory management policies. Such analysis is particularly important for adaptive policies, which by their very nature make it difficult to generalize from a small number of experiments.

Acknowledgements

We would like to thank Nigel Horspool for very helpful comments on an earlier draft of this communication.

References

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

[2] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.