Flexible Reference Trace Reduction for VM Simulations

Scott F. Kaplan
Amherst College
sfkaplan@cs.amherst.edu

Yannis Smaragdakis
Georgia Institute of Technology
yannis@cc.gatech.edu

Paul R. Wilson
University of Texas at Austin
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. Both policies can easily be modified and extended to maintain timing information, thus allowing for exact simulation of the Working Set and VMIN policies. OLR also satisfies an optimality property: for a given original trace and chosen memory size, it produces the shortest possible reduced trace that has the same LRU behavior as the original for a memory of at least the chosen size. We present a proof of this optimality of OLR, and show that SAD, while not optimal, yields nearly optimal performance in practice.

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 vast 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, making them suitable for on-line trace reduction. Neither requires that the simulator be modified to accept the reduced trace.

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. Most commonly, it is assumed that the *Least Recently Used (LRU)* policy or some variant of it will be simulated using the reduced traces, as LRU and its derivatives are most commonly used in real systems. Furthermore, it is assumed that *OPT* (also known as *MIN*) [Bel66], the optimal, off-line replacement policy, will also be simulated using the reduced traces as a baseline. Based on these assumptions, many references can be identified as "insignificant" and thus eliminated from an original reference trace.

However, the existing 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*—that is, a memory size selected at reduction time that dictates the magnitude of the reduction, where larger reduction memory sizes imply smaller reduced traces. Roughly speaking, the smallest memory size that will be simulated using a reduced trace will be a good choice of reduction memory size. SAD is the simpler of the two algorithms presented: 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.

LRU variants (e.g., GLRU [FLW78], SEQ [GC97], FBR [RD90], EELRU [SKW99]) are policies that maintain the most recently used pages in LRU order, but may keep the less recently used pages in some other order. More formally, an LRU variant keeps some number k of most recently referenced pages in an m-page memory, where $(m > k)$. (For pure LRU, all m most recently accessed pages are kept resident.) 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 *LRU approximations*, such as CLOCK and SEGQ (*segmented queue*—also known as *hybrid FIFO-LRU* [BF83] or *segmented FIFO* [TL81]). 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, rather than pure LRU, but because references to recently used pages matter only to replacement on the timescale of much smaller memories than are managed by virtual memory systems. These high frequency references don't influence replacement decisions, nor does the hardware in real systems 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 for most cases. We also compare SAD and OLR to *Stack Deletion (SD)* [Smi77], 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. Further, SD introduces more error (often in excess of 10%) into simulations based on the reference traces used here than the original research on SD would indicate.

Additionally, the ability of the SAD algorithm to produce reduced traces valid for exact simulation of the OPT policy (a.k.a. Belady's MIN) [Bel66] is a pleasant side-effect: it means that a single trace may be used for all experiments in a virtual memory study. Such studies often compare a new algorithm to LRU and OPT.

Finally, SAD and OLR can easily be extended to reduce traces that contain annotations important to different kinds of simulations. The records of a reference trace may contain information such as timestamps, read/write operation indicators, and the referenced data itself. SAD and OLR are easily extended to maintain that information, again allowing for accurate simulations without modification to the simulator. We will show how SAD and OLR can be applied to simulations of a compressed cache [WKS99] and of a variable space allocation policy such as *Working Set (WS)* [Den76].

In a shorter, earlier paper [KSW99] we presented the SAD and OLR algorithms, and provided a comparison to SD. In this paper, we present several extra results. The main result is the proof of optimality of the OLR algorithm. We also present the SAD2 algorithm, which is a supplement to SAD, and we show an experimental comparison of our methods to the trace reduction algorithm used by Glass and Cao in their study on the SEQ replacement algorithm [GC97]. Additionally, this paper contains a more detailed discussion of the use of SAD and OLR on reference traces that contain annotations, as well as the presentation of SAD-WS, which is a variant of SAD that generates reduced traces that can be used for exact Working Set and VMIN simulations.

Road-map: In Section 2, we discuss previous work in trace reduction, comparing our algorithms to existing ones. In Section 3, we provide a detailed presentation of the SAD algorithm. Section 4 provides a description of the OLR algorithm, as well as the proof of its optimality. In Section 5, we discuss the ease with which these algorithms can be modified for traces with different annotations, and follow that with Section 6, where we present the SAD-WS algorithm for dynamic memory management simulations. Section 7 presents experiments comparing SAD and OLR to SD and the Glass and Cao technique. Finally, we conclude in Section 8.

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. Here, we will address the most critical such techniques; for a more thorough survey of existing techniques, see [UM97]. In Section 2.1, we present an overview, and in Section 2.2, we compare our method to the most closely related techniques.

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 (most notably LRU and OPT simulations) are exact on the reduced traces.

2.1.1 Lossless Reduction Techniques

A straightforward approach to lossless trace reduction is to apply standard data compression techniques on a trace. Simple Lempel-Ziv compression algorithms provide reduction factors of about 5 for typical traces [UM97]. Higher degrees of reduction can be achieved by combining compression algorithms with difference encoding techniques. The best known such instances are the Mache [Sam89] and PDATS [JH94] systems, which explore spatial locality in the reference trace to encode it differentially. Subsequently, standard text compression techniques are applied and result in 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. Only a reduction in the number of records in a trace will reduce the simulation time.

2.1.2 Lossy Reduction Techniques

When performing trace reduction, one usually has some knowledge of the future uses of the 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.

The simplest lossy reduction technique is *blocking* [AH90]. Blocking replaces references to individual addresses with references to larger blocks of address space, such as 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 [Den76], which must track every single memory reference. 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.

Blocking is also interesting in that it is exploiting a different kind of regularity than most reduction techniques. Whereas other lossy reducers concentrate on the *temporal* locality of a program trace, blocking exploits *spatial* locality and results in an extra significant factor of reduction.

Many trace reduction methods are kinds of *trace sampling* or *trace stripping* (see [Puz85]). Both are intended for the simulation of high-speed hardware caches; because they introduce inaccuracy into fully-associative memory policy simulations, they are not well suited to virtual memory simulations.

The remaining types of lossy trace reduction methods 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. Because SAD and OPT are also lossy techniques, other work in this area is the most relevant for comparison to our new methods.

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 a few approaches that stand out as particularly related to ours.

- Smith's *Stack Deletion (SD)* [Smi77] only keeps references that cause pages to be fetched into a k-page LRU memory. That is, SD eliminates references to pages that would already be resident in an LRU-managed k-page memory. SD is directly comparable to the SAD algorithm. Both techniques are very simple and have similar preconditions: for both, the reduction memory size chosen when the trace is reduced determines the minimum simulation memory size for which the results will be accurate. However, SAD guarantees that no error is introduced for simulations of LRU and OPT for that simulation memory size, while SD does not guarantee exact results for any replacement policy. For example, SD may eliminate the *last* reference to a page before it would be evicted from a k-page LRU memory, thus allowing an LRU simulation based on an SD trace to evict that page sooner than it would have been if the original trace had been used. Smith experimentally demonstrated that the error introduced by SD is small. However, that error is small only if the reduction memory size 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 will show that SD introduces larger error than both SAD and OLR for CLOCK and SEGQ simulations based on 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.

- Coffman and Randell's technique [CR70] can be seen as an alternative to both SAD and OLR for LRU simulations. Their approach consists of generating the *LRU behavior sequence*—the sequence of pages fetched and evicted caused by some sequence of references—for a k-page LRU memory. These behavior sequences can then be used to perform exact simulations of LRU memories no smaller than k pages. The behavior sequence is typically very short, even for small values of k. The biggest drawback of this approach is that the product of reduction is not itself a reference trace. At the least, the simulator, as well as any other tools (e.g., trace browsers), will need to be modified 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 will 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 [GC97] is applicable to exact virtual memory simulations of some policies. Their technique divided execution into fixed-length segments of instructions. Roughly speaking, at the end of each segment, their method would emit a record for each page that was referenced for the first time in more than one segment, and a record for each page that had no longer been referenced in more than one segment. It was expected that these "in" and "out" records would be interpreted by a simulator that consumed the reduction representation. Like Coffman and Randell's method, the Glass and Cao technique suffers from the need to modify the simulator to accept this reduced format, which is not a reference trace. In this case, the modifications are substantial, and it can be hard to use the reduced trace information for simulations of policies other than those studied in [GC97] (LRU, OPT, and SEQ—an experimental replacement algorithm that itself could not be exactly simulated using Glass and Cao's own reduction method). 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). These OLR-reduced traces were 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 will be presented in Section 7.3.

- Phalke and Gopinath developed *Inter-Reference Gap (IRG) filtering* [Pha95]. For this lossy method, a working set memory that stores those pages touched in the last θ references is simulated. Any reference to a page already in the memory (that is, in that working set) is eliminated in the reduced trace. Therefore, IRG filtering is analogous to SD, except that a WS memory is used instead of an LRU memory. Phalke and Gopinath demonstrate that this method yields significant reductions in trace size. However, IRG filtered traces have severe limitations. For WS and VMIN [PF76], IRG filtered traces yield fully accurate results only for the number of misses.[1] However, full simulations of these policies—simulations in which the order of fetches and evictions is correct—cannot accurately be performed using IRG filtered traces. As a result, information such as the space-time product, commonly used for WS simulations, cannot be obtained. For other policies, such as LRU, IRG filtered traces introduce greater inaccuracy than SD for traces reduced to comparable sizes. We will show that SAD can easily be modified to allow for exact WS and VMIN simulations, just as standard SAD allows for exact LRU and OPT simulations.

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., [Bab81]).

Finally, our techniques are complementary to lossy reduction algorithms that exploit different principles. Since the output of either of our algorithms is itself a trace, other trace reduction techniques can be applied (e.g., [JH94, AH90]). Furthermore, lossless techniques, including simple file compressors like `gzip`, can be applied to our reduced traces to yield much smaller files.

[1] They also note that, with one additional integer for the reduced trace, mean memory sizes can be accurately calculated for WS and VMIN.

3 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 7, that it introduces very little error into the simulation of LRU approximations such as CLOCK and SEGQ. Finally, we will show, in Section 6, that SAD can be adapted to provide exact results for other policies, like Working Set.

3.1 Finding References to Drop

For any two references to the same page in a trace, we can define the *LRU distance* between them as the number of *other* pages referenced between the two references to the same page.[2] The idea behind SAD is simple: *For any three references to the same page in a trace, if the LRU distance between the first and third references is less than k, then removing the middle reference does not affect the outcome of LRU and OPT simulations on memories of size no greater than k.* Section 3.4 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*. SAD searches the trace from left to right in search of 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.

```
... A B C A D E A...
    ↑     ↑     ↑
```

A_{first} A_{second} A_{third}

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 five pages.

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.

[2] In other words, the LRU distance between two references to the same page is the LRU queue position in which the page would be found when the second reference occurred, assuming that the page is not referenced at all between those two references.

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.

3.2 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, reduction memory 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 stored 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. Specifically, 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.

Given 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 [Kap].

3.3 Improvements in Reference Elimination: SAD2

Ideally, we would like a trace reduction algorithm that eliminates *every* possible reference from a given trace without affecting its LRU behavior on a memory of no fewer than k pages. In other words, an *optimal reference eliminating algorithm* is one that would remove exactly those references that do not affect the LRU behavior of a k-page memory, such that the removal of any remaining reference *would* affect the behavior.

SAD is *not* an optimal reference eliminating algorithm. To see that it is not, consider the following reference sequence:

```
A B C B C B A
```

Applying SAD with a reduction memory of 2 pages will yield the following reduced reference sequence:

`A B C C B A`

The middle reference to page `B` was dropped, making adjacent two references to page `C`. Ideally, these two references to `C` should be collapsed into one, but SAD does not perform this particular reduction. Although SAD could be augmented to identify this case, there is a more general technique we can use to eliminate even more references.

The SAD2 algorithm. Within a reference sequence (even after it is reduced by SAD), there may be a pair of temporally local references to some page where no other reference between those two would cause a fault. The first of those two references is not needed, as the second reference can ensure that the page is fetched before the next page fault, and because the second reference will dictate the LRU queue position of that page when that next fault occurs.

SAD2 is the application of this observation, where given such a pair of references, the first is dropped. More specifically, SAD2 examines a trace from left to right, searching for pairs of references to the same page such that:

1. The first reference to the page would require that the page be fetched into a k-page LRU memory, and
2. No reference between the first and the second causes a page fault.

When such a pair has been found, the first reference is dropped, thus forcing the second reference to become one that will cause the page to be fetched into the k-page LRU memory. To illustrate this algorithm, consider the following reference sequence that, given a reduction memory size of 3 pages, SAD cannot reduce:

`A B C B A C D A B D`

Notice that there are two references to page `C`. The first one will cause a fault, as `C` would not yet be resident in a k-page LRU memory. However, between the first and second references to `C`, there are references to pages `A` and `B`—both pages that would be resident, and would not cause page faults. Therefore, the first reference to `C` can be dropped, leaving the second `C` to cause the fault, leaving the following, reduced sequence:

`A B B A C D A B D`

Note, however, that while this sequence is shorter than the original, it still contains other references that can safely be dropped. The elimination of the first reference to `C` has revealed that there is a pair of references to `B`, one of which is superfluous. Reapplication of SAD2 yields the following reference sequence:

`A B A C D A B D`

Note that in the original reference sequence, we could not safely remove the first reference to `B` because between the first and second references to that page was a reference to `C`—a reference that would cause a page fault. However, the first application of SAD2 *delayed* the first reference to `C` and its corresponding fault. Thus, with the intervening, faulting reference eliminated, the two references to `B` could be safely collapsed. In general, an application of SAD2 may reveal opportunities for further application of either SAD or SAD2. For best results both algorithms can be applied repeatedly until they reach a fixpoint.

Limitations of SAD2. While SAD2 is the application of a simple rule, its computational requirements are greater than those of SAD. As we have seen, SAD2 may have to be re-applied multiple times, as each application may reveal opportunities for further reduction. Therefore, SAD2 cannot easily be used as an online algorithm to eliminate as many references as possible.

Because of this limitation with SAD2, we will not evaluate it further in this paper. We present it to demonstrate that SAD does not eliminate all possible references, and that the further elimination of references requires substantially more computation. We do not even claim that SAD and SAD2, when combined, constitute an optimal reference eliminating algorithm. We will see, in Section 7, that SAD alone realizes nearly all of the possible reduction as determined by OLR, which itself can exceed the reduction of an optimal reference eliminating algorithm. If a simple, easily modifiable algorithm is needed, then SAD is an excellent choice. If further reduction is of critical importance, then OLR, which can synthesize the shortest trace possible and be applied online, is a better choice than using both SAD and SAD2.

3.4 Exact Simulation of LRU and OPT

If SAD reduces a trace using a k-page reduction memory, then that reduced trace can be used for the exact simulation of both LRU and OPT memories that are at least k pages. We will provide arguments for the exactness under both of these policies.

Exact LRU simulation. First, 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.*

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 front 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 followed 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.

Exact OPT simulation. 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.

That SAD is applicable to OPT is no surprise: OPT could be described as the *Least Soon Used (LSU)* policy that keeps a queue of pages ordered from most to least soon used, just as LRU keeps an queue of pages ordered from most to least recently used. *LSU distance* could be defined just as *LRU distance* is. LSU is simply the forward-looking variant of LRU, and the elimination of inconsequential references obeys the same rules.

4 Optimal LRU Reduction (OLR)

The SAD algorithm obtains significant reduction factors for actual traces. Nevertheless, SAD is a reference elimination algorithm, and reduced traces from such algorithms may not be the smallest for which either LRU or OPT simulations are exact. For instance, consider the reference sequence:

$$\texttt{A B C B A C D A B D} \tag{1}$$

For an LRU memory of $k = 3$ pages, the *behavior* of this sequence is:

$$\langle \texttt{A}, NF\rangle, \langle \texttt{B}, NF\rangle, \langle \texttt{C}, NF\rangle, \langle \texttt{D}, \texttt{B}\rangle, \langle \texttt{B}, \texttt{C}\rangle, \textsc{Last} \tag{2}$$

The above behavior sequence consists of pairs of elements $\langle x, y\rangle$, where x is being fetched into the memory, and y is being evicted from the memory. The special value LAST signals the end of the sequence, while NF denotes that the LRU memory is not full and, hence, the insertion of one element does not cause the eviction of another.

SAD would not be able to eliminate references from sequence 1 with a 3-block memory.[3] Although repeated application of SAD and SAD2 would eliminate some references, in this section we are interested in an optimal solution. More specifically, we define the *LRU trace reduction problem* as follows: *Given a reference sequence, find a shortest sequence that yields identical behavior for a k-block LRU memory.* Re-stated, the problem becomes: given a behavior sequence, find a shortest reference sequence that has that behavior.

Note that the reference sequence needs to be at least as long as the behavior sequence (as it contains all references that cause interesting behavior to take place), and typically it is significantly longer. For instance, in our above example (sequence 2) it is *not* enough to take the first part of each element of the behavior sequence:

$$\texttt{A B C D B} \tag{3}$$

The reason is that the above sequence does not have the desired behavior: when block `D` is referenced, block `A` is evicted from memory instead of block `B`. Thus, we need extra reference(s) that will re-organize the memory blocks without causing evictions. It is easy to confirm that the following sequence has the behavior of sequence 2 when $k \geq 3$: (Indeed this sequence is a solution to the LRU trace reduction problem for sequence 2 and $k = 3$.)

$$\texttt{A B C A D B} \tag{4}$$

We designed and implemented OLR: an algorithm that produces solutions to the LRU trace reduction problem. In this section, we will analyze the problem, introduce OLR, and prove that it solves the LRU trace reduction problem. We need to warn the reader that despite our best efforts, the description and proof of optimality of OLR

[3] We use the terms *block* and *page* as synonyms—a page defines the block size of interest for virtual memory studies.

remain rather tedious. (Nonetheless, the description in this article is much simpler than that of our previous, more formal algorithm statement and proof [Sma98].)

4.1 Background and Observations

Before we present an algorithm for the LRU trace reduction problem, we will examine some characteristics of traces that will help us understand the algorithm later. The concepts introduced here require careful attention, as it is difficult to understand OLR or its proof otherwise.

For a given reference trace and a queue size k, the *annotated LRU trace* consists of a complete description of the results of applying the reference trace to an LRU queue of size k. For instance, for the reference sequence 1 (seen earlier) and $k = 3$, the corresponding annotated LRU trace is:

0	1	2	3	4	5	6	7	8	9	10
$\langle \texttt{A}, NF\rangle$	$\langle \texttt{B}, NF\rangle$	$\langle \texttt{C}, NF\rangle$	$\langle \texttt{B}, none\rangle$	$\langle \texttt{A}, none\rangle$	$\langle \texttt{C}, none\rangle$	$\langle \texttt{D}, \texttt{B}\rangle$	$\langle \texttt{A}, none\rangle$	$\langle \texttt{B}, \texttt{C}\rangle$	$\langle \texttt{D}, none\rangle$	LAST

We will call the entries of an annotated trace *LRU events*. Each entry has the form $\langle x, y\rangle$, where x is the referenced page, and y is the page evicted from the k-page LRU queue or the special value *none*. If y is *none*, then x must have already been in the LRU queue, and so no eviction was necessary. If the LRU events where y is *none* are removed from an annotated trace, we get the LRU behavior sequence for the reference trace.

Solving the LRU trace reduction problem means computing the smallest set of LRU events to add to a given behavior sequence to yield a valid annotated LRU trace. The behavior sequence can be generated by basic LRU simulation from an original reference trace, and the annotated trace can be translated back into a reference trace by keeping only the first element from each pair that composes an event.

Consider a sequence of LRU events. We will use the term *interval* for any of its consecutive subsequences. Two concepts that are important for our later development are those of a *run* and a *tight run*:

Definition 1 (run) *A* fetch-evict run *(or just* run*) is an interval in an LRU event sequence, such that:*

1. *It begins with a pair $\langle x, y\rangle$ and ends either with a pair $\langle z, x\rangle$ or with* LAST.
2. *No other LRU event in the run has block x as its first element.*

Intuitively, a run describes the behavior of an LRU queue between the point where an element is *last* referenced before it will be evicted, and the point where the element is evicted. We will describe intervals (and runs) using the starting and finishing indices in the sequence where they occur. A run (s_0, f_0) (that is, a run starting at index s_0 and finishing at index f_0) will be said to *contain* another run (s_1, f_1) iff $s_0 < s_1$ and $f_1 < f_0$.[4]

[4]Note that the strict inequality in the definition implies that runs that are terminated by LAST do not contain one another.

Definition 2 (tight run) *A* tight fetch-evict run *(or just* tight run*) is a run that contains no other runs.*

In our previous annotated trace example, there are five runs: $(3,6)$, $(5,8)$, $(7,10)$, $(8,10)$, and $(9,10)$. All five runs are tight, and this is not a coincidence—otherwise this would not reflect a legal LRU execution: a block would have to be evicted without being the least recently used one for a run not to be tight.

Lemma 1 *All runs in an annotated LRU trace are tight. Any sequence of event pairs such that all its runs are tight is an LRU annotated trace.*

Proof: Immediate from the definition of LRU and the definition of a run. □

Continuing our example, now consider the behavior sequence (2) from Section 4 (reproduced below with indices):

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ \langle \mathtt{A}, NF\rangle & \langle \mathtt{B}, NF\rangle & \langle \mathtt{C}, NF\rangle & \langle \mathtt{D}, \mathtt{B}\rangle & \langle \mathtt{B}, \mathtt{C}\rangle & \textsc{Last} \end{array} \tag{5}$$

Not all runs in this sequence are tight: for instance, run $(0,5)$ contains both runs $(1,3)$ and $(2,4)$. To derive an annotated trace from a behavior sequence, we need to add "extra" references to ensure that all runs are tight. In particular, if one run contains another run, then it must be the case that the block corresponding to the outer run must be referenced during the inner run. The following lemma captures this relationship between outer and inner runs:

Lemma 2 *If a run (s_0, f_0) of a behavior sequence contains another run (s_1, f_1), then, in order to produce an annotated trace, a reference to the page referenced at s_0) needs to be added between the references in positions s_1 and f_1 of the behavior sequence.*

Proof: From Lemma 1 and inspection of the possibilities: The containment of a run in another can change only if the outer run shrinks, and the outer run will shrink only if the "last reference before eviction" for the corresponding block moves closer to the eviction event. □

We will call this process of adding a reference to tighten an outer run *clipping* the outer run. Clipping a run produces a new run that is a suffix of the original.

In the example of sequence 5, run $(0,5)$ contains runs $(1,3)$ and $(2,4)$. Lemma 2 tells us that there needs to be a reference to the page referenced at index 0 (that is, `A`) between indices 1 and 3, as well as between 2 and 4. A reference to `A` after position 2 satisfies both constraints, as seen in sequence 4. The corresponding annotated trace is:

$$\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \langle \mathtt{A}, NF \rangle & \langle \mathtt{B}, NF \rangle & \langle \mathtt{C}, NF \rangle & \langle \mathtt{A}, none \rangle & \langle \mathtt{D}, \mathtt{B} \rangle & \langle \mathtt{B}, \mathtt{C} \rangle & \textsc{Last} \end{array} \tag{6}$$

Lemma 2 is important because it gives us *constraints* for the extra references needed to turn a behavior sequence into an annotated trace. The constraints of a block b can be described as intervals (s, f), meaning that a reference to b must exist in the OLR output between the references in positions s and f of the input. OLR effectively provides a minimal set of references that satisfy these constraints.

4.2 The OLR Policy and Algorithm

As we just saw, the LRU trace reduction problem is equivalent to finding the smallest number of extra references that, if added to a behavior sequence, will make all runs tight—that is, they will turn the behavior sequence into an annotated trace. We propose that the following policy yields such an optimal solution:

OLR: Examine *in order* each fetch-evict event, e, in the behavior sequence. At each one, apply the following rules:

1. **Late clipping step:** Clip every run that contains the one ending at e by adding an event $\langle x, none \rangle$ **before** e (where x is the block corresponding to the outer run). For each outer run that must be clipped, the events added before e can be placed in any order.

2. **Early clipping step:** If the event e is the beginning of a tight run, clip all runs ending after the end of that tight run and before the end of the next tight run (or at the end of input, if the end of the next tight run is the end of the input), in the order that they end. The clip is accomplished by inserting for each such run an event $\langle x, none \rangle$ **after** e.

(Note that the late clipping step adds references before the event examined, while the early clipping step adds references after the event. Thus the two steps do not conflict—they can be applied in any order.) Let us see the application of the two steps in examples.

First we will examine a scenario where the early clipping step is necessary for optimality. The early clipping step is applied at the beginning of a run (specifically a tight run), and exists to eagerly clip sequences as early as possible so that the clipping will not cause a previously tight run to become non-tight. As an example, consider the following behavior sequence:

$$\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \langle \mathtt{A}, NF \rangle & \langle \mathtt{B}, NF \rangle & \langle \mathtt{C}, NF \rangle & \langle \mathtt{D}, \mathtt{B} \rangle & \langle \mathtt{E}, \mathtt{A} \rangle & \langle \mathtt{B}, \mathtt{C} \rangle & \textsc{Last} \end{array} \tag{7}$$

Here, the tight run $(1,3)$ is contained by the non-tight run $(0,4)$. Also notice that there is another tight run in this sequence, $(2,5)$. Now consider what will happen if, instead of using the early clipping step, we clip $(0,4)$ by applying the late clipping step at event 3, thus forming the new sequence:

$$\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \langle \mathtt{A}, NF\rangle & \langle \mathtt{B}, NF\rangle & \langle \mathtt{C}, NF\rangle & \langle \mathtt{A}, none\rangle & \langle \mathtt{D}, \mathtt{B}\rangle & \langle \mathtt{E}, \mathtt{A}\rangle & \langle \mathtt{B}, \mathtt{C}\rangle & \textsc{Last} \end{array} \tag{8}$$

With the new event 3 in this sequence, there is now a run $(3,5)$ corresponding to block `A`. However, this run is now contained by the run $(2,6)$ on block `C`, which had previously been tight in behavior sequence 7. While we could now clip this non-tight run by applying the late clipping step again, we would have inserted two new events. Alternatively, if the early clipping rule is applied to behavior sequence 7 at event 1, which is the beginning of the contained run on block `B`, then no previously tight run is made non-tight; the sequence is made into an annotated LRU trace by adding only one event:

$$\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \langle \mathtt{A}, NF\rangle & \langle \mathtt{B}, NF\rangle & \langle \mathtt{A}, none\rangle & \langle \mathtt{C}, NF\rangle & \langle \mathtt{D}, \mathtt{B}\rangle & \langle \mathtt{E}, \mathtt{A}\rangle & \langle \mathtt{B}, \mathtt{C}\rangle & \textsc{Last} \end{array} \tag{9}$$

Intuitively, the early clipping step is applied exactly to ensure that the newly clipped runs are as long as possible (since they are clipped as early as possible) so that we avoid having them contained in previously tight runs (which would then themselves require clipping).

The late clipping step, on the other hand, ensures that long sequences (ones that do not end before the end of the next tight run) are clipped as late as possible so that the clipped runs do not end up containing other runs. For a scenario where the late clipping step is necessary for optimality, consider the following behavior sequence for a 3-page memory.

$$\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \langle \mathtt{A}, NF\rangle & \langle \mathtt{B}, NF\rangle & \langle \mathtt{C}, NF\rangle & \langle \mathtt{D}, \mathtt{B}\rangle & \langle \mathtt{E}, \mathtt{C}\rangle & \langle \mathtt{F}, \mathtt{E}\rangle & \textsc{Last} \end{array} \tag{10}$$

This sequence has two non-tight runs: $(0,6)$ and $(3,6)$. The $(0,6)$ run contains three different tight runs: $(1,3)$, $(2,4)$, and $(4,5)$. The $(3,6)$ run contains only the $(4,5)$ tight run. Applying OLR to all events in forward order results into an application of the late clipping step on event 3. This means that an event $\langle A, none\rangle$ needs to be added before event 3 of the above sequence. (Two more events $\langle A, none\rangle$ and $\langle D, none\rangle$ will be added by the early clipping step after position 4 of sequence 10.) The intuitive reason why we need to clip run $(0,6)$ late (at position 3) and not earlier (at position 1) is that in this way, the resulting clipped run does not end up containing run $(2,4)$.

The end result of applying OLR on sequence 10 is:

$$\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \langle \mathtt{A}, NF\rangle & \langle \mathtt{B}, NF\rangle & \langle \mathtt{C}, NF\rangle & \langle \mathtt{A}, none\rangle & \langle \mathtt{D}, \mathtt{B}\rangle & \langle \mathtt{E}, \mathtt{C}\rangle & \langle \mathtt{A}, none\rangle & \langle \mathtt{D}, none\rangle & \langle \mathtt{F}, \mathtt{E}\rangle & \textsc{Last} \end{array} \tag{11}$$

Note that the late clipping step of the OLR policy ensures that the output has the same behavior as the input: all non-tight runs are tightened at the point right before their contained run ends.

Before we analyze why the OLR policy is optimal, let us note that it can be implemented by an efficient algorithm. The algorithm simulates an LRU queue on which the reduced trace is being applied. To detect the condition of the late clipping step (runs containing the one ending at the current event, $e = \langle x, y \rangle$) the queue needs to return the set of blocks that are less recently accessed than y. If the queue (of size k) is implemented as an annotated self-balancing tree, the complexity of this operation is $O(\log k + s)$, where s is the size of the returned set. Similarly, testing whether an event is the beginning of a tight run is an $O(\log k)$ complexity operation, since it can be reduced to testing set membership of the evicted block in the set of blocks fetched since the beginning of the previous tight run. In total, an algorithm based on these ideas has a running time of $O(l \cdot logk + n)$, where l is the size of the input behavior sequence, and where n the size of the output reduced trace. (Note that the inequality $l \leq n \leq l \cdot k$ holds.) Considering that the LRU simulation (required to produce the behavior sequence from the original trace) has a complexity of $\Omega(n \cdot logk)$, the running time of OLR is negligible.

Finally, an algorithm implementing OLR needs only $O(k)$ space. All data structures used have at most k elements, and although the algorithm needs to "look ahead" in the input, it needs to do so only until the end of the next tight run, which will always be at most $2k$ events from the current event. This modest storage requirement makes the algorithm ideal for online implementations (i.e., reduction can take place while a trace is being produced).

4.3 Optimality of the OLR Clipping Policy

To show that the OLR policy outlined previously indeed yields the smallest number of extra references, we will first develop a lower bound on these extra references. The bound will be algorithmic—that is, we will show an algorithm to compute a lower bound on the extra references required to tighten a given behavior sequence. Then we will prove that the OLR policy adds references with a one-to-one correspondence to the extra references prescribed by the lower bound.

A Lower Bound: As seen earlier, Lemma 2 can be applied to all pairs of runs contained within one another to give a set of *constraints* for the necessary extra references. The constraints have an interval form: a constraint (s, f) on block b means that a reference to b must exist in the OLR output between the references in positions s and f of the input.

We can compute the least number of references that can satisfy all constraints for a single block with a greedy algorithm. More generally, the problem is: Given intervals $(s_0, f_0), \ldots, (s_j, f_j)$, compute a smallest set of points S, such that for each (s_i, f_i), there exists $p \in S$ with $s_i < p < f_i$. This problem is almost identical to the classic *activity-*

selection problem, as stated in [CLR89], p.330: Given *activities* represented as intervals, compute the maximum-size set of non-overlapping activities.

A reference satisfies constraints $(s_0, f_0), \ldots, (s_j, f_j)$ iff it belongs in their intersection, i.e., the interval $(\max\{s_0, \ldots, s_j\}, \min\{f_0, \ldots, f_j\})$. (That is, the reference will fall within all of the given intervals if it is between the latest starting point and the earlier ending point of those intervals.) A greedy algorithm (henceforth called *the lower bound algorithm*) computes the minimum set of references required to satisfy these constraints:

- Considering the constraints in increasing starting index order, compute maximal intersection intervals—i.e., the intersections of the next j constraints such that they that have a non-empty common intersection, but no common intersection exists for the next $j+1$ constraints.

All of the constraints in an intersection interval computed by the algorithm can be satisfied by a single reference. The proof that this greedy algorithm computes the minimum set of extra references is by simple induction. It is essentially identical to Theorem 17.1 in [CLR89].

For our purposes, this algorithm yields a lower bound on the number of extra events that need to be added to tighten all non-tight runs involving the same block in a behavior sequence (i.e., to solve the LRU trace reduction problem). In fact, we will show that OLR matches the lower bound exactly.

To see why the OLR algorithm and proof is more complex than just applying the above greedy algorithm, consider that there is no guarantee that all necessary extra references can be added without introducing new runs that cause new constraints that affect the outcome of the lower bound algorithm. Avoiding interference by newly introduced events is the essence of the OLR algorithm, and especially of its early clipping step.

Our next observation is that the outcome of the lower bound algorithm does not depend on constraints between non-tight runs.

Lemma 3 *The intersection intervals computed by the lower bound algorithm, if constraints between non-tight runs are removed from its input, are also a valid solution for the original input.*

Proof: If a run r_1 contains a non-tight run r_2, there must be a tight run, r_3 contained by r_2, and, hence, also by r_1. Any reference satisfying the constraint between r_1 and r_3 will also satisfy the constraint between r_1 and r_2. Additionally, since we are only removing constraints of the input and the algorithm computes maximal intersection intervals, the output cannot contain more intersection intervals, and thus is a minimal solution for the original problem, as well. □

Similarly, we can show that the OLR policy only adds references at the beginning or end of tight runs. This property is part of the definition of the "early clipping step" of OLR, but it is also true of the "late clipping step".

Lemma 4 *The OLR policy's late clipping step is only applied when the current event is the end of a tight run.*

Proof: Let us call r_1 the run ending at the current event, and r_2 the run that the late clipping step is used to clip (r_2 contains r_1). If r_1 is not tight, it contains a tight run, r_3. In that case, r_2 also contains r_3. But then r_2 would have been clipped at the end of r_3 by the late clipping step. That point is within r_1, which means that r_2 does not contain r_1, which is a contradiction. Therefore, r_1 must be tight. □

OLR Optimality: We can now show how the OLR policy is linked to the outcome of the lower bound algorithm. First it is useful to describe informally some insights on the behavior of OLR. The early clipping step looks ahead and detects when a run r contains the current tight run but not the next one. In this case, according to the lower bound algorithm, r can be clipped anywhere within the current tight run. The early clipping step clips r right after the beginning of the current tight run because otherwise the run resulting from the clipping may be small enough to be contained by another run and thus change the existing constraints. Also, all other runs that, like r, contain the current tight run but not the next one are clipped in an order that guarantees that they are tight (i.e., they do not contain one another). The late clipping step is used to match the "greediness" of the lower bound algorithm we examined. It only clips runs at the very last instant (right at the end of their contained tight run) so that the same reference can satisfy as many constraints as possible.

Consider the constraints induced by non-tight runs in a behavior sequence. The proof that OLR matches the output of the lower bound algorithm consists of two parts. First, we show that OLR adds events to the behavior sequence in such a way that newly introduced constraints are already covered by existing constraints (i.e. satisfying the existing constraints will also satisfy the new ones). Then, we show that every event added by OLR corresponds to exactly one intersection interval in the current form of the trace (with the extra events already added by OLR), as computed by the lower bound algorithm. Since the additions of events by OLR do not affect the intersection intervals, the extra events of OLR also correspond one-to-one to the original intersection intervals, thus matching the lower bound.

Theorem 1 *Imagine applying the lower bound algorithm every time OLR processes an event of the input sequence. If OLR adds an event that introduces a new constraint then the references (computed by the lower bound algorithm) to satisfy the pre-existing constraints will also satisfy the newly introduced one.*

Proof: Consider first the case of events added by the early clipping step. Since the runs are clipped by this step in the order that they end, no new constraint can be created between two newly clipped runs. (That is, no newly clipped runs can contain each other since they begin in the same order as they end.) The only change in the input of the lower bound algorithm is that the clipped runs may now be contained in other runs, hence yielding new constraints. Nevertheless, every run r_1 clipped by the early clipping step has to end between the end of the current tight run, r_2, and the next tight run. Thus, if r_1 is contained in another run, then so is r_2. Since r_1 is clipped *immediately* after the beginning of r_2, the output of the lower bound algorithm does not change: the intersection of intervals will remain the same, since the new constraint contains an existing one.

Now consider the case of events added by the late clipping step. An interesting observation is that the late clipping step only clips non-tight runs that will remain non-tight after clipping them! Let us call r_1 the run ending at the current event (which is tight according to Lemma 4), and r_2 the first tight run starting after the current event. Every new run r created by using the late clipping step has to contain r_2—otherwise it would have been clipped by application of the early clipping step at the beginning of some earlier tight run. (Here is this last argument in more detail: Recall that the early clipping step clips all the runs ending between the ends of the next two tight runs. Since r_1 is a tight run, every run containing r_1 and ending before the end of r_2 will be clipped inside r_1 by the early clipping step—either at the beginning of r_1 or at the beginning of a tight run between r_1 and r_2, so the late clipping step can only be applied to runs ending after the end of r_2. Hence, every run clipped by the late clipping step contains r_2, and since r_2 begins after the clipping point, the newly created run r will also contain it.) Thus, r is not tight and constraints caused because r is contained in another run do not change the outcome of the lower bound algorithm (according to Lemma 3). □

Now we are ready to finish the proof of optimality for the OLR policy by showing that each extra reference it adds corresponds to a different intersection interval computed by the lower bound algorithm.

Theorem 2 *The extra references added by OLR correspond one-to-one to the intersection intervals computed by the lower bound algorithm.*

Proof: Consider three subsequent tight runs (when all tight runs are ordered by starting point), r_1, r_2, and r_3. At the beginning of r_1, OLR clips all runs containing r_1 but not r_2 using the early clipping step. All runs containing both r_1 and r_2 are clipped either at the beginning of r_2 by a similar application of the early clipping step or at the end of r_1 by the late clipping step. The former is the case if r_1 and r_2 have a non-empty intersection (i.e., r_2 begins before r_1 ends) and the containing run does not contain r_3. Otherwise the latter happens. Both cases conform to the approach of the lower bound algorithm: a containing run is clipped inside the latest tight run possible, in order to satisfy as many constraints as possible with a single reference. Additionally, no two extra references could correspond to the same intersection interval, as the algorithm clips runs on a need-basis: if the containing run has already been clipped, it is not clipped again. □

This concludes the proof of optimality of OLR. The policy matches the lower bound in terms of extra references added, and, thus, produces a smallest set of extra references that can be added to a behavior sequence to turn it into an annotated LRU trace. The annotated LRU trace corresponds to the reduced trace that solves the LRU trace reduction problem.

5 Modification for Annotations and Re-Blocking

For a trace reduction algorithm to be useful in practice, several realistic concerns have to be addressed. These may have to do with maintaining additional event information (e.g., how can the algorithm maintain page dirtiness when the only dirtying reference is the one that is being removed?), with stability under re-blocking (e.g., is the reduced trace safe for simulations with a twice as large page size? Is the reduced trace optimally small in that case?), etc.

These concerns can be addressed easily with SAD and OLR. The algorithms require at most modest modifications to deal with dirtiness information, timings, page images, etc. We have implemented versions of SAD and OLR for several of these scenarios and used them extensively in our own virtual memory experiments. This section discusses some of the issues in more detail.[5]

5.1 Re-blocking

As discussed in Section 2.1.2, most reference traces are blocked to exploit spatial locality. Not all simulations are performed on memories that use the same block size. Often, a trace will be blocked using the smallest block size that the researchers anticipate they will need. Later, those traces will be re-blocked on a larger block size, where the larger size is a multiple of the smaller size. This situation is common for virtual memory systems, where pages can be as small as 512 bytes, and as large as 16 KBytes.

A reduced trace produced with a reduction memory of size k can be re-blocked for a larger page size and simulations will continue to be accurate for memories of size k or larger. It is important to note, however, that the memory size k refers to the number of pages, and not the number of bytes in the memory. The actual minimum memory size in bytes for which simulations are exact is larger after the re-blocking. The ability to re-block a reduced

[5]We concentrate on the SAD algorithm more thoroughly in this section because its simplicity allows for easier modification. We also describe how OLR can be analogously modified, even though we do not show in as much detail how those modifications would be performed in the cases presented.

trace of this kind is a consequence of the stack algorithm [MGST70] properties of LRU and OPT. Note also that no optimality guarantees are preserved after re-blocking: an OLR-reduced version of a trace derived with a certain page size does not remain optimally short for a larger page size.

5.2 Maintaining Annotations Needed at Eviction

In our discussion of SAD and OLR, we have assumed the most simple form of a reference trace—one for which each record contains only the address information for the referenced page. Real trace formats may need to contain additional information in each record, such as the operation that required a reference (i.e. an instruction, read, or write operation), the exact instruction causing that operation, the program counter (or any timer information), etc. For some trace-driven simulations, this information becomes relevant only at the moment that a page is evicted from the simulated memory. Until an eviction occurs, the information provided by such annotations must simply be carried along with associated pages. Upon eviction, the simulator acts on the information it has maintained. Here we will provide examples of this kind of information, and describe how both SAD and OLR can easily be made to maintain that information for fully accurate simulation of evictions.

Dirtiness: 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 maintain these annotations such that the simulated number of evicted pages that are dirty is unaffected by the reduction.

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.

For SAD, performing this task can be described in terms of the triplets identified by the algorithm (as described in Section 3). Assume that each record in the trace has a read/write annotation. For each triplet found within a given k page reduction window, if the middle reference is annotated as a write operation, then the last reference must be annotated as a write operation. The middle reference would have dirtied the page while it was in the k-page memory; since that reference is being dropped, the last reference must become a dirtying reference, thus ensuring that the page will be dirty before it is evicted.

Implementing this modification for SAD requires only that the annotations be read and written, that space be made to store the annotations for each record, and that the "forwarding" of annotations from dropped, middle references to last references be performed. As a point of reference, we performed these changes to our implementation of SAD within a few hours of work.

Page images for compressed caching: In a *compressed caching virtual memory system* [Wil91, Dou93, WKS99, Kap99], main memory is divided into two distinct levels. The first level contains pages in their normal, uncompressed form, which the second level contains pages that have been compressed. Pages are evicted from the uncompressed level into the compressed level, and then from the compressed level to the backing store.

Simulation of compressed caching includes the compression of pages as they are evicted from the uncompressed cache, and then the decompression of pages as they are fetched into the uncompressed cache. In order to accurately simulate the cost and effectiveness of the compression, the simulator must be able to compress and decompress the data contained in those pages. Therefore, reference traces for compressed caching contain actual images of the page data in every trace record.

Just as with dirtiness information, these page image annotations are used when a page is evicted. For both SAD and OLR, we need to associate each page in memory with its most recently read page image. When a page is evicted from memory, its page image at the time of eviction is the page image annotation that should appear in the reduced trace with the last reference to that page, preceding its eviction. Therefore, when a simulation is performed with the reduced trace, the last reference prior to an eviction will be accompanied by the correct page image.

Under SAD, keeping this annotation updated is trivial. When a triplet is found and a middle reference dropped, the annotation with that reference can safely be dropped as well. The last reference in that triplet will already contain the correct page image annotation. As a point of reference, we added support for page images to a SAD implementation within an hour.

Maintaining Timing Information: Timing information can be critical to some simulations, such as those that simulate the dynamic memory management of multiple processes whose execution is interleaved by a scheduler. This timing information is critical for simulating the behavior of the scheduler itself. Timestamps may represent wall-clock time, CPU cycle time, instruction time, and reference time, for instance. SAD and OLR can be modified to maintain these timestamp annotations.

Timing information is trivial to maintain for SAD, since the algorithm only removes references from the original trace. The records that are not dropped still contain their same timestamps, ensuring the references causing fetches or evictions are marked with the appropriate timing information. For OLR, where references are synthesized, timestamps from the original trace can be maintained and attached to the synthesized records that correspond to fetch-evict events (i.e., page faults). Since fetches and evictions are guaranteed to occur in the correct order when processing the reduced trace, the timestamps for each fetch will be correct.

6 Exchanging Reference Distance for LRU Distance: SAD-WS

So far we have concentrated on simulations of replacement (or *demand-paging*) policies. Such policies evict a page to disk only when memory is full and a new page needs to be fetched. Nevertheless, some memory management policies evict pages spontaneously, e.g., by responding to the passage of time between fetches and evictions. Working Set (WS) is such a policy, as it evicts pages as reference time passes, regardless of whether or not any of those references have caused a fetch to occur. For these policies, more significant changes to the trace reduction method are needed. Here, we will describe modifications to SAD that allow for exact WS simulations. We will call the modified algorithm *SAD-WS*. We will also address the VMIN policy, which is the forward-looking, offline counterpart to WS. We will show that, just as OPT simulations are exact for standard SAD, so too are VMIN simulations with SAD-WS.

Recall that SAD simulates a k-page LRU memory, and detects triplets of references to the same page that all fall within an LRU distance of k of one another. This principle does not apply to WS, because WS does not base its eviction decision on the LRU distance between subsequent references to a page. Instead, WS evicts a page based on the *reference distance* between subsequence references to a page. SAD-WS correctly detects these reference distances, and finds references that can be dropped.

The WS policy is simply stated as follows: *Those pages referenced within the last τ references are kept resident.* As with standard SAD, the references that are safe to drop are those that merely re-order the resident pages, and have no effect on the fetch or eviction order. Thus, we will first address the problem of finding those references in a trace that can safely be dropped.

Reduced traces are needed for WS and VMIN simulations at least as much as they are for LRU and LRU-like policies. If the only result needed were the total number of faults for a given trace and given value of τ, then there would be no need for reduction—the simulation could be performed once and the results stored for future use. However, simulations of dynamic memory management depend not only on the management of memory for a single process, but also the interactions between the many processes and the scheduler.[6] For these kinds of simulations, previously computed total fault counts for processes managed in isolation are insufficient to gather the desired results about the multiprogrammed system. With SAD-WS, we can codify our assumptions about the simulation environment (namely, that τ has at least a certain value) and obtain significantly shorter traces, valid for any subsequent simulations for which the assumption holds.

[6]Consider, for example, that WS requires that some process be deactivated if the working sets of all active processes do not fit in main memory. Whether or not the sum of the working set sizes exceeds the main memory capacity at any given moment depends on the scheduled interleaving of process execution.

Finding references to drop: In order to find the references that can safely be eliminated, we need only simulate a WS memory where the user chooses a τ reference reduction window. Thus, by scanning the original trace and keeping a history of the last τ references, we can again detect triplets of references to the same page, among the last τ references.

If a triplet is found, then it must be the case that not enough reference time would pass between the first and third references for the page to be evicted, *even if* the second reference were dropped. Just as with standard SAD, we can identify these middle references as ones that do not affect the sequence of fetches and evictions of WS with a τ element reference window.

Nevertheless, we cannot simply drop these references in order to form a reduced trace. Because the WS policy relies on the passage of reference time, and because reference time is normally inferred from the number of references present in the trace (as opposed to there being explicit timestamps on each reference), the dropping of a reference will incorrectly change the amount of reference time that passes. As a result, evictions of other pages will occur at the wrong time.

Modifying the trace format: Given this characteristic of WS, it seems that the dropping of any reference would alter the results. This limitation is applicable to any trace reduction method, including blocking: Any reduction method that eliminates references will yield incorrect results for simulations of WS if the reduced trace does not somehow account for the eliminated references. Therefore, any reduced trace that can be used with WS must contain annotations that provide reference timing information. While it is undesirable that we must change any simulator to understand this annotated trace format, the change is minimal, and would be required for any trace reduction method.

We can add a single integer annotation, which we will call the *time_passed* field, to each record of a reference trace. This field can be interpreted as the *amount of reference time that passed between the previous record and the current record.* An original reference trace can easily be pre-processed to conform to this format: For each record in the original trace, add the *time_passed* field with a value of 0, as no extra reference time passes between any two records of an original, unreduced trace.

Given this new annotation, SAD-WS can drop references without eliminating the needed reference time information. Specifically, for any reference that can be dropped, increment the *time_passed* field of the *next* record. Although a record has been eliminated, its contribution to the passage of reference time has been preserved.

Any WS simulator that can process a trace with this annotation needs simply to advance the reference time clock according to the *time_passed* field of each record before processing the reference indicated in that record. Provided that the τ chosen for simulation is no smaller than the τ chosen for reduction, all fetches and evictions will occur in the correct order, and at the correct reference time.

The modifications required to transform SAD into SAD-WS are straightforward. For instance, changing our implementation to track reference distance instead of LRU distance required the modification of only a few lines of code.

Applicability to VMIN: The VMIN policy is an offline policy that can also be easily described: *After a reference to a page, that page is kept resident if it will be referenced again within the next τ references, and evicted otherwise.* This policy is the forward-looking analog to WS; if the forward reference distance between two references to a page is greater than τ, then the page is evicted.

Our argument that traces reduced by SAD-WS are applicable to VMIN simulation is analogous to the argument that traces reduced by SAD are applicable to OPT, as described in Section 3.4. SAD-WS drops references only if the reference distance between two other references to the same page are close enough to ensure the page's residency. Whether the reference distance is examined in the backwards (WS) or forwards (VMIN) direction, the calculation of distance will be identical. Therefore, SAD-WS reduced traces will yield fully correct results for VMIN simulations provided that the τ chosen for simulation is at least as large as the reduction τ.

7 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` [LCB+98]. 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`,[7] 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 [Smi77], a well known reduction method. We chose to simulate CLOCK and SEGQ because they are the two replacement policies most used in real

[7] These applications, original traces, and trace gathering tool are available at [Kap].

systems. As approximations of LRU, they are similar to many replacement policies that discard information about references to the most recently used pages.

7.1 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 composes at most (and usually exactly) five bytes. Thus, there is a direct correspondence between the number of bytes and the number of records in a trace.

The plots in Figure 2 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 Table 1, 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). These sizes were chosen to show that significant reduction can be achieved even with a reduction memory that is likely much smaller than the smallest desired simulation memory.

It is worth noting that our reduced traces can be further compressed by applying lossless trace reduction techniques

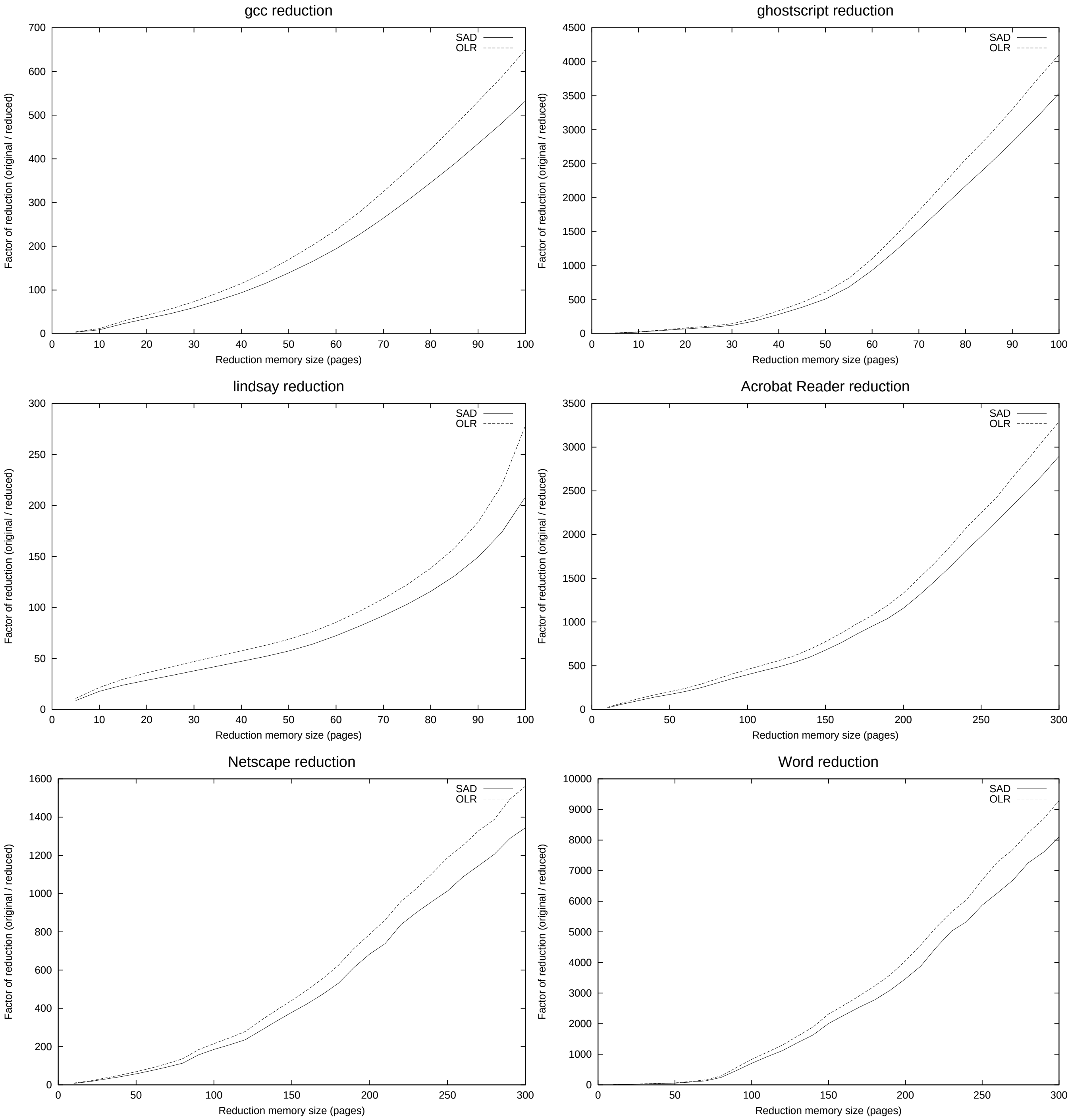

Figure 2: SAD and OLR reduction factors as a function of reduction memory size for six of the fifteen traces. The reduction factors for the traces not shown grow similarly with the reduction memory size. Note that the range for each y-axis is different.

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

Table 1: Even for small reduction memories, significant reduction factors can be achieved.

(for instance, [JH94, Sam89]). 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. Table 2 shows the compression factors achieved by the `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 of all reduction memory sizes. As reduction memories become larger (and reduced traces become dramatically smaller), compression factors shrink. Eventually, compression ratios approach a ratio of 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.

7.2 CLOCK and SEGQ Simulations

We simulated both the CLOCK and SEGQ replacement policies using traces reduced by SAD, OLR, and Smith's SD method. 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. Pure LRU is not used in real systems because of the overhead required to track the order of use of every page. However, recency tends to be an excellent predictor of future reference patterns, and so approximations of LRU are desirable. CLOCK and SEGQ avoid the overhead of pure LRU by discarding information about references to the most recently referenced pages, yet they successfully approximate the order of last reference among the less recently used pages in memory. Processors that provide *hardware reference*

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

Table 2: Reduced traces are often highly compressible with standard text-compression utilities.

bits—a per-page bit that indicates whether or a given page has been referenced—typically use the CLOCK algorithm, while those without reference bits rely on algorithms like SEGQ.

Our CLOCK simulator simulates a single-hand, two-reference-bit implementation. 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 to be arranged circularly (as though on the face of a clock), 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 both clear, 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 *two level replacement*, *hybrid FIFO-LRU* [BF83], or *segmented FIFO* [TL81]) 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.

While SAD and OLR cannot be used to perform exact simulations of CLOCK and SEGQ, we show that little error is introduced into the simulation if the ratio of simulation memory size to reduction memory size is sufficiently large.

7.2.1 Results and Comparison to Stack Deletion

An often referenced form of trace reduction is Smith's *stack deletion* (SD) [Smi77]. It is interesting to compare SD to our reduction techniques because its value has been demonstrated only empirically: 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. (The error is defined as the absolute value of the difference in the number of page faults incurred using the unreduced and reduced traces.)

Error introduced for fixed-size traces. We performed CLOCK and SEGQ simulations using each reduced trace. 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). We could have used the same reduction memory size for each method, but that approach would have been unfair for SD since it keeps less information for a given reduction memory size than either OLR or SAD.

Subsets of the results are shown in Figure 3 (for the CLOCK replacement algorithm) and Figure 4 (for SEGQ). These plots show the percent error (i.e., absolute difference in number of page faults) introduced by SAD and SD. Note that the plots do not show the results for OLR because its inclusion makes these noisy plots difficult to read. OLR performed similarly to SAD for SEGQ simulations and between SAD and SD for CLOCK simulations. For each replacement algorithm we selectively show results for six of the fifteen traces we studied. Nevertheless, we selected these traces to be representative of the results on all traces obtained for the algorithm in question. We statistically analyze the error for all fifteen traces later in this section.

For the CLOCK plots of Figure 3, we see that SAD exhibits smaller error than SD at almost every memory size. Although not shown, on average OLR introduces more error than SAD but less than SD. 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. However, there are specific cases in which each of the methods yields unacceptably large error values given small simulated-to-reduction memory size ratios. 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.

For the SEGQ simulations of Figure 4, the size of the FIFO segment is fixed (at approximately twice the reduction size for SAD), and the percent error is shown for increasing LRU segment sizes. The results were similar to those for CLOCK, although less error was introduced on average for all reduction methods. Note that the error introduced is

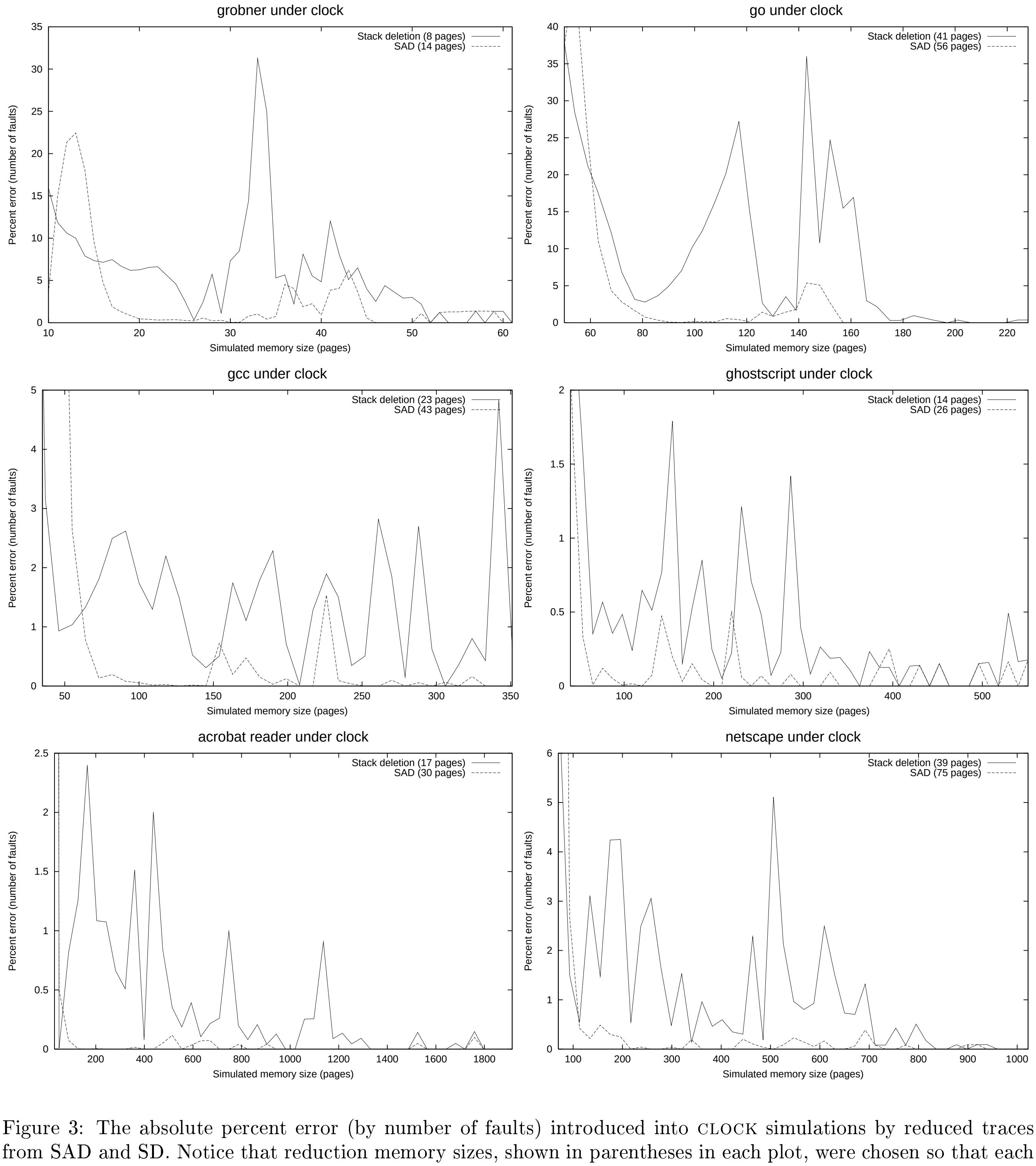

Figure 3: The absolute percent error (by number of faults) introduced into CLOCK simulations by reduced traces from SAD and SD. Notice that reduction memory sizes, shown in parentheses in each plot, were chosen so that each reduced trace was approximately 100 times smaller than its original.

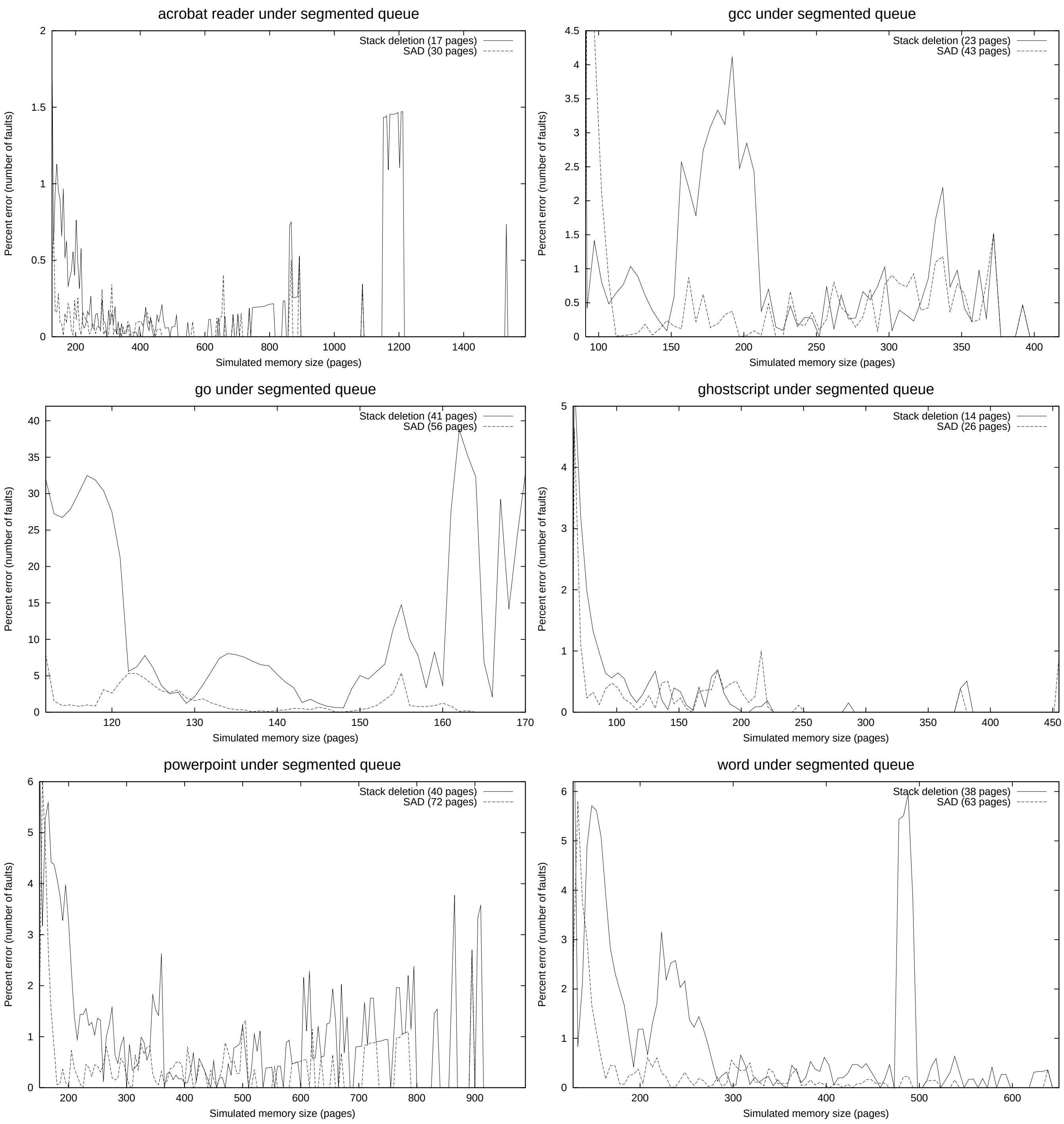

Figure 4: The absolute percent error (by number of faults) introduced into SEGQ simulations by reduced traces from SAD and SD. Again, the reduction memory sizes (given in parentheses as part of each legend) were chosen so that each trace was reduced by a factor of 100. For each plot, the FIFO segment size is fixed, and the total memory size reflects the combination of the FIFO and LRU segments.

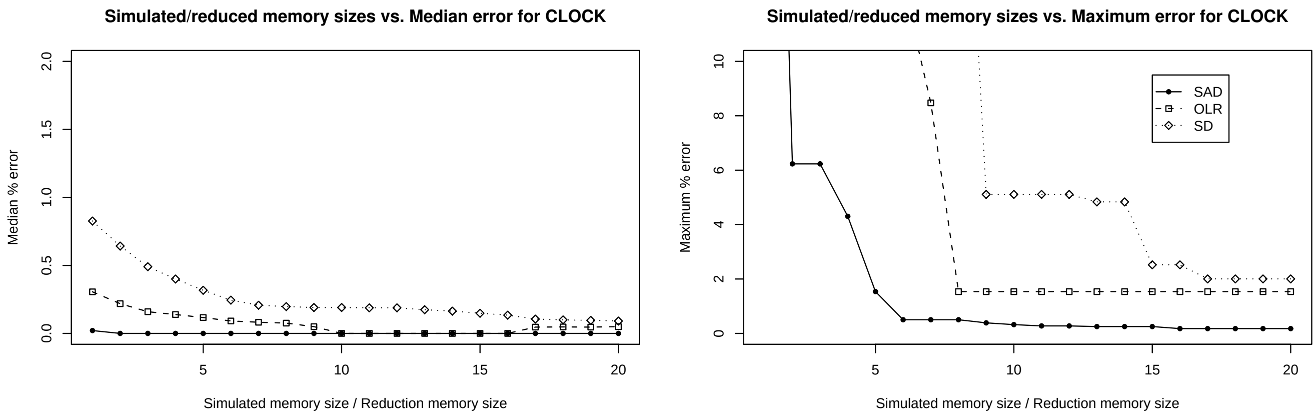

Figure 5: Under CLOCK, the median and maximum error percentages as a function of simulated-to-reduction memory size ratio. For all three methods, there is less than 1% median error, even for a simulated memory size that equals the reduction memory size. However, the maximum error values reveal that larger simulated-to-reduction memory size ratios are required to ensure small error.

irregular; the behavior of SEGQ is dominated by FIFO, which is not a stack algorithm and can produce unpredictably different results with slightly different memory sizes. SAD introduces less error than SD for most (but not all) memory sizes. Although 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.

Finally, we should point out that the reduced traces sometimes caused too many misses, and sometimes too few. Nevertheless, we observed no pattern or bias for reduction or increase in miss numbers.

Statistical analysis and reduction size guidelines. In Figures 5 and 6 we show the median and maximum percent error values as a function of the simulated-to-reduction memory size ratio for the CLOCK and SEGQ replacement algorithms, respectively. These results were computed for all fifteen of the traces and all of the simulated memory sizes.[8] These statistics are important not so much for direct comparison between SAD, OLR, and SD (this was covered earlier) but for providing guidelines on choosing a reduction size for a given error tolerance and desired simulation size.

For CLOCK (Figure 5), the median error is less than 1% for all three reduction methods, even for a simulated memory size equal to the reduction memory size. Nevertheless, the maximum error values reveal that larger simulated-to-reduction memory size ratios are needed to ensure small error. SAD requires a ratio of 5:1 to achieve 2% error in the worst instance, while OLR requires an 8:1 ratio. Smith claimed that a ratio of 2:1 would be sufficient for

[8] We did exclude memory sizes for which the total number of faults was less than 100. So few faults are insignificant, but the raw differences between such small numbers would yield large percentage differences, thus skewing the results.

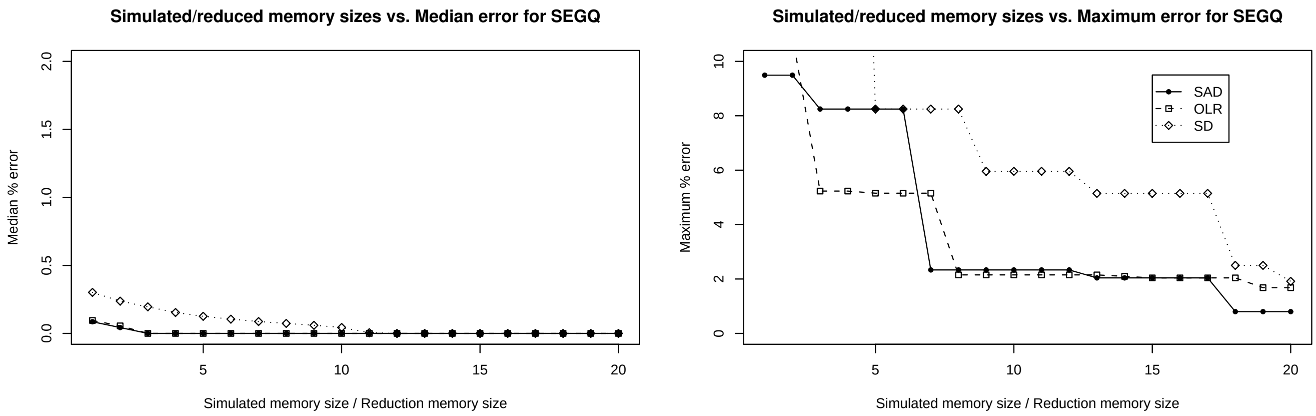

Figure 6: Under SEGQ, the median and maximum error percentages as a function of simulated-to-reduction memory size ratio. As with CLOCK, the median error is less than 1% at all ratios, but larger ratios are required to ensure acceptably small maximum error values.

experimentation with SD, but we see here that a ratio of 17:1 is needed to ensure smaller error.

For SEGQ (Figure 6), the median error is particularly low at all ratios at less than 0.5%. As with CLOCK, however, the maximum error percentages dictate the choice of memory size ratio. SAD nears 2% error at 7:1, and OLR at 8:1, with the maximum error decreasing gradually at larger ratios. SD requires an 18:1 ratio before approaching 2% error; at any smaller ratio, SD yields more than 5% error.

We should note that, although not directly shown, these results favor OLR and SAD over SD. For instance, if we want to limit the maximum CLOCK error to 2%, we need to choose a simulated-to-reduction memory size ratio of 17:1 for SD, 5:1 for SAD and 8:1 for OLR. Even though SD keeps less information than SAD or OLR for the same reduction size, the difference in ratios is so big as to favor SAD and OLR. For all reasonable simulation memory size ranges and all programs examined, the SD trace for a 17:1 ratio will be larger than the SAD trace for the 5:1 ratio and than the OLR trace for the 8:1 ratio.

7.2.2 Conclusions on CLOCK and SEGQ Error

The two main conclusions from our CLOCK and SEGQ experiments are:

- SAD and OLR introduce little error for CLOCK and SEGQ simulations. For the vast majority of memory sizes, less than 2% error was observed for both reduction methods. More importantly, we can limit the maximum error by choosing an appropriate reduction memory size for the desired simulation size.
- 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. In addition, SAD and OLR

Program	Min. simulatable mem. size (KB)		Trace size (events)		Ratio (GlCa:OLR)
	LRU	OPT	GlCa	OLR	
applu	2432	972	267863	149803	1.79
blizzard	5332	4772	989866	79775	12.41
coral	7084	6780	2126367	575291	3.7
gcc	1900	1052	410761	37844	10.85
gnuplot	1552	476	111488	52860	2.11
ijpeg	1112	748	947219	409392	2.31
m88ksim	1964	328	246135	124568	1.98
murphi	2132	1472	231993	156036	1.49
perl	9636	8428	3239741	705598	4.59
swim	6932	6216	180782	101881	1.77
trygtsl	2444	1400	118262	69384	1.7
turb3d	7720	6360	1964495	417783	4.7
vortex	3024	2028	374378	59473	6.29
wave5	3652	1708	159454	59830	2.67

Table 3: Reduction of the traces from [GC97]. Ratios of the **number of events** in traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger.

both allow for the exact simulation of LRU (and LRU variants like GLRU [FLW78], SEQ [GC97], FBR [RD90], EELRU [SKW99]), and SAD allows for the exact simulation of OPT.

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.

7.3 Comparison to the Glass and Cao Technique

A limited comparison of our reduction technique to the method of Glass and Cao is presented here. As we had no access to the unreduced form of the traces used in [GC97] it was not possible to reduce them using SAD, which requires an original reference sequence in order to reduce it. We were able, however, to obtain a behavior sequence from the reduced Glass and Cao traces, and from those behavior sequences we produced OLR-reduced traces. Both the Glass and Cao- and OLR-reduced traces are valid only for the simulation of sufficiently large memory sizes, where the smallest such size was determined by the reduction of the Glass and Cao technique.

As mentioned in Section 2.2, the reduction technique of Glass and Cao does not allow the user to select the memory range for which simulations should be exact. Instead, the CPU-time granularity of sampling for reduction is chosen by the user, and this value determines a minimum memory size for which LRU and OPT simulations are

Program	Min. simulatable mem. size (KB)		Trace size (KB)		Ratio (GlCa:OLR)
	LRU	OPT	GlCa	OLR	
applu	2432	972	4443	1074	4.14
blizzard	5332	4772	16417	543	30.21
coral	7084	6780	35268	3996	8.83
gcc	1900	1052	6766	251	26.96
gnuplot	1552	476	1832	398	4.6
ijpeg	1112	748	15558	3148	4.94
m88ksim	1964	328	4048	843	4.8
murphi	2132	1472	3847	1053	3.65
perl	9636	8428	53711	5270	10.19
swim	6932	6216	3000	691	4.34
trygtsl	2444	1400	1962	526	3.73
turb3d	7720	6360	32543	3105	10.48
vortex	3024	2028	6205	400	15.53
wave5	3652	1708	2632	442	5.95

Table 4: Reduction of the traces from [GC97]. Ratios of the **trace file size** from traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger.

Program	Min. simulatable mem. size (KB)		Trace size gzipped (KB)		Ratio (GlCa:OLR)
	LRU	OPT	GlCa	OLR	
applu	2432	972	1964	259	7.59
blizzard	5332	4772	7107	88	80.84
coral	7084	6780	15455	999	15.47
gcc	1900	1052	2963	38	77.92
gnuplot	1552	476	732	108	6.78
ijpeg	1112	748	6838	494	13.83
m88ksim	1964	328	1904	249	7.66
murphi	2132	1472	1673	234	7.15
perl	9636	8428	23066	1396	16.53
swim	6932	6216	1176	181	6.49
trygtsl	2444	1400	642	142	4.51
turb3d	7720	6360	11527	837	13.77
vortex	3024	2028	2733	78	35.05
wave5	3652	1708	1154	127	9.06

Table 5: Reduction of the traces from [GC97]. Ratios of the **compressed trace file size** from traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger.

possible (see any of Tables 3 to 5). Critically, the resulting minimum memory size cannot be determined until after reduction has occurred, and controlling that minimum memory size by changing the sampling granularity may be difficult.

We obtained OLR-reduced traces for the minimum LRU simulatable size (or any size greater than that) by performing a simple (2 source-line) change to the LRU simulator of Glass and Cao. This modification caused the behavior sequence of the LRU memory to be output. OLR was then used to construct the smallest trace exhibiting this behavior for the given memory size.

Tables 3, 4, and 5 compare the size of the OLR-reduced trace (for a reduction memory size equal to the minimum simulatable LRU size) with the reduced trace of Glass and Cao. Since the traces are in a different format (our implementation of OLR uses a simple text format, while the traces of Glass and Cao are memory images) there are many possible ways to compare them.

The first metric, shown in Table 3, presented is events-per-trace. This is as close as possible to an "apples-to-apples" comparison. Any possible error would be in favor of the Glass and Cao technique, as OLR events are strictly smaller than events in the Glass and Cao representation, which contains at least the reference information of OLR as well as extra ordering information.

The second metric, used for Table 4, shows the size of each reduced trace file in KBytes. While it is useful to see that traces reduced by OLR are smaller in a raw, uncompressed representation, this metric is probably the least informative. The choice of representation can have a significant effect on file size, and these two reduced trace formats differ significantly in that respect.

In order to reduce the effect of differing representations, we compressed each of the reduced trace files with the `gzip` utility. The results of these compressions are shown in Table 5. Given a compressed encoding for which redundant information has been largely removed, OLR has removed more information from the trace while still providing for the same exacting simulations.

8 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.

Furthermore, there is no single trace format appropriate for all kinds of simulations. SAD and OLR can both be modified with minimal effort to handle different kinds of annotations, maintain timing information, and even reduce traces on a fundamentally different scale than LRU distance. This flexibility simplifies the problem of storing and processing traces for specialized simulations.

We have implemented SAD and OLR, and have made them freely available on our web site at [Kap]. 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 offline with existing traces, or online as traces are gathered.

References

[AH90] A. Agarwal and M. Huffman. Blocking: Exploiting spatial locality for trace compaction. In *Proceedings, ACM SIGMETRICS*, pages 48–57, 1990.

[Bab81] Ozalp Babaoglu. Efficient generation of memory reference strings based on the LRU stack model of program behaviour. In *Proceedings, PERFORMANCE '81*, pages 373–383, 1981.

[Bel66] L. A. Belady. A study of replacement algorithms for virtual storage. *IBM Systems Journal*, pages 5:78–101, 1966.

[BF83] Ozalp Babaoglu and Domenico Ferrari. Two-level replacement decisions in paging stores. *IEEE Transactions on Computers*, C-32(12):1151–1159, December 1983.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw Hill and The MIT Press, 1989.

[CR70] E. G. Coffman and B. Randell. Performance predictions for extended paged memories. *Acta Informatica*, 1:1–13, 1970.

[Den76] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 19(5):285–294, 1976.

[Dou93] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, January 1993.

[FLW78] E. B. Fernandez, T. Lang, and C. Wood. Effect of replacement algorithms on a paged buffer database system. *IBM Systems Journal*, 22(2):185–196, 1978.

[GC97] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *SIGMETRICS The 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 25, pages 115–126. ACM Press, June 1997.

[JH94] E. E. Johnson and J. Ha. Pdats: Lossless address trace compression for reducing file size and access time. In *Proceedings, IEEE International Conference on Computers and Communications*, pages 213–219, 1994.

[Kap] Scott F. Kaplan. WWW trace reduction web-page. `<http://www.cs.amherst.edu/~sfkaplan/research/trace-reduction>`.

[Kap99] Scott F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, August 1999.

[KSW99] Scott F. Kaplan, Yannis Smaragdakis, and Paul R. Wilson. Trace reduction for virtual memory simulations. In *1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* [SIG99], pages 47–58.

[LCB+98] 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.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.

[PF76] B. G. Prieve and R. S. Fabry. VMIN – an optimal variable space page-replacement algorithm. *Communications of the ACM*, 19(5):295–297, May 1976.

[Pha95] Vidyadhar Phalke. *Modeling and Managing Program References in a Memory Hierarchy*. PhD thesis, Rutgers University, 1995.

[Puz85] Thomas R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachussetts, Dept. of Electrical and Computer Engineering, February 1985.

[RD90] J. Robertson and M. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS*, 1990.

[Sam89] A. Dain Samples. Mache: No-loss trace compaction. In *ACM SIGMETRICS*, pages 89–97, May 1989.

[SIG99] *The 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM Press, June 1999.

[SKW99] Yannis Smaragdakis, Scott F. Kaplan, and Paul R. Wilson. EELRU: Simple and efficient adaptive page replacement. In *1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* [SIG99], pages 122–133.

[Sma98] Yannis Smaragdakis. Optimal trace reduction for LRU-based simulations. Technical Report 98-25, The University of Texas at Austin, 1998.

[Smi77] Alan J. Smith. Two methods for the efficient analysis of address trace data. *IEEE Transactions on Software Engineering*, SE-3(1), January 1977.

[TL81] R. Turner and H. Levy. Segmented fifo page replacement. In *SIGMETRICS The 1981 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM Press, 1981.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *Computing Surveys*, 29(2):128–170, 1997.

[Wil91] Paul R. Wilson. Operating system support for small objects. In *International Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, California, October 1991. IEEE Press.

[WKS99] Paul R. Wilson, Scott F. Kaplan, and Yannis 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.