Trace Reduction for Virtual Memory Simulations

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

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

Abstract

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

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

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

1 Introduction

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

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

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

the length of a reduced trace. Both algorithms are efficient in practice, and significantly reduce storage *and* processing costs.

Guaranteeing accurate simulation for LRU and OPT may not seem exciting at first. If the trace were used only with these policies, the simulation could be run only once and the results stored and re-used. Our approach is effective, however, for simulations of *many* virtual memory replacement policies. Nearly all replacement policies used or studied with real workloads are either variants or approximations of LRU. This similarity of common page replacement policies is hardly surprising—good replacement algorithms should not evict pages that are in current use.

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

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

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

Additionally, the ability of the SAD algorithm to produce reduced traces valid for exact OPT simulations is a pleasant side-effect: it means that a single trace can be used for all experiments in a virtual memory study. Such studies often compare a new algorithm to LRU and OPT. All these experiments can be conducted with perfect accuracy (in the case of LRU, OPT, and LRU variants) or small error (in the case of LRU approximations).

2 Background and Motivation

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

2.1 Overview of Related Work

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

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

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

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

The simplest lossy reduction technique is *blocking*. Blocking replaces references to individual addresses with references to memory pages. Subsequent references to addresses within the same page can then be

reduced to a single reference. This reduction does not affect the simulation of *time-independent paging algorithms*—algorithms that do not consider the exact time of each reference in making replacement decisions. Such algorithms are LRU, OPT, etc., but not, for instance, Working Set [Denn68]. Blocking is so widely applicable that it is practically assumed in most simulation work. For the remainder of this paper, when we refer to an *original* trace, we are referring to a blocked trace.

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

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

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

2.2 The Value of Our Techniques

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

- Smith's stack deletion [Smit77] consists of only keeping references that cause pages to be fetched to an LRU memory of size k. Stack deletion is directly comparable to the SAD algorithm. Both techniques are very simple and have similar preconditions: both require that the reduced trace be used with memories no smaller than the memory used for reduction. Nevertheless, SAD guarantees that no error is introduced for LRU and OPT simulations, contrary to stack deletion. Smith argued experimentally that the error of stack deletion is small. Still, it is a significant drawback for virtual memory experiments. For one thing, it is not clear how large the error will be for experimental (i.e., not strictly LRU) replacement algorithms. For another, the error of stack deletion is indeed small but only if the depth of the stack (i.e., the size of the memory used for reduction) is much smaller than the simulated memory (typically 20% to 50% of its size). Hence, SAD can use a much larger reduction memory and achieve exact results. As we show in our experiments, the larger reduction memory size translates into significantly more reduction. Additionally, we show that stack deletion introduces larger error than both SAD and OLR for CLOCK and SEGQ simulations (for reduced traces of the same size). In conclusion, SAD and OLR are both safer (i.e., introduce less error) and more effective (i.e., yield smaller traces useful for comparable purposes) than stack deletion.

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

- Just like our techniques, the reduction method used by Glass and Cao [GlCa97] is applicable to exact virtual memory simulations. Like Coffman and Randell's method, the Glass and Cao technique suffers from needing to modify the simulator to accept the reduced trace format. The modifications are far from trivial and it can be hard to use the reduced trace information for simulations of policies other than those studied in [GlCa97] (LRU, OPT, and SEQ—an experimental replacement algorithm). Another drawback of this technique is its lack of control over the interesting memory ranges. It is not possible to specify directly the memory sizes for which the simulation should be exact. Instead, the trace filter intro-

duces uncontrollable factors which determine the memory sizes for which the simulation is valid. Also, the method seems to be less efficient than our approach, at least for LRU simulations. We did not have access to the traces used by Glass and Cao in unreduced form, but were able to derive the OLR-reduced form of these traces (directly from the Glass and Cao reduced traces). This was several times shorter than the reduced form used by Glass and Cao, both in terms of absolute size and in terms of significant events. We present these results in Appendix A.

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

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

3 The Algorithms

3.1 Safely Allowed Drop (SAD)

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

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

3.1.1 Finding References to Drop

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

SAD is an application of this observation. The user specifies a *reduction memory size*, k. Then SAD searches the trace from left to right, to find triples 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 triples are eliminated.

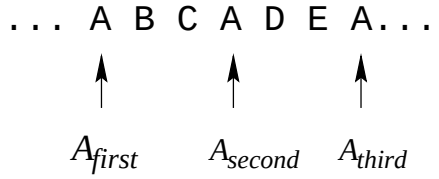

Figure 1: A_{second} can be eliminated because the LRU distance between A_{first} and A_{third} is less than the reduction memory size of 5 pages.

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

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

3.1.2 SAD Algorithm Implementation

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

As the trace is processed, the algorithm maintains an LRU queue of the requested size. It also stores

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

Although it is necessary to store recent references to find these triplets, the number of references can be bounded. It is only necessary to store at most $2k+1$ of the most recent references in order to find the LRU distance between first and third most recent references to a page. Because of space limitations, we will not describe how to derive this bound. With something like a hash table to help find recent references to pages, performing this reduction is little more than an augmented LRU queue simulation; it can be executed efficiently. For more details, we refer you to our implementation of SAD at `<http://www.cs.utexas.edu/users/oops/>`.

3.1.3 Exact Simulation of LRU and OPT

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

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

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

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

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

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

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

3.2 Optimal LRU Reduction (OLR)

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

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

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

`A B C A D B`

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

$\langle \mathtt{A}, NF \rangle, \langle \mathtt{B}, NF \rangle, \langle \mathtt{C}, NF \rangle, \langle \mathtt{D}, \mathtt{B} \rangle, \langle \mathtt{B}, \mathtt{C} \rangle$

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

The importance of LRU for actual virtual memory systems motivated the design of the Optimal LRU Reduction algorithm (OLR). OLR takes a reference trace as input and outputs the smallest trace that has the same LRU behavior as the input for a memory of size k or larger. The output of OLR is a function of the behavior of the input trace (and not directly of the trace itself). Hence, the first step of OLR is to simulate the input trace on an LRU memory of size k and derive its behavior sequence. Then, the OLR_CORE algorithm shown in Figure 2 is applied to produce the shortest trace having the given behavior. Note that we use the term *block* as a synonym for *page*.

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

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

Due to space limitations we cannot present an extensive treatment and proof of correctness for the OLR_CORE algorithm. Such analysis can be found in [Smar98]. Here we will only note that OLR_CORE is very efficient, so that the main component of the running time of OLR is the LRU simulation performed on the input trace to derive its behavior sequence. That is, OLR execution is about as fast as a simple LRU simulation on the input trace for a memory of size k. The algorithm performs just a single forward pass with bounded look-ahead (at most k elements) and, thus, is ideal for online applications. Our free implementation of OLR can be found at `<http://www.cs.utexas.edu/users/oops/>`.

3.3 Trace Manipulation Issues

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

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

  Note that both SAD and OLR guarantee that the sequence of fetches into and evictions from a k page LRU memory are the same as specified by the original trace. 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 as specified by the original trace. If a page is dirtied while in the reduction memory, then the last reference to that page before it is evicted is marked as a *write* operation. A simulation based on the reduced trace will mark the page as dirty before it is evicted from a k page or larger memory.
- **Maintaining Timing Information**: timing information is trivial to maintain for SAD, since the algorithm only removes references from the original trace. For OLR, where reference reordering may occur, it makes sense to keep time information for references causing a page to be fetched into memory. These are guaranteed to be exactly the same (and, hence, in the same order) as in the original trace.

4 Experimental Results

We applied our trace reduction methods to traces collected both on Windows NT and UNIX platforms. The nine Windows NT traces include the full set of

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

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

Figure 2: OLR_CORE is the core of the OLR algorithm

the commercially distributed traces gathered using the utility `Etch` [LCBAB98]. These include well-known Windows NT applications (Acrobat Reader, Netscape, Photoshop, Powerpoint, Word) as well as various other programs (CC, Compress, Go, Vortex). The six UNIX traces (Espresso, GCC, Grobner, Ghostscript, Lindsay, P2C) were gathered using `VMTrace` [Kapl98], a portable tracing tool based on user level page protection; these traces are freely available on our web site. The Windows NT traces were blocked for 4 Kbyte pages so that they would be appropriate for virtual memory simulations. The UNIX traces were generated as references to 4 Kbyte pages.

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

4.1 Reduction Results

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

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