Trace Reduction for Virtual Memory Simulations

Scott F. Kaplan, Yannis Smaragdakis, and Paul R. Wilson*

Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
{sfkaplan|smaragd|wilson}@cs.utexas.edu

Abstract

The unmanageably large size of reference traces has spurred the development of sophisticated trace reduction techniques. In this paper we present two new algorithms for trace reduction — *Safely Allowed Drop* (SAD) and *Optimal LRU Reduction* (OLR). Both achieve high reduction factors and guarantee *exact simulations* for common replacement policies and for memories larger than a user-defined threshold. In particular, simulation on OLR-reduced traces is accurate for the LRU replacement algorithm, while simulation on SAD-reduced traces is accurate for the LRU and OPT algorithms. OLR also satisfies an optimality property: for a given trace and memory size it produces the shortest possible trace that has the same LRU behavior as the original for a memory of at least this size.

Our approach has multiple applications, especially in simulating virtual memory systems; many page replacement algorithms are similar to LRU in that more recently referenced pages are likely to be resident. For several replacement algorithms in the literature, SAD- and OLR-reduced traces yield exact simulations. For many other algorithms, our trace reduction eliminates information that matters little: we present extensive measurements to show that the error for simulations of the CLOCK and SEGQ (segmented queue) replacement policies (the most common LRU approximations) is under 3% for the majority of memory sizes. In nearly all cases, the error is much smaller than that incurred by the well known *stack deletion* technique.

SAD and OLR have many desirable properties. In practice, they achieve reduction factors up to several orders of magnitude. The reduction translates to both storage savings *and* simulation speedups. Both techniques require little memory and perform a single forward traversal of the original trace, which makes them suitable for on-line trace reduction. Neither requires that the simulator be modified to accept the reduced trace.

*This research was supported by IBM, Novell, and the National Science Foundation (under Research Initiation Award CCR-9410026).

SIGMETRICS '99 5/99 Atlanta, Georgia, USA

1 Introduction

Trace driven simulation is a common approach to studying virtual memory systems. Given a *reference trace*—a sequence of the virtual memory addresses that are accessed by an executing program—a *simulator* can imitate the management of a virtual memory system. Thanks to reference traces, experiments on virtual memory management policies can be reproduced in a controlled environment. Unfortunately, these traces can be extremely large, easily exceeding the capacities of modern storage devices even for traced executions lasting only a few seconds. The size of traces impedes both their storage and processing. *Trace reduction* is the compression of reference traces (either lossless or lossy) so that they can be stored and processed efficiently.

There are many existing methods for trace reduction. However, these methods have undesirable characteristics for virtual memory simulation: Some discard so much reference information that the reduced trace introduces significant error into the simulation of common page replacement policies. Other methods make it difficult to control how much information is discarded, and thus what size memories can be simulated accurately. Some methods reduce the storage costs without reducing the number of references and thus the time required to process a trace.

We present two trace reduction methods—*Safely Allowed Drop* (SAD) and *Optimal LRU Reduction* (OLR)—that do not suffer from these deficiencies. Both allow a user to control the degree of reduction by the specification of a *reduction memory size*. SAD is the simpler of the two: it removes references that are guaranteed not to affect the LRU and OPT behavior of a trace, provided that the simulated memory sizes are no smaller than the reduction memory size. Under the same assumption, the OLR algorithm yields the shortest possible trace that can be used for exact LRU simulations in place of the original trace. OLR is useful both because it provides greater reduction than SAD and because its output gives a lower bound for the length of a reduced trace. Both algorithms are efficient in practice, and significantly reduce storage *and* processing costs.

Guaranteeing accurate simulation for LRU and OPT may not seem exciting at first. If the trace were used only with these policies, the simulation could be run only once and the results stored and re-used. Our approach is effective, however, for simulations of *many* virtual memory replacement policies. Nearly all replacement policies used or studied with real workloads are either variants or approximations of LRU in a weak sense that is sufficient for our trace reduction techniques. This similarity of common page replacement policies

is hardly surprising good replacement algorithms should not evict pages that are in current use.

Variants of LRU (e.g., GLRU [FeLW78], SEQ [GlCa97], FBR [RoDe90], EELRU [SKW98]) keep the k most recently referenced pages in memory, even though not all m pages in memory ($m > k$) are the m most recently accessed (as they are in pure LRU). Our approach to trace reduction is applicable in all such cases for a reduction memory size of at most k. Even small values of k (10 to 100) are enough to allow OLR and SAD to achieve reduction factors of up to several orders of magnitude, while guaranteeing exact simulations.

SAD and OLR are also useful when studying *approximations of LRU*. The most prominent approximations are CLOCK and SEGQ (segmented queue—also known as *hybrid FIFO-LRU* [BaFe83] or *segmented FIFO* [TuLe81]). These replacement policies ignore the same high-frequency referencing information that nearly any replacement policy will ignore, and that SAD and OLR discard from traces. This information is ignored not because these are LRU approximations, but because references to recently used pages don't affect replacement decisions, and because hardware often does not support the efficient collection of such information.

We show that the error introduced by SAD and OLR for both CLOCK and SEGQ replacement simulations is small—under 2% in number of faults in most cases. We also compare SAD and OLR to *stack deletion* [Smit77], which is a commonly known technique for removing high frequency reference information from virtual memory traces. Given reduced traces of comparable size created using all three methods, SAD and OLR introduce less error on average into CLOCK and SEGQ simulations.

Additionally, the ability of the SAD algorithm to produce reduced traces valid for exact OPT simulations is a pleasant side-effect: it means that a single trace can be used for all experiments in a virtual memory study. Such studies often compare a new algorithm to LRU and OPT.

2 Background and Motivation

Given the importance of trace reduction, it is not surprising that there has been a wealth of research work on reduction techniques. It is impossible to exhaustively reference all the approaches—instead Section 2.1 presents an overview and Section 2.2 positions our method relative to the most closely related techniques. A good further reference is the recent survey of trace-driven simulation by Uhlig and Mudge [UhMu97].

2.1 Overview of Related Work

Like all data compression, trace reduction techniques are divided into *lossless* and *lossy* approaches. In a lossless approach, the entire trace can be reconstructed from its reduced form, while lossy reduction does not preserve all information in the original trace. Our technique is lossy in nature but guarantees that certain kinds of simulations (namely LRU and OPT simulations) are exact on the reduced traces.

Lossless Reduction. A straightforward approach to lossless trace reduction is to apply standard data compression techniques on a trace. Simple Lempel-Ziv compression results into reduction factors of about 5 for typical traces [UhMu97]. Higher degrees of reduction can be achieved by combining compression algorithms with differential encoding techniques. The best known such instances are the Mache [Samp89] and PDATS [JoHa94] systems, which explore spatial locality in the reference trace to encode it differentially. Subsequently, standard text compression techniques are applied and result into further reduction of its size.

Lossless techniques can be used to reconstruct a trace accurately for all purposes. Nevertheless, the compression ratios achieved are not as high as those possible with lossy trace reduction. More importantly, traces need to be uncompressed before simulation is performed. Thus, the reduction gains of lossless compression do not translate into simulation speedups.

Lossy Reduction. When performing trace reduction, one usually has some knowledge of the future uses of a program trace. Lossy trace reduction techniques attempt to exploit such knowledge so that the trace size is reduced dramatically but enough information is maintained for the intended uses of a trace.

The simplest lossy reduction technique is *blocking*. Blocking replaces references to individual addresses with references to memory pages. Subsequent references to addresses within the same page can then be reduced to a single reference. This reduction does not affect the simulation of *time-independent paging algorithms*—algorithms that do not consider the exact time of each reference in making replacement decisions. Such algorithms are LRU, OPT, etc., but not, for instance, Working Set [Denn68]. Blocking is so widely applicable that it is practically assumed in most simulation work. For the remainder of this paper, when we refer to an *original* trace, we are referring to a blocked trace.

Recent work on trace reduction includes the technique of Agarwal and Huffman [AgHu90]. Whereas most lossy reduction techniques concentrate on the *temporal* locality of a program trace, their approach exploits *spatial* locality and results in an extra significant factor of reduction.

Other trace reduction methods include *trace sampling* and *trace stripping* (e.g., see [Puza85]). Both are better suited for high-speed hardware cache simulations, as they introduce inaccuracy into fully-associative virtual memory policy simulations.

The majority of lossy trace reduction methods, however, are oriented towards virtual memory simulations. These techniques address the same concerns as our algorithms and are directly comparable to them. The next section discusses such related reduction techniques in detail.

2.2 The Value of Our Techniques

Our approach fills a prominent gap in the spectrum of trace reduction techniques. Most existing techniques either do not guarantee accurate simulations or do not achieve the same high reduction factors as our method. We isolate three approaches that stand out as particularly related to ours.

- Smith's *stack deletion* (SD) [Smit77] consists of only keeping references that cause pages to be fetched to an LRU memory of size k. SD is directly comparable to the SAD algorithm. Both techniques are very simple and have similar preconditions: both require that the reduced trace be used with memories no smaller than the memory used for reduction. Nevertheless, SAD guarantees that no error is introduced for LRU and OPT simulations, unlike SD. Smith argued experimentally that the error of SD is small. However, that error is small only if the depth of the stack (i.e., the

size of the memory used for reduction) is much smaller than the simulated memory (typically 20% to 50% of its size). Hence, SAD can use a much larger reduction memory, which will yield greater reduction, and still achieve exact results. Additionally, we show that SD introduces larger error than both SAD and OLR for CLOCK and SEGQ simulations for reduced traces of the same size. In conclusion, SAD and OLR are both safer (i.e., introduce less error) and more effective (i.e., yield smaller traces useful for comparable purposes) than SD.

- The technique of Coffman and Randell [CoRa70] can be seen as an alternative to both SAD and OLR for LRU simulations. Their approach consists of using the *LRU behavior sequence* (i.e., the sequence of pages fetched and evicted) for an LRU memory of size k to perform exact simulations of LRU memories of size larger than k. The behavior sequence is typically very short, even for small values of k. The biggest drawback of the Coffman and Randell approach, however, is that the product of reduction is not itself a trace. For instance, it is not clear how the LRU behavior sequence of a trace can be used for OPT simulations. In the best case, the simulator as well as any other tools (e.g., trace browsers) will need to change to accept the new format. This is a practical burden to the simulator implementors and makes it hard to distribute traces in a compatible form. This is the main reason why this simple technique has not become more widespread. Our OLR algorithm is complementary to the approach of Coffman and Randell: it offers an efficient way to turn the behavior sequence format into the shortest possible trace exhibiting this LRU behavior. Other advantages of our algorithms exist. For instance, SAD is also applicable to OPT simulations and we show that both SAD and OLR introduce little error for simulations of CLOCK and SEGQ.

- Just like our techniques, the reduction method used by Glass and Cao [GlCa97] is applicable to exact virtual memory simulations. Like Coffman and Randell's method, the Glass and Cao technique suffers from needing to modify the simulator to accept the reduced trace format. The modifications are far from trivial, and it can be hard to use the reduced trace information for simulations of policies other than those studied in [GlCa97] (LRU, OPT, and SEQ—an experimental replacement algorithm). Another drawback of this technique is its lack of control over the interesting memory ranges. It is not possible to specify directly the memory sizes for which the simulation should be exact. Instead, the trace filter allows only indirect control over the minimum memory sizes for which the simulation is valid; worse, that minimum size cannot be determined until *after* the trace has been gathered. The method seems to be less efficient than our approach, at least for LRU simulations. We did not have access to the traces used by Glass and Cao in unreduced form, but were able to derive the OLR-reduced form of these traces (directly from the Glass and Cao reduced traces). This was several times shorter than the reduced form used by Glass and Cao, both in terms of absolute size and in terms of significant events. The detailed results of this comparison can be found in [KSW98].

Other applications of our algorithms are possible. Because of its optimality properties, OLR is ideal for the purposes of trace analysis. It provides an estimate of the amount of reordering done inside an LRU memory. This is useful for evaluating whether a trace will behave similarly under LRU and under LRU approximations (e.g., CLOCK or SEGQ implementations). Another possible application of OLR is in trace synthesis. Given any exact sequence of fetched and evicted pages from an LRU memory, OLR can produce a minimum length trace that will cause the same fetches and evictions. This could provide an alternative to statistical trace synthesis techniques (e.g., [Baba81]).

Finally, we should mention that our techniques are complementary to reduction algorithms that exploit different principles. Since the output of our algorithms is itself a trace, other trace reduction techniques can be applied (e.g., [JoHa94, AgHu90]). As we will see, simple file compression of our reduced traces with the `gzip` utility yields much smaller files, further decreasing storage requirements.

3 The Algorithms

3.1 Safely Allowed Drop (SAD)

Full traces commonly contain a large number of references that are ignored by virtual memory replacement policies. These references account for the majority of space required to store a trace, and consume the majority of time required to perform a virtual memory simulation. Safely Allowed Drop (SAD) removes references from a trace that do not affect the order of fetches into and evictions from an LRU memory of some user-specified size.

We will show that SAD allows for exact simulations not only of LRU, but also of OPT. We will also show, in Section 4, that it introduces very little error into the simulation of LRU approximations such as CLOCK and SEGQ.

3.1.1 Finding References to Drop

For any two references to the same page in a program trace, we can define their *LRU distance* as the number of distinct *other* pages referenced between the two references. The idea behind SAD is simple: *For any three references to the same page in a trace, if the LRU distance of the first and third reference is d, then removing the middle reference does not affect the outcome of LRU and OPT simulations on memories of size greater than d.* Section 3.1.3 describes why the elimination of these middle references has no effect on LRU and OPT.

SAD is an application of this observation. The user specifies a *reduction memory size*, k. Then SAD searches the trace from left to right, to find triplets of the above form—references to the same page, such that the LRU distance between the first and third reference is less than k. All middle references of such triplets are eliminated.

Figure 1 shows three references to page `A`. The LRU distance between the first reference A_{first} and the third reference A_{third} is 4, as there are four distinct pages (`B`, `C`, `D`, and `E`) that are referenced between A_{first} and A_{third}. If the memory size chosen for reduction is at least 5, then we can safely drop A_{second} without affecting the results of an LRU or OPT simulation.

Nearly all programs frequently reference pages that were recently used. Due to this temporal locality, references eliminated by SAD constitute the vast majority of references in usual program traces, even for small reduction memories.

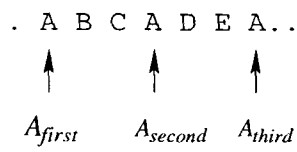

Figure 1: A_{second} can be eliminated because the LRU distance between A_{first} and A_{third} is less than the reduction memory size of 5 pages.

3.1.2 SAD Algorithm Implementation

SAD needs only to determine LRU distances between pairs of references to the same page in order to find middle references that can be eliminated. The search proceeds from left to right, allowing reduction to be performed in a single forward traversal of the original trace.

As the trace is processed, the algorithm maintains an LRU queue of the requested size. It also stores some of the most recently input references from the original trace. By keeping both the LRU queue and a recent history of references, the algorithm can find groups of three references to the same page where the LRU distance between the first and third references is less than the reduction memory size. Therefore, this information is enough to find middle references that can be eliminated.

Although it is necessary to store recent references to find these triplets, the number of references can be bounded. It is only necessary to store at most $2k+1$ of the most recent references in order to find the LRU distance between first and third most recent references to a page.[1] With something like a hash table to help find recent references to pages, performing this reduction is little more than an augmented LRU queue simulation; it can be executed efficiently. For more details, we refer you to our implementation of SAD at <http://www.cs.utexas.edu/users/oops/>.

3.1.3 Exact Simulation of LRU and OPT

If SAD reduces a trace using a k page memory, then that reduced trace can be used for the exact simulation of both LRU and OPT memories that are at least k pages.

Recall the definition of *LRU distance*: Given two references to the same page, the LRU distance between them is the number of other distinct pages referenced between those two references. Therefore, if the LRU distance between two references to a page is less than k, then *that page will not be evicted from an LRU memory of at least k pages.*

First, consider an LRU queue of unbounded length and its contents for both the unreduced and the reduced trace. By dropping references, SAD allows pages to drift further away from the top of the LRU queue, as each page is referenced less often. These pages, however, are guaranteed to be in the first k positions of the queue; each eliminated reference is following by another reference to the same page that is an LRU distance less than k from the previous reference.

Other pages are not adversely affected by removing a reference. Their position in the LRU queue can only be closer to the top for the reduced trace than it would have been for the original one. The only positions in the queue that may have different contents for reduced traces are the ones from 1 to k. Therefore, the results of LRU simulations for memories of size k or larger will be identical for the reduced and the unreduced trace.

We illustrate this argument by examining Figure 1. For a memory of size 5 or larger, A will remain in memory between A_{first} and A_{third}. The middle reference A_{second} has no effect on LRU replacement and if it is dropped, the reference A_{third} will ensure that A is not incorrectly evicted.

SAD-reduced traces also yield exact simulations for OPT memories of at least k pages. Consider again the three references in Figure 1. When OPT must choose a page for eviction, it selects the resident page first referenced furthest in the future. We can show, case by case, how the removal of A_{second} does not affect the replacement decisions made by OPT:

- If OPT is processing references before A_{first}, then the removal of A_{second} will not affect its eviction choices, as A_{first} is the reference that OPT will use to determine whether A is evicted.
- If OPT is processing references between A_{first} and A_{third}, then we already know that fewer than k distinct pages are referenced between those two references to A. Note also that the page currently being referenced is not already in memory (since it caused a replacement) and cannot be a candidate for eviction, making the number of other distinct referenced pages preceding A_{third} less than $k-1$. Therefore, if the memory size is at least k, page A cannot be the one first referenced furthest into the future (because of reference A_{third}). The absence of A_{second} does not affect the replacement decision.
- If OPT is processing references that follow A_{third}, then none of these three references to page A will affect decisions. OPT examines future references to make its decisions, so the missing reference A_{second} will have no effect.

3.2 Optimal LRU Reduction (OLR)

The SAD algorithm obtains significant reduction factors for actual traces. Nevertheless, SAD-reduced traces are not necessarily the smallest for which either LRU or OPT simulations are exact. For instance, consider the reference sequence:

A B C B A C D A B D

Applying SAD with a reduction memory of 3 pages to this trace yields no reduction. Nevertheless, the shorter trace

A B C A D B

has exactly the same LRU behavior as the original for a memory of size 3 or larger. Recall that the LRU behavior of a trace for a memory of size k is the sequence of pairs of pages fetched into and evicted from memory when the given trace is applied. The LRU behavior of the two above traces for a memory of size 3 is:

$\langle A, NF\rangle, \langle B, NF\rangle, \langle C, NF\rangle, \langle D, B\rangle, \langle B, C\rangle$

where the special value NF denotes that the memory is not full and, hence, the insertion of one element does not cause the eviction of another.

The importance of LRU for virtual memory systems has motivated the design of the OLR algorithm for computing

[1] The implementation needs to store at most the two most recent references for those pages in k-page LRU queue, plus a third reference to one of those pages as a triplet is found. If a triplet is found, the middle reference is eliminated, and again only two recent references for that page are stored. Triplets could not possibly be found for other pages, so their recent references need not be stored.

such optimally short sequences. OLR takes a reference trace as input and outputs the smallest trace that has the same LRU behavior as the input for a memory of size k or larger. The output of OLR is only a function of the behavior—two different input traces exhibiting the same behavior for an LRU memory of size k will produce the same output. Hence, the first step of OLR is to simulate the input trace on an LRU memory of size k and derive its behavior sequence. That sequence is the input to the OLR_CORE algorithm shown in Figure 2. Following common terminology, we will often use the term *block* as a synonym for *page* and *touch* as a synonym for *reference*.

Some explanation of the conventions followed in the algorithm description is necessary: The input sequence, *behavior*, is represented as an array for simplicity. The special value LAST signals the end of the sequence. OLR_CORE uses a data structure *queue*, which is an LRU queue augmented with two operations:

- *blocks_after(block)*: returns the set of blocks touched less recently than *block*, but still in the data structure (i.e., within the last k distinct blocks touched). If *block* has the special value NF, the returned set is empty (this is useful for uniform treatment of the boundary case where the structure is being filled up).
- *more_recent*($block_1, block_2$): returns a boolean value indicating whether $block_1$ was touched more recently than $block_2$. If $block_1$ has the special value LOWER-LIMIT, or $block_2$ has the special value NF, FALSE is returned.

Due to space limitations we cannot present an extensive analysis and proof of correctness of the OLR_CORE algorithm. Such an analysis can be found in [Smar98]. Here we will discuss the algorithm at an intuitive level which will hopefully convey some of the insights behind its development.

Recall that OLR_CORE takes as input the sequence representing the behavior of a trace for an LRU memory of size k. This behavior sequence consists of pairs of fetched and evicted pages for the LRU memory. The sequence of references to fetched pages has to be a subsequence of the output of OLR_CORE, as every reference that causes a fetch operation must remain in the reduced trace so that the sequence of fetches is preserved. The purpose of OLR_CORE is to find the minimum set of *extra* references that need to be added so that the sequence of evictions is also preserved. That is, each reference in the reduced trace either causes the fetch and eviction specified by the behavior sequence, *or* it causes a reordering of the simulated LRU queue such that a later fetch will cause the *correct* eviction.

At every point during the algorithm's execution, the *queue* data structure reflects the contents of an LRU queue of size k, to which the output (up to the present point) has been applied. At every iteration of the outer loop of the algorithm in Figure 2 (lines 4-20) the *current* index points to successive elements of the input behavior sequence.

For each iteration of the outer loop, before a reference that causes a fetch is output (by line 18), all pages in the LRU queue that were less recently referenced than the corresponding expected evicted page are touched. The set *must_touch* contains exactly these pages (line 5), and lines 16-17 ensure that they get touched. This way, for a reference causing a "fetch", the least recently touched page in the LRU queue is the corresponding evicted page, as indicated by the input.

Before we describe the essence of OLR_CORE it is useful to briefly review a simple algorithmic problem:

> Given an LRU queue and a desired recency ordering for some of the pages currently in the queue, what are the fewest page references required to reorganize the queue so that the desired ordering holds?

One can show that there is a simple algorithm to produce such a minimal sequence of references:

> Examine the desired ordering in inverse order (i.e., least recent element first). Find the first page that is out-of-order in the queue (i.e., it is less recent than its previous page in the desired ordering). Produce references for this page and all subsequent pages in the desired ordering.

This is exactly the algorithm implemented by the inner loop of the OLR_CORE algorithm (lines 7-15). The LRU queue that is modified is *queue*. The desired ordering is described by the evicted pages in the smallest subsequence of *behavior* that begins at position *current* + 1 and contains events where a page is fetched in the LRU queue and the same page is evicted. The latter condition is detected using the *fetched_in_future* set in the algorithm of Figure 2, which contains the pages fetched after position *current* in the behavior sequence. The *lookahead* variable is used as an index for elements of *behavior* beyond position *current*. When the evicted part of such an element is found in *fetched_in_future* (line 8), the inner loop ends.

In short, the essence of the OLR_CORE algorithm is that for each reference to a page not in the LRU queue, as few as possible extra references to pages already in the LRU queue are produced, so that the recency order in the queue matches the expected eviction order *up to the point where a page is evicted that has not yet been last fetched.* Intuitively, the reason the latter condition is necessary is that the entire LRU queue will need to be reorganized (i.e., every page will need to be touched) between the points that a page is fetched and the same page is evicted. Hence, no benefit can be achieved by "looking further" than the eviction of pages that have not yet been last fetched. No reordering should be done, since all pages will be reordered later, when they are last referenced before one of them is evicted. The above argument conveys parts of the intuition behind the development of the algorithm but does not constitute a proof of overall minimality for its output. A rigorous proof (as in [Smar98]) is quite lengthy, and, thus, beyond the scope of this paper.

Finally, we should note that OLR_CORE is very efficient, so that the main component of the running time of OLR is the LRU simulation performed on the input trace to derive its behavior sequence [Smar98]. That is, OLR execution is about as fast as a simple LRU simulation on the input trace for a memory of size k. The algorithm performs just a single forward pass with bounded look-ahead (at most k elements) and, thus, is ideal for online applications. Our free implementation of OLR can be found at `<http://www.cs.utexas.edu/users/oops/>`.

3.3 Trace Manipulation Issues

In our discussion of SAD and OLR we used a simplified form of reference traces (only containing address information for the page being referenced). Real trace formats may need to contain other information, such as the kind of reference (instruction, read, or write), the instruction causing it, the program counter (or any timer info), etc. Additionally, a trace may need to be re-blocked so that experiments can

```
OLR_CORE(behavior, k)
 1  lookahead ← 0, current ← 0, fetched_in_future ← ∅, previous_evict ←LOWERLIMIT
 2  queue ← ∅(k)
 3  ▷ ∅(k) denotes an empty LRU queue of size k
 4  while behavior[current] ≠LAST
 5        do must_touch ← queue.BLOCKS_AFTER(behavior[current].evict)
 6           lookahead_done ←FALSE
 7           while behavior[lookahead] ≠ LAST and ¬lookahead_done
 8                 do if behavior[lookahead].evict ∈ fetched_in_future
 9                       then lookahead_done ←TRUE
10                       else if queue.MORE_RECENT(previous_evict, behavior[lookahead].evict)
11                               then PRODUCE_REFERENCE(behavior[lookahead].evict)
12                                    must_touch ← must_touch \ {behavior[lookahead].evict}
13                            previous_evict ← behavior[lookahead].evict
14                            fetched_in_future ← fetched_in_future ∪ {behavior[lookahead].fetch}
15                            lookahead ← lookahead + 1
16           for x ∈ must_touch
17               do PRODUCE_REFERENCE(x)
18           PRODUCE_REFERENCE(behavior[current].fetch)
19           fetched_in_future ← fetched_in_future \ {behavior[current].fetch}
20           current ← current + 1

PRODUCE_REFERENCE(block)
 1  queue.TOUCH(block)
 2  OUTPUT(block)
```

Figure 2: OLR_CORE accepts the sequence of fetched and evicted pages for a k page LRU memory, and produces the shortest reference trace that would cause the same behavior for that same memory.

be conducted for different page sizes. Such standard trace manipulation is perfectly compatible with both SAD and OLR. For instance:

- **Re-blocking**: a reduced trace for a reduction memory of size k can be re-blocked for any larger page size and simulations will continue to be accurate for memories of size k or larger (note that the size refers to the number of pages—the actual minimum memory size in Kbytes for which simulations are exact is larger after the re-blocking). This is a consequence of the stack algorithm [MGST70] properties of LRU and OPT.
- **Maintaining Dirtiness Information**: many virtual memory studies measure the cost of writing dirty pages to a backing store upon eviction. Such studies require traces in which each reference is marked as a *read* or *write* operation. Both SAD and OLR can be augmented to tag references with the appropriate operation.

  In order to maintain the dirtiness information about each page in reduced traces, the reduction methods must notice which pages would be modified by a *write* operation while in a k page LRU memory. Since both methods maintain such a memory during reduction, an implementation can record whether a page is dirtied while in that memory. If a page is dirtied while in the reduction memory, then the last reference to that page before it is evicted is marked as a *write* operation. A simulation based on the reduced trace will mark the page as dirty before it is evicted from a k page or larger memory.
- **Maintaining Timing Information**: timing information is trivial to maintain for SAD, since the algorithm only removes references from the original trace. For OLR, where reference reordering may occur, it makes sense to keep time information for references causing a page to be fetched into memory. These are guaranteed to be exactly the same (and, hence, in the same order) as in the original trace.

4 Experimental Results

We applied our trace reduction methods to traces collected both on Windows NT and UNIX platforms. The nine Windows NT traces include the full set of the commercially distributed traces gathered using the utility `Etch` [LCBAB98]. These include well-known Windows NT applications (Acrobat Reader, Netscape, Photoshop, Powerpoint, Word) as well as various other programs (CC, Compress, Go, Vortex). The six UNIX traces (Espresso, GCC, Grobner, Ghostscript, Lindsay, P2C) were gathered using `VMTrace`, our portable tracing tool based on user level page protection; these traces are freely available on our web site. The Windows NT traces were blocked for 4 Kbyte pages so that they would be appropriate for virtual memory simulations. The UNIX traces were generated as references to 4 Kbyte pages.

In this section, we show the reduction factors achieved over a range of reduction memory sizes. We also used reduced traces to simulate both the CLOCK and SEGQ replacement policies. These two policies cannot be simulated exactly using reduced traces, but we show that the error introduced into their simulation is small in practice. We also show that the error introduced is significantly less than with stack deletion [Smit77], a well known reduction method. We chose to simulate CLOCK and SEGQ because they are the two replacement policies most used in real systems. As approximations of LRU, they are similar to many replacement policies that discard information about references to the most recently used pages.

4.1 Reduction Results

Each of the traces was reduced using both SAD and OLR over a range of reduction memory sizes. Recall that the "original" traces are blocked on 4 Kbyte pages, and yet are hundreds of Mbytes to a few Gbytes each. We measured the number of bytes required to store the original trace and each of the reduced traces. Because each reference in these traces is a text representation of the virtual memory page number in hexadecimal, each record comprises at most (and usually exactly) five bytes. Thus, there is a direct correspondence between number of bytes and number of records in a trace.

The plots in Figure 3, show the reductions achieved by SAD and OLR on six of the fifteen original traces. The curves shown plot the reduction ratio achieved as a function of increasing reduction memory size. We chose to show the reduction results from three of the original traces per platform due to space limitations. The remaining programs show similar increase in reduction with memory size, as well as equally high reduction factors.

Note that the reduction factors increase quickly as the memory size grows. The reduction achieved for a particular reduction memory size is a direct result of the locality exhibited by the traced program. Since the vast majority of references are to pages that have been recently used, a small reduction memory can yield large benefits. Note that the size of the OLR-reduced trace is a good measure of program locality: it is the smallest trace that has the same LRU behavior as the original for a memory at least as large as the reduction memory.

Since many virtual memory systems simulate hundreds or even thousands of pages, traces can be made hundreds of times smaller while still being appropriate for experimental studies. Using a reduced trace can allow a researcher to perform simulations that much more quickly, as the simulation time is usually proportional to the length of the input trace.

Also note that SAD achieves reduction factors close to those of OLR. Although SAD is a much simpler algorithm, it provides nearly optimal reduction, while still allowing for exact OPT simulation as well as exact LRU simulation.

It is hard to tell from our plots if high reduction ratios can be achieved for small reduction memory sizes. As we show in the table below, both SAD and OLR perform very well even for very small reduction memories (20 pages for the Windows NT plots and 5 pages for the Unix plots, as the Windows NT programs have much larger footprints).

It is worth noting that our reduced traces can be further compressed by applying lossless trace reduction techniques (for instance, [JoHa94, Samp89]). Even though we did not experiment with any such methods, we found that SAD and OLR reduced traces are highly compressible using standard text compression tools. The next table shows the compression factors achieved by the Unix `gzip` utility on our reduced traces (the ratios shown are "reduced trace size" divided by "compressed reduced trace size"). The results below are not representative for all reduction memory sizes. As reduction memories become larger (and reduced traces become dramatically smaller), compression factors shrink. Eventually, compression ratios become almost as low as 3:1, which is largely an artifact of representing each reference as text. These traces, however, are thousands of times smaller than the originals, and their storage requirements are negligible.

Trace	Reduction memory size	Reduction ratio	
		(SAD)	(OLR)
acroread	20	62.01	75.72
cc1	20	16.12	19.52
compress	20	7.32	8.11
go	20	5.16	6.34
netscape	20	16.76	20.24
photoshop	20	61.06	72.76
powerpoint	20	10.81	12.66
vortex	20	7.04	8.68
winword	20	14.62	18.01
espresso	5	29.03	43.44
gcc	5	3.39	4.31
grobner	5	8.17	10.78
ghostscript	5	9.97	12.26
lindsay	5	8.66	10.82
p2c	5	5.39	6.91

Figure 4: Even for small reduction memories, significant reduction factors can be achieved.

Trace	Reduction memory size	gzip compression ratio	
		(SAD)	(OLR)
acroread	20	31.21	25.48
cc1	20	19.3	18.59
compress	20	17.55	14.46
go	20	13.77	12.25
netscape	20	26.48	21.52
photoshop	20	74.76	64.11
powerpoint	20	30.73	25.08
vortex	20	40.52	42.12
winword	20	38.5	32.74
espresso	5	13.74	14.03
gcc	5	13.6	11.67
grobner	5	11.58	10.36
ghostscript	5	22.2	20.55
lindsay	5	6.8	5.36
p2c	5	9.71	8.69

Figure 5: Reduced traces are often highly compressible.

4.2 CLOCK and SEGQ Simulations

We simulated both the CLOCK and SEGQ replacement policies using traces reduced by SAD, OLR, and Smith's *stack deletion* (SD) methods. The results of these simulations were compared with simulations based on the original, unreduced traces.

We chose these two policies not only because they are so common, but also because they are similar to any page replacement policy likely to be used in practice. They are approximations of LRU because recency information tends to be an excellent predictor of future reference patterns. They also discard information about recently referenced pages because of the hardware available in all machines. If hardware reference bits are supported, CLOCK can be used. SEGQ was designed for machines that did not have such hardware support, yet allowed efficient recency based page replacement.

Our CLOCK simulator simulated a single-hand, two-reference-bit implementation. For a CLOCK-managed memory, there are *reference bits* associated with each resident page. When a resident page is referenced, its primary reference bit is set.

If a non-resident page is referenced, some other page must be chosen for eviction. If we imagine the resident pages

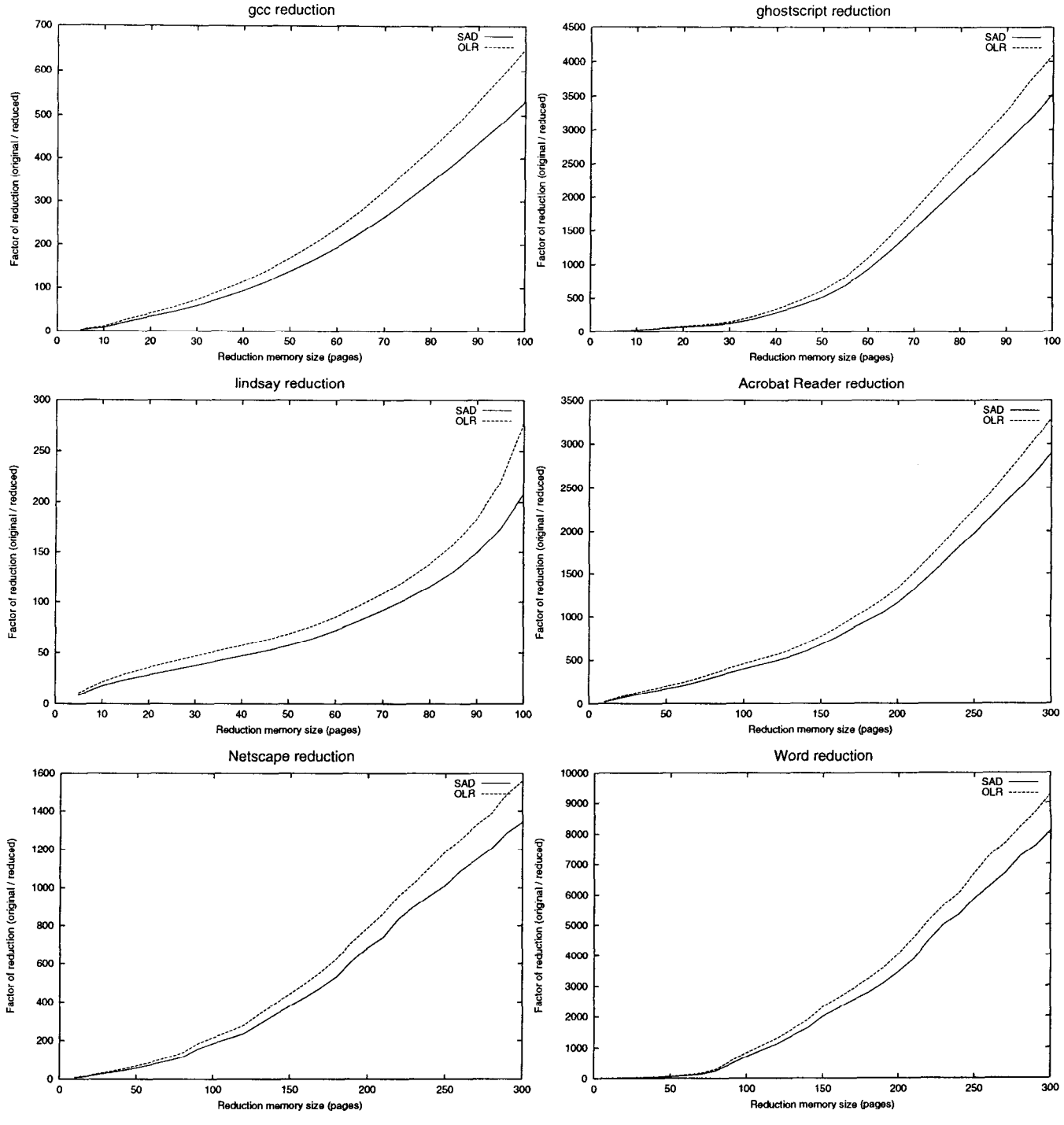

Figure 3: SAD and OLR reduction factors over many reduction memory sizes for six of the fifteen traces. The reduction factors for the traces not shown grow similarly with the reduction memory size.

to be arranged circularly, a *clock hand* sweeps around that circle. If the hand encounters a page with either of its two reference bits set, it shifts the contents of the primary reference bit into the secondary, and then clears the primary. The hand then examines the next page.

When the clock hand encounters a page whose reference bits are not set, that page is chosen for eviction. This mechanism is designed so that CLOCK selects a page that has not been referenced recently. Any recently referenced page is likely to have at least one of its reference bits set.

We also simulated the SEGQ replacement policy (segmented queue—also known as *hybrid FIFO-LRU* [BaFe83] or *segmented FIFO* [TuLe81]) with the original and reduced traces. This replacement policy orders resident pages in two segments. The first segment is a FIFO queue that holds some fixed number of the most recently referenced pages. Pages evicted from the first level are inserted at the front of the second level, an LRU queue. Pages evicted from the LRU queue are evicted from memory.

4.2.1 Error Introduced by SAD and OLR

While SAD and OLR cannot be used to perform exact simulations of CLOCK and SEGQ, we found that little error is introduced into simulation if the ratio of simulation memory to reduction memory is sufficiently large. (The error is defined as the absolute value of the difference in the number of page faults incurred using the unreduced and reduced traces.) In practice, a ratio of 5:1 yields uniformly low error. A ratio of 2:1 also yields small error for the majority of memory sizes, but sometimes introduced unacceptable error in excess of 10%—large enough to lead to erroneous conclusions due to inaccurate results. Programs that occupy a small footprint yielded our largest errors, while programs with a large footprint suffered the smallest errors. Overall, the larger this ratio, the better the results will be.

Due to space constraints, we cannot show the results of each simulation with each reduced trace (many results are shown in the next section in comparison with the stack deletion method). However, we summarize them in a few observations. These observations are valid for simulations in which the simulated memory to reduction memory ratio is at least 5:1, and at least 1,000 paging events occur (a virtual memory study with traces causing fewer faults is unlikely).

- For the vast majority of memory sizes, less than 2% error was observed for both reduction methods.
- Under CLOCK, SAD never introduced more than 3% error. OLR performed slightly worse than SAD on average, and in isolated cases exhibited nearly 10% error.
- Under SEGQ, neither OLR nor SAD exhibited more than 6% error. OLR performed as well as SAD on average.
- The reduced traces sometimes caused too many misses, and sometimes too few. However, there was no pattern nor bias for reduction or increase in miss numbers.
- The smaller the footprint of the program, the larger the observed error. For programs with larger footprints (at least hundreds of pages), error was often near zero for all memory sizes.

It is crucial to note that the ratio between the simulation memory size and the reduction memory size has a large effect on how much error is introduced. Most virtual memory studies are of simulated memories with sizes in the hundreds or thousands of pages. We have shown that large reduction factors can be achieved with reduction memories whose sizes are in the tens of pages. It should therefore be possible to produce significantly reduced traces with a ratio of at least 10:1 to allow for acceptably accurate simulations.

4.2.2 Comparison to Stack Deletion

An often referenced form of trace reduction is Smith's *stack deletion* (SD) [Smit77]. It is interesting to compare SD to our reduction techniques because its value has been demonstrated exclusively through experimental arguments. SD does not guarantee exact simulations, but has been shown to introduce small error into the simulation of replacement policies (namely, LRU, OPT, and CLOCK). We compared SAD and OLR to SD with our suite of fifteen traces and found that our techniques, particularly SAD, consistently yield smaller error.

For each reduction method, we chose a reduction memory size that would yield a reduced trace that was 100 times smaller than the original (that is, the traces for all three methods were approximately the same size). An alternative would be to use the same reduction memory size for each method. This would be unfair for SD since it keeps less information than both OLR and SAD.

We performed CLOCK and SEGQ simulations using each reduced trace. A subset of the CLOCK results are shown in Figure 6. These plots show the percent error (i.e., difference in number of page faults) introduced by SAD and SD on CLOCK simulations. For clarity, we omitted the OLR results (including them would obscure parts of the plots.) The traces from `go` and `grobner` were chosen to represent programs that use a small footprint, with 61 and 228 pages respectively. The programs `gcc` and `ghostscript` occupy medium sized footprints of 450 and 551 pages each. `acrobat reader` and `netscape` are larger footprint programs that use 1914 and 1022 pages, respectively.

For each of the programs, SAD and OLR match or exceed the accuracy provided by SD. Although its results are not shown, on average OLR introduces more error than SAD but less than SD. SAD provides smaller error than SD at almost every memory size.

Note that the leftmost portion of each plot contains large error, as simulations of memories comparable to the reduction memory size are inaccurate. For all of the plots, the error drops significantly around a ratio of 2:1 of simulated memory size to reduction memory size. In most cases, SD reaches a reasonable level of error at a slightly smaller memory size than SAD, as it used a smaller reduction memory size.

Smith claimed that a ratio of 2:1 would be sufficient for experimentation with SD. We found that, with the 2:1 ratio, there is significant error in specific cases. For example, SD introduces more than 30% error into the simulation of `grobner` at a ratio of 4:1. It also introduces more than 35% error into `go` at a ratio of about 3.5:1. SAD and OLR also suffer unacceptably large error at these ratios in isolated cases. Consequently, we recommend a ratio of at least 5:1 for uniformly negligible error.

For SEGQ simulations the results were similar, although less error was introduced on average for all reduction methods. We again selected six traces from the original fifteen, as the results of these are similar to the rest. Note that the error introduced is irregular; the behavior of SEGQ is

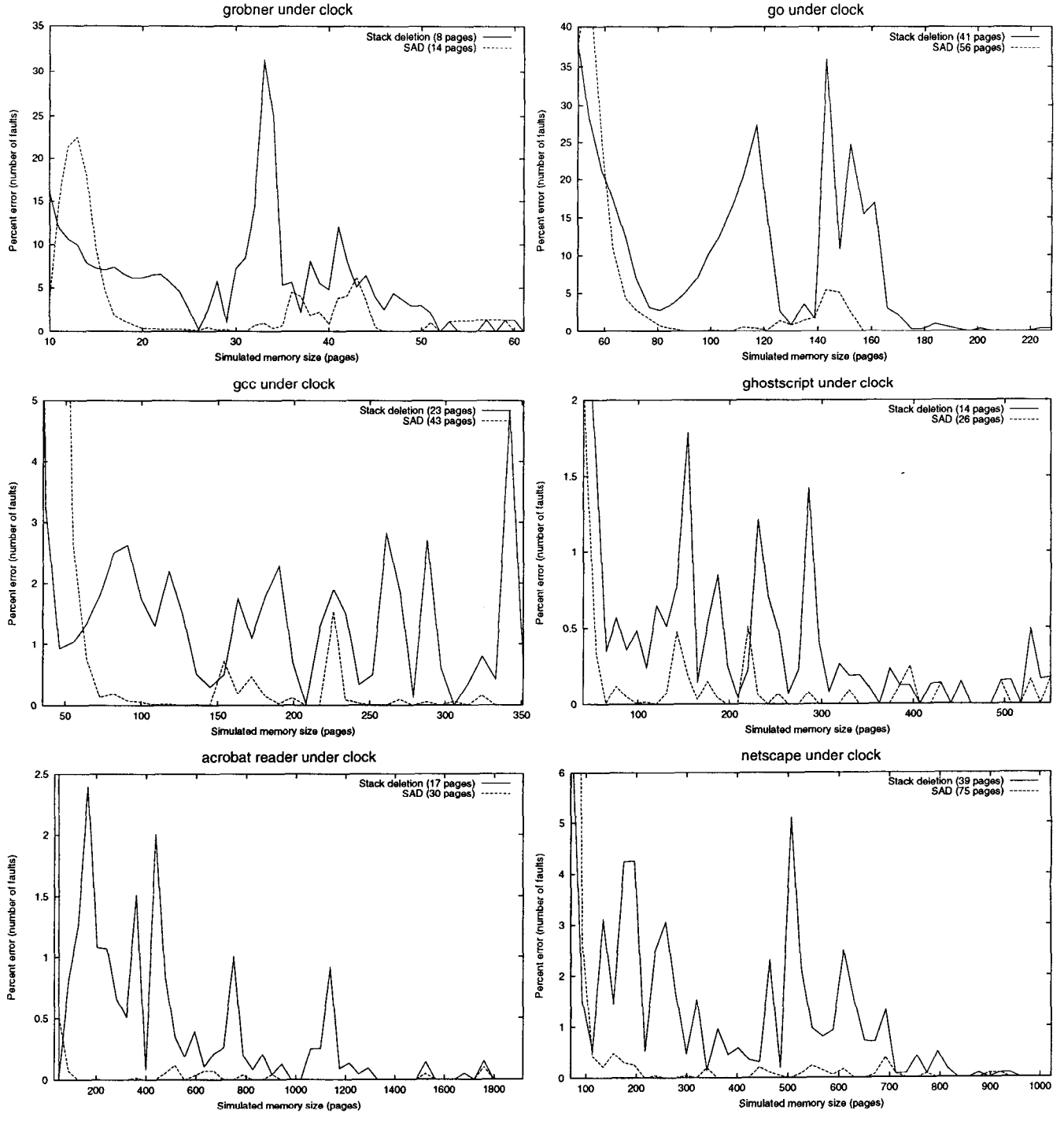

Figure 6: The absolute percent error (by number of faults) introduced into CLOCK simulations by reduced traces from SAD and SD. Notice that reduction memory sizes were chosen so that each reduced trace was approximately 100 times smaller than its original.

dominated by FIFO, which is not a stack algorithm and can produce unpredictably different results with slightly different memory sizes.

In the plots shown in Figure 7, the size of the FIFO segment is fixed, and the percent error is shown for increasing LRU segment sizes. The FIFO segment size chosen for these plots is approximately twice the reduction size for SAD. Thus, the total simulated segmented queue memory is at least twice as large as the reduction memory for both reduction methods.

For both programs, SAD and SD introduce sufficiently small error as to be acceptable for most studies. SAD introduces less error for many memory sizes, although at a few memory sizes SD is the better method. Although it is not shown, OLR performed even better for SEGQ than it did for CLOCK, and its performance was comparable to SAD's. For many of these traces, the error introduced by either method is not significant.

Overall, SAD and OLR would be preferable to SD in practice. While all three introduce small error into simulations, SD introduces slightly more on average, and SAD consistently introduces the least. Further, SAD and OLR both allow for the exact simulation of LRU (and LRU variants like GLRU [FeLW78], SEQ [GlCa97], FBR [RoDe90], EELRU [SKW98]), and SAD allows for the exact simulation of OPT.

5 Conclusions

Storing and processing long memory reference traces is costly. We have proposed SAD and OLR: two new methods for drastically reducing traces to alleviate *both* storage *and* processing requirements. These reduction methods are designed to eliminate information about references to the most recently used pages. Both allow for the exact simulation of LRU memories of a minimum size chosen explicitly by the user. SAD also allows for the exact simulation of OPT memories.

SAD and OLR are invaluable for realistic virtual memory studies. Most studied virtual memory policies are either variants or approximations of LRU. Traces reduced with SAD or OLR provide for accurate simulations with LRU variants (for memories larger than a user-defined threshold). Additionally, we have shown that our reduced traces introduce very little error into the two most commonly used LRU approximations, CLOCK and SEGQ.

We have implemented SAD and OLR, and have made them freely available on our web site. These utilities have been useful to us in our studies, and we invite others to take this portable C++ code and use it in theirs. Both reduction tools can be used off-line with existing traces, or online as traces are gathered.

References

[AgHu90] A. Agarwal and M. Huffman, "Blocking: Exploiting spatial locality for trace compaction", Proc. *SIGMETRICS '90*, pp.48-57.

[Baba81] O. Babaoglu, "Efficient Generation of Memory Reference Strings Based on the LRU Stack Model of Program Behaviour", Proc. *PERFORMANCE '81*, pp.373-383.

[BaFe83] O. Babaoglu and D. Ferrari, "Two-Level Replacement Decisions in Paging Stores", *IEEE Transactions on Computers*, 32(12) (1983).

[CoRa70] E.G. Coffman and B. Randell, "Performance Predictions for Extended Paged Memories", *Acta Informatica* 1, pp.1-13 (1970).

[Denn68] P.J. Denning, "The Working Set Model for Program Behavior", *CACM* 19(5) pp.285-294 (1976).

[FeLW78] E.B. Fernandez, T. Lang, and C. Wood, "Effect of Replacement Algorithms on a Paged Buffer Database System", *IBM Journal of Research and Development*, 22(2) pp.185-196 (1978).

[GlCa97] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior", Proc. *SIGMETRICS '97.*

[JoHa94] E.E. Johnson and J. Ha, "PDATS: Lossless Address Trace Compression for Reducing File Size and Access Time", Proc. *IEEE International Conference on Computers and Communications* pp.213-219 (1994).

[KSW98] S.F. Kaplan, Y. Smaragdakis, and P. Wilson "Trace Reduction for Virtual Memory Simulations", UTexas CS Tech. Report 98-24.

[LCBAB98] D.C. Lee, P.J. Crowley, J.L. Baer, T.E. Anderson, and B.N. Bershad, "Execution Characteristics of Desktop Applications on Windows NT", Proc. *Int. Symp. Comp. Arch. (ISCA) '98).*

[MGST70] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies", *IBM Systems Journal* 9 pp.78-117 (1970).

[Puza85] T. Puzak, *Analysis of Cache Replacement Algorithms*, Ph.D. Dissertation, University of Massachusetts, 1985.

[RoDe90] J.T. Robinson and M.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement", In Proc. *SIGMETRICS* (1990).

[Samp89] A.D. Samples, "Mache: No-Loss Trace Compaction", Proc. *SIGMETRICS '89*, pp.89-97.

[SKW98] Y. Smaragdakis, S.F. Kaplan, and P.R. Wilson, "EELRU: Simple and Efficient Adaptive Page Replacement", in the same Proceedings.

[Smar98] Y. Smaragdakis, "Trace Reduction for LRU-based Simulations", UTexas CS Tech. Report 98-25, revised version available at `<http://www.cs.utexas.edu/users/smaragd/research.html>`.

[Smit77] A.J. Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data", *IEEE Transactions on Software Engineering* SE-3(1) (1977).

[TuLe81] R. Turner and H. Levy, "Segmented FIFO Page Replacement", In Proc. *SIGMETRICS* (1981).

[UhMu97] R.A. Uhlig and T.N. Mudge, "Trace-Driven Memory Simulation: A Survey", *ACM Computing Surveys* 29(2), pp.128-170 (1997).

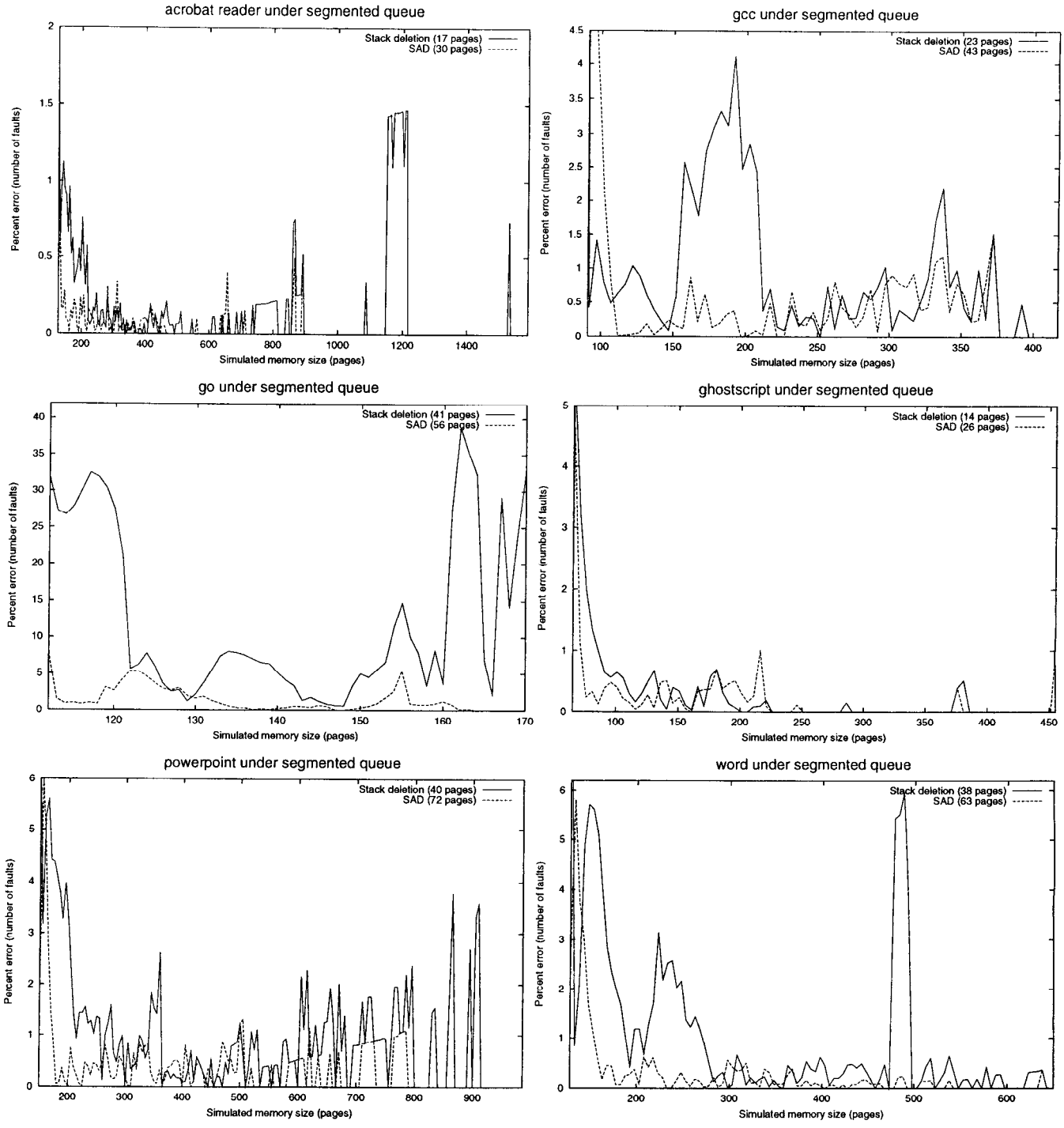

Figure 7: The absolute percent error (by number of faults) introduced into SEGQ simulations by reduced traces from SAD and SD. For each plot, the FIFO segment size is fixed, and the plot shows the error introduced for every possible size of LRU segment that could follow the FIFO segment. The x-axis shows the total memory size obtained by combining the FIFO and LRU segments.