FASTSLIM: Prefetch-Safe Trace Reduction for I/O Cache Simulation

Wei Jin, Xiaobai Sun, and Jeffrey Chase
Department of Computer Science
Duke University
{jin, xiaobai, chase}@cs.duke.edu

Simulation is an indispensable tool for evaluating I/O systems, complementing benchmarking and analytical modeling. This paper presents a new algorithm, called FASTSLIM, to reduce the size of I/O traces. FASTSLIM improves performance of trace-driven simulation of I/O caching systems without compromising simulation accuracy.

FASTSLIM is more general than existing trace reduction schemes in two ways. First, it is ***prefetch-safe***, i.e., traces reduced by FASTSLIM yield provably exact simulations of I/O systems with prefetching, a key technique for improving I/O performance. Second, it is compatible with a wide range of cache replacement policies, including widely used practical approximations to LRU. FASTSLIM-reduced traces are safe for simulations of storage hierarchies and systems with parallel disks.

This paper gives a formal treatment of prefetching and replacement issues for trace reduction, introduces the FASTSLIM algorithm, proves that FASTSLIM and variants are safe for a broad range of I/O caching and prefetching systems, and presents experimental results from applying FASTSLIM to a representative set of virtual out-of-core (VOOC) applications. The results show that FASTSLIM reduces trace volume by factors of 10^2 to 10^3 for the applications studied.

General Terms: Trace Reduction, Trace-Driven Simulation, Prefetching, File Caching, Virtual Memory, I/O Architectures, Operating Systems, Performance Evaluation

---

Support was provided in part by Advanced Research Projects Agency under contract P-95006, by NSF/CISE under grant CDA-9726370, by DARPA/DSO under grant DABT63–98–1 –0001, by the National Science Foundation (CCR-96-24857, CDA-95-12356, EIA-99-72879 and EIA-9870724) and an equipment grant from Intel Corporation.

1. INTRODUCTION

A key challenge for high-performance computer systems is to bridge the widening gap in bandwidths and access times between internal memory and external storage. One approach to attacking the I/O bottleneck is through software innovations to manage the memory/storage hierarchy more effectively. Research in this direction has given rise to two important trends. First, many systems extend the storage hierarchy to incorporate parallel disks [Patterson et al. 1988], tertiary storage, and multiple levels of DRAM-based I/O cache, e.g., network memory [Feeley et al. 1995]. Second, many systems employ new techniques to *proactively* manage movement and placement of data in the storage hierarchy. In particular, prefetching is now a part of every advanced I/O system, and recent research has yielded a family of integrated prefetching and caching schemes [Cao et al. 1995; Kimbrel et al. 1996; Patterson et al. 1995].

The most flexible and powerful tool for evaluating new I/O structures is simulation, which handles varying assumptions about the workload and the hardware. Unfortunately, high overhead compromises the usefulness of simulation. While *execution-driven* simulation eliminates the need to store and process large trace files, it leads to unacceptably high processing costs for evaluating I/O systems. On the other hand, trace files for *trace-driven* simulation may be very large, leading to excessive costs to store the trace and run the simulation. The problem of "trace bloat" is particularly acute for programs that use virtual memory paging to drive I/O. This approach, sometimes called *virtual out-of-core* (VOOC) computing [Mowry et al. 1996], is appealing to application programmers because it allows uniform access to very large data sets while isolating programs from the details of I/O. A complete trace for a VOOC application includes every memory reference, because any memory reference could potentially result in I/O. For example, the complete page-level access trace for an *eigen* benchmark used in Section 5 is on the order of a terabyte, even with a problem size of 30 megabytes.

Practical use of trace-driven simulation depends on development of *trace reduction* techniques to make the simulation feasible with limited disk space and in reasonable simulation time [Puzak 1985; Coffman and Randell 1971; Smith 1977; Glass and Cao 1997; Kaplan et al. 1999]. The purpose of trace reduction is to trim from a given reference trace as many references as possible, while retaining sufficient information to accurately simulate the range of systems to be studied. The primary contribution of this paper is to address the problem of accuracy-preserving trace reduction for I/O caching systems that proactively manage a storage hierarchy, particularly for systems that use prefetching. We are not aware of any previous trace reduction work that addresses the prefetching aspect, despite the volume of research in prefetching algorithms [Cao et al. 1995; Kimbrel et al. 1996; Patterson et al. 1995] and prefetching for virtual memory [Trivedi 1976; Mowry et al. 1996; Voelker et al. 1998].

This paper presents a *prefetch-safe* trace reduction algorithm, called FastSlim, that yields exact simulations for a large class of prefetching schemes with integrated caching and prefetching [Cao et al. 1995]. Section 2 defines this class formally; it includes the early prefetching algorithm DPMIN [Trivedi 1976] and more recent prefetching schemes such as AGGRESSIVE [Cao et al. 1995], FORESTALL [Kimbrel et al. 1996], JUST-IN-TIME [Patterson et al. 1995], and COMPILER-DIRECTED PREFETCHING [Mowry et al. 1996]. The FastSlim algorithm is also compatible with a broad range of block replacement policies including LRU, OPT (MIN), and their widely used practical approximations, and it is general enough to accommodate parallel disks [Kimbrel et al. 1996], hierarchical caches, and cooperative prefetching in clusters incor-

porating network memory [Voelker et al. 1998]. We also introduce a variant of FASTSLIM called FASTSLIM-DEMAND for the demand paging (no-prefetch) systems targeted by earlier trace reduction schemes; FASTSLIM-DEMAND is not prefetch-safe, but like FASTSLIM it is compatible with a broad range of replacement policies.

This paper is organized as follows: Section 2 presents the motivation for our work in more detail, surveys previous work in trace reduction and I/O prefetching, and outlines the challenges of prefetch-safe trace reduction. Section 3 presents the FASTSLIM algorithm for prefetch-safe trace reduction, and the variant FASTSLIM-DEMAND. Section 4 proves that FASTSLIM is safe for a broad and well-defined class of integrated I/O caching and prefetching systems, and that FASTSLIM-DEMAND is safe for demand-paged I/O systems with a full range of theoretical and practical variants of LRU and OPT replacement. Section 5 presents results from experiments with FASTSLIM on a representative set of VOOC applications, showing that FASTSLIM reduces trace volume substantially, typically by a factor of 10^2 to 10^3. Section 5 also shows that the reduction ratios achieved by FASTSLIM are within a factor of 10 of SAD [Kaplan et al. 1999], a recent near-optimal trace reduction algorithm that is not prefetch-safe, and that the reduction ratios achieved by FASTSLIM-DEMAND are competitive with SAD. Section 6 concludes.

2. TRACE REDUCTION AND PREFETCHING

This section presents background material on trace-driven simulation, trace reduction, and prefetch scheduling algorithms. It then defines the problems raised by the interaction of prefetching and trace reduction.

Simulation is an indispensable tool for studying the dynamic behavior of new I/O structures. It is complementary to analytical modeling, which is applicable to a limited range of workload assumptions and system environments, and benchmarking, which is useful only for complete implementations running on the target systems. While *execution-driven* or "on-the-fly" simulation is often used for memory system studies, it is less effective for I/O simulations. First, CPU behavior of a workload is largely independent of I/O, so it is not necessary to execute the workload to get a detailed and accurate I/O simulation. Second, it is inefficient to execute the workload multiple times because execution forces the I/O to actually occur, unless the workloads are constrained to fit in the memory of the simulation platform. The preferred alternative, *trace-driven* simulation, is to instrument each program and execute it once, recording data references in a trace file as an input to off-line simulations.

Formally, a *trace* is an ordered sequence of *trace items*. Each I/O trace item is a pair (p, t) representing a reference to a page or block p of an external dataset, occurring at time t. The t values may encode detailed timing information, such as the processor time of the reference relative to the start of the program, i.e., the time at which the reference would occur in a system with instantaneous I/O. In this case, the total execution time for the program on a simulated system can be determined by adding the I/O stall time reported by the simulation to the time t of the last reference in the trace.

This paper assumes only that the t values in the trace increase monotonically. The figures depicting traces in this paper represent the time t of each reference (p, t) by the position of the p value on a horizontal time axis.

One advantage of the trace-driven approach is that a saved trace can drive any number of simulations with varying hardware and software assumptions. Multiple simulations may

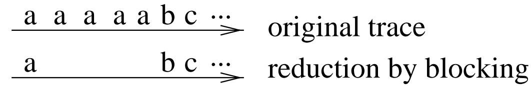

Fig. 1. A simple example of trace reduction by BLOCKING.

be combined in a single run. For example, Mattson's *stack processing* technique simulates a stack replacement algorithm on different memory sizes in a single pass [Mattson et al. 1970].

2.1 Trace Reduction

There are two basic approaches to reducing the size of trace files. First, trace files may be compressed using standard data compression techniques or a compression function that is specific to the trace format [Samples 1989]. While *trace compression* reduces storage costs, it does not reduce simulation time. In contrast, *trace reduction* seeks to reduce both storage and simulation time by reducing the number of items in the trace. The premise is that many references in a given trace hit in an I/O cache in the simulated system, thus only a subset of the references affect I/O demands and I/O decisions. The goal of a trace reduction algorithm is to identify and eliminate the unimportant references. Trace reduction can reduce trace sizes by factors of 10 to 10,000 or more, depending on the specific reduction algorithm and the locality characteristics of the trace (see Section 5). Most importantly in a time of declining storage costs, trace reduction improves simulation time proportionally, since simulation time is at best linear in the number of items in the trace.

To illustrate trace reduction, Figure 1 depicts a simple technique called BLOCKING. BLOCKING removes all consecutive references to a given block or page, preserving only the first reference of each run. Trace reduction by BLOCKING produces accurate simulation for any time-independent replacement policy for demand paging, such as LRU, CLOCK, MRU, and OPT, since no such scheme can evict a page between two consecutive references to that page. One drawback of BLOCKING is that it rarely reduces traces by more than 50%, as Section 5 shows. Also, BLOCKING is not prefetch-safe, as Section 2.3 shows.

Several trace reduction algorithms exist that are more effective than BLOCKING while preserving simulation accuracy for systems without prefetching. A common approach is to pass the trace through some sort of *filter* cache of size B blocks, retaining only the references that miss in the cache. The reduced trace can then be used to drive a simulation of any system with I/O cache sizes larger than B, for a restricted range of caching policies compatible with the policies used for the filter cache. Larger B values typically yield better reduction ratios at the expense of narrowing the range of target systems that can be accurately simulated using the reduced trace.

Computer architects commonly use trace reduction algorithms based on this principle to evaluate memory system designs. For example, Puzak's TRACE STRIPPING scheme [Puzak 1985] works for simulations of set-associative memory caches. It reduces the trace by recording only the references that miss in a direct-mapped cache. The reduced trace can then be used to obtain exact cache miss ratios for caches with higher degrees of set associativity.

In contrast to Puzak's scheme, several other trace reduction algorithms are applicable to demand paging and I/O caching systems, which are fully associative. Table 1 summarizes the key algorithms in the literature. Several of these reduction schemes manage the filter cache as a stack, and are applicable to stack replacement algorithms such as LRU. Coffman and Randell's

TRACE REDUCTION SCHEME	COMPATIBLE REPLACEMENT POLICY	SIMULATION MEMORY SIZE (K)
Stack Deletion (Smith [Smith 1977])	not accuracy-preserving	$K \geq B$
Coffman and Randell [Coffman and Randell 1971]	filter buffer policy	$K \geq B$
Glass and Cao [Glass and Cao 1997]	LRU or OPT	I/O sequence dependent
OLR: Kaplan et. al. [Kaplan et al. 1999]	LRU	$K \geq B$
SAD: Kaplan et. al. [Kaplan et al. 1999]	LRU or OPT	$K \geq B$

Table 1. Comparison of trace reduction schemes for demand paging systems.

EXTENSION MEMORY technique [Coffman and Randell 1971] records every *push* (a miss) and every *pull* (the replacement) for the reduced trace. Smith's STACK DELETION [Smith 1977] simplifies Coffman and Randell's algorithm, recording only the missing references under an LRU replacement policy. It does not, however, guarantee exact simulations for any replacement policy. Recently, Kaplan et. al. proposed two techniques, OLR and SAD [Kaplan et al. 1999], which rely on simulating an LRU stack. OLR translates the push/pull behavior sequence, obtained as in Coffman and Randell's technique, into the shortest reference trace with the same push/pull behavior. SAD uses the LRU stack simulation to identify and remove references that do not affect an LRU or OPT replacement policy.

Glass and Cao's technique [Glass and Cao 1997] is not based on stack simulation. Instead, it divides program instruction time into fixed-length intervals and generates a sequence of IN and OUT records for pages at the end of every interval. This algorithm generates IN records for pages referenced in the current interval but not the previous interval; it generates OUT records for pages referenced in the previous interval but not the current interval. The reduced trace yields accurate simulations for demand paging with LRU and/or OPT replacement. However, the ranges of compatible target memory sizes must be determined by inspecting the trace after the IN/OUT sequence is generated; it cannot be derived directly from the parameter B.

2.2 Integrated Caching and Prefetching

The trace reduction algorithms reviewed in Table 1 are intended for demand paging systems with no prefetching. Prefetching is an effective approach to reducing I/O stall times when sufficient bandwidth exists and the system is able to predict future references. Prediction is an active research topic [Vitter and Krishnan 1996; Chang and Gibson 1999; Mowry et al. 1996; Kroeger and Long 1996], and prefetching will become more valuable as prediction techniques continue to improve. Potential gains from prefetching have motivated theoretical and experimental research in *integrated prefetch scheduling* algorithms that balance prefetching and caching [Cao et al. 1995; Kimbrel et al. 1996; Patterson et al. 1995; Trivedi 1976; Voelker et al. 1998].

Integrated prefetch scheduling is based on the following simplified formal model, which introduces terminology and notation used in later sections. A *prefetch scheduler* $\mathcal{P}$ manages a physical memory of K pages using revealed knowledge of the access sequence (trace) σ. $\mathcal{P}$ controls both fetch and replacement policy; it decides which page to fetch next and when, and which page to evict to make room for each fetched page. As the program executes, $\mathcal{P}$ consumes the trace, looking ahead to anticipate references before they occur. The current execution point of the program is represented by a hypothetical *reference cursor* (RC) that

Algorithms	Timing of a Fetch
Aggressive (Cao et. al. [Cao et al. 1995], Kimbrel et. al. [Kimbrel et al. 1996])	Fetch whenever a disk is idle and the DO-NO-HARM set of a missing page on that disk is not empty.
Conservative (Cao et. al. [Cao et al. 1995])	Perform the same fetch/replacement sequence as demand paging using OPT, but fetch at first opportunity.
Just In Time (Patterson et. al. [Patterson et al. 1995])	Issue a fetch at most one I/O latency ahead of the reference.
Forestall (Kimbrel et. al. [Kimbrel et al. 1996])	Fetch whenever a future stall is detected.

Table 2. Comparison of practical prefetching algorithms.

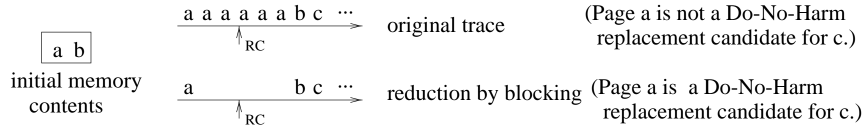

Fig. 2. Blocking is not prefetch-safe.

visits the trace items in sequence. The next reference to each missing page after the *RC* is called a *hole*. $\mathcal{P}$ may choose to fill any hole at any time by issuing a fetch for the missing page, at which point it must select a page to evict, which may create a new hole further ahead in the reference stream. Once $\mathcal{P}$ initiates the fetch, the fetched page arrives in memory after some delay determined by the I/O system, reducing or eliminating the I/O stall time incurred by the program when the *RC* advances to reference the page. The goal of $\mathcal{P}$ is to select a sequence of fetch and replacement actions that minimizes overall I/O stall time.

Table 2 summarizes some important prefetch scheduling algorithms that have been proposed and studied. The key principle underlying these integrated caching and prefetching schemes is DO-NO-HARM: never evict a page p to fetch q unless q will be referenced before p [Cao et al. 1995]. For given memory contents and *RC*, the DO-NO-HARM rule defines a *replacement candidates set* for any hole: to fetch the missing page, $\mathcal{P}$ must select a victim page from among the resident pages whose next reference after the *RC* occurs after that hole. A *prefetch opportunity* exists when the replacement candidates set for a hole is nonempty; DO-NO-HARM schedulers initiate prefetching only when an opportunity exists.

2.3 Prefetch-Safe Trace Reduction

The purpose of this paper is to present a trace reduction scheme — FastSlim — that is *prefetch-safe* in that it yields provably exact simulations for practical prefetch scheduling algorithms including those listed in Table 2. Previous work has defined and proved formal properties of these algorithms, and has evaluated them by benchmarking, and by simulation using small traces. A comprehensive simulation study of these prefetching algorithms is impractical without a prefetch-safe trace reduction scheme, particularly for virtual out-of-core (VOOC) applications in which the generated traces are very large.

The trace reduction algorithms surveyed in Section 2.1 are not compatible with prefetching: a trace reduced by one of these algorithms may induce a prefetch scheduler to take different fetch and replacement actions than it would have made on the original trace. To see why this is so, consider the simple BLOCKING reduction scheme introduced in Figure 1. Figure 2

presents a trivial example to show that trace reduction by BLOCKING removes trace items needed by a scheduler to conform to the DO-NO-HARM rule. Assume that the memory has two page frames capacity ($K = 2$) and initially holds pages a and b. The first missing page is c. By the DO-NO-HARM rule, $\mathcal{P}$ may not evict page a until after its fifth reference. In the trace reduced by BLOCKING, $\mathcal{P}$ may evict page a after its first reference, since subsequent references to a are omitted. The more sophisticated trace reduction schemes in Table 1 have a similar effect, since they were designed for simulating demand paging systems without prefetching.

The FASTSLIM algorithm presented in Section 3 is prefetch-safe for any prefetch scheduler that conforms to the DO-NO-HARM rule, uses LRU(n) or OPT(n) replacement (see Section 2.4), and makes deterministic fetch and replacement choices considering only σ, the RC, the memory contents, the I/O system status, and internal state built from information revealed in the trace. We refer to qualifying integrated prefetch scheduling algorithms as *Class-A* algorithms. Section 4 proves that FASTSLIM is safe for Class-A prefetch schedulers. All of the prefetch scheduling algorithms in Table 2 are Class-A algorithms.

As it turns out, FASTSLIM is safe even for prefetch schedulers that respect only a weaker form of the DO-NO-HARM rule: never evict a page p unless it is in the replacement candidates set for the first hole after the RC. The weak DO-NO-HARM rule allows the scheduler to evict a page p in order to issue an early fetch for a page q whose next reference is after the next reference to p, provided there is some other missing page whose next reference is before the next reference to p. The weak DO-NO-HARM rule allows use of FASTSLIM in conjunction with a wider class of prefetch scheduling algorithms. For example, the scheduler is permitted to make eviction choices in advance in order to maintain a reservoir of free page frames; the scheduler is legal "Class-A" if it evicts only pages whose next reference is after the first hole.

Another reason to relax the DO-NO-HARM rule is that Kimbrel and Karlin have shown that the optimal prefetching schedule for parallel disk systems may violate the strong DO-NO-HARM rule [Kimbrel and Karlin 1996]. This is because it may be useful to evict a page from a lightly loaded disk in order to issue an earlier fetch for a page from a heavily loaded disk. To be sure, an optimal scheduler for parallel disks may fail to conform even to the weak DO-NO-HARM rule. Even so, all practical prefetch scheduling algorithms adhere to some form of the DO-NO-HARM rule; the algorithms proposed to derive optimal or near-optimal schedules that may violate DO-NO-HARM [Cao et al. 1995; Albers et al. 1998; Kimbrel and Karlin 1996] are computationally expensive and therefore impractical. FASTSLIM is safe for known practical prefetching algorithms for any number of disks.

2.4 Replacement Policy

Replacement policy is a key aspect of a prefetch scheduler as defined in Section 2.2. In theory, a prefetch scheduler can use optimal (OPT or MIN) replacement, since prefetching assumes that the program's access sequence is revealed in advance. Optimal replacement requires that the *entire* access sequence be available in advance; this is impractical and unnecessary for effective prefetching. In practice, prefetching can be used in conjunction with any replacement policy that evicts only pages from the replacement candidates set as defined by the strong or weak DO-NO-HARM rule.

FASTSLIM is compatible with a wider class of replacement policies than the previous trace reduction algorithms summarized in Table 1. The proof in Section 4 considers schedulers that use deterministic LRU(n) or OPT(n) replacement. LRU(n) denotes a replacement policy that

replaces any of the n least recently referenced pages in the replacement candidates set, for some fixed value of n. Opt(n) denotes a replacement policy that replaces any of the n pages in the replacement candidates set whose next reference is furthest in the future, for some fixed value of n.

The lru(n) and opt(n) families of replacement policies encompass the important theoretical algorithms lru and opt (which are equivalent to lru(1) and opt(1) respectively) together with a continuum of practical approximations and extensions. For example:

—fifo-with-second-chance [Draves 1990] (FIFO-2C) is an lru($K-i$) replacement policy where K is the memory size and i is the size of the second-chance or *inactive* queue. The second-chance queue acts like an lru stack below a fifo queue of size $K-i$. Any access to a page in the second-chance stack brings that page up to the tail of the fifo queue and pushes the page on the head of fifo queue down to the top of the second-chance stack. FIFO-2C is an lru approximation, but any disturbance to the lru ordering among pages is bounded by the size of the fifo queue. That is, for any page p on the second-chance stack, among all pages above p in the stack there are at most $K-i-1$ pages older than p (the pages that were behind p when it was at the head of the fifo queue). It follows that the next replacement candidate — the page on the bottom of the stack — is one of the $K-i$ least recently used pages.

—Application-directed replacement [Glass and Cao 1997] or compiler-directed replacement [Mowry et al. 1996] are opt(n) policies, assuming that the n most attractive eviction candidates are always revealed to the scheduler.

—global lru, a policy for distributed network memory management used in the Global Memory System (GMS) [Feeley et al. 1995], can be modeled as an lru(n) policy. global lru makes eviction choices at global coordination points dividing execution into a sequence of intervals. The system identifies the n oldest pages to discard from the network memory at the start of each interval; it then probabilistically replaces those n victims in an arbitrary order during the interval. Thus global lru is an lru(n) policy for workloads that do not reference any page p in the same interval that evicts p.

Note that random replacement is an lru(K) or opt(K) policy, where K is the memory size. This means that for a given memory size, any time-independent replacement policy is an lru(n) or opt(n) policy for some value of n. This appears to show that FastSlim is compatible with any replacement policy (although that policy must be deterministic in order to prove equivalence). This is not as interesting as it might seem because any lru(n) or opt(n) policy is compatible only with a specific range of parameters to the FastSlim algorithm, determined by n. Like some of the other techniques surveyed, FastSlim processes the trace through a filter buffer whose size is given by a parameter (B). The value of B defines a balance of generality and effectiveness: larger filter buffers often absorb more references, but the resulting traces are compatible with a narrower range of systems. In the case of FastSlim, the reduced traces produce exact simulations for any Class-A prefetch scheduler and system in which $B \leq K-n+1$. This means that setting $n=K$ limits FastSlim to a filter buffer with a single entry. While the resulting reduced traces are safe even for random replacement, this yields reductions even worse than simple blocking. (Once the algorithm is understood, it can be seen that FastSlim with $B=1$ is a variant of blocking that retains the last reference in each run as well as the first reference.)

2.5 Summary

We summarize the key points of this section as follows. Trace-driven simulation is the most flexible tool for evaluating I/O caching and prefetching systems, but it relies on effective trace reduction to be practical. Virtual memory I/O (VOOC) in particular generates larger trace files, which must be reduced. Previous work in trace reduction has yielded trace reduction schemes that apply to demand paging systems with a limited range of replacement policies, and do not consider prefetching. We designed the FastSlim algorithm to reduce traces effectively while preserving simulation accuracy for "Class-A" *integrated prefetch schedulers*, defined by deterministic scheduling of fetches and evictions in conformance with the strong or weak do-no-harm rule. This class of prefetch scheduling algorithms has been the focus of theoretical and applied work in I/O prefetch scheduling. The FastSlim trace reduction algorithm will enable a comprehensive simulation study of these prefetching schemes, including VOOC workloads.

A second contribution of our work with FastSlim is that it shows how trace reduction can handle a wide range of replacement policies easily. This contribution extends to the demand paging systems targeted by the earlier trace reduction schemes, since demand paging is in effect the null do-no-harm prefetching algorithm. To support this point, the next section presents FastSlim along with a simple variant called Fastslim-Demand that is not prefetch-safe, but is competitive with known near-optimal trace reduction algorithms. Like FastSlim, Fastslim-Demand accommodates the full range of lru(n) and opt(n) replacement policies for demand paging systems.

3. FastSlim: A PREFETCH-SAFE TRACE REDUCTION ALGORITHM

This section presents the FastSlim trace reduction algorithm and its variant FastSlim-Demand. The input to the algorithm is a trace σ, an ordered sequence of trace items (p, t) representing a reference to a page p occurring at time t. The output is a reduced trace, a subsequence of the original trace.

FastSlim reduces the original trace by passing it through a *filter buffer* of size B. The filter buffer retains the most recent references for up to B of the most recently accessed pages; some repeat references to pages already present in the buffer are omitted from the reduced trace. We emphasize that the FastSlim filter is a buffer rather than a simulated cache. In particular, the buffer has no replacement policy: FastSlim flushes the buffer each time it fills up. Also, FastSlim may retain an item in the reduced trace even if that item hits in the buffer, if it is possible that a prefetch scheduler would need that item to make a replacement decision. These characteristics distinguish FastSlim from earlier trace reduction algorithms, and are the keys to its generality.

Figure 3 gives pseudocode for FastSlim and Fastslim-Demand. For brevity, Figure 3 ignores the mechanism for emitting items of the reduced trace in timestamp order. The algorithm examines each item (p_i, t_i) of the original trace in sequence, placing it in the filter buffer. If an earlier item referencing the page p_i is not already present in the buffer, then FastSlim emits (p_i, t_i) into the reduced trace output, possibly along with one or more of the items stored in the filter buffer. For each item (p_i, t_i), the algorithm considers three cases:

(1) If the filter buffer contains an item associated with page p_i, then replace that item with the current item (p_i, t_i), and *mark* this item as a repeat reference.

FastSlim(B)

```
/* Input:  B (filter buffer size), program trace {(p_i, t_i)} */
/* Output:  a reduced trace */
/* The filter buffer is empty initially */
for (i = 0; i < length(original trace); i++) {
    if (p_i is in the buffer)
       mark this item, and update its time stamp to t_i;
    else {
       if (FastSlim)
          emit all marked items (preserving time stamp order),
          and unmark them;
       if (buffer is full) {
          if (FastSlim-Demand)
             emit all marked items (preserving time stamp order);
          empty the buffer;
       }
       emit (p_i, t_i) (preserving time stamp order);
       save (p_i, t_i) in the buffer as an unmarked item;
    }
}
```

Fig. 3. The FASTSLIM /FASTSLIM-DEMAND trace reduction algorithm.

(2) If the filter buffer does not contain an item associated with p_i and the buffer is not full (i.e., it holds fewer than B items), then for prefetch-safety (FASTSLIM only), (a) emit all *marked* items in timestamp order and unmark them, (b) emit (p_i, t_i), and (c) place (p_i, t_i) in the buffer as an unmarked item. This item (p_i, t_i) is considered to be the first item of a new *phase*.

(3) If the filter buffer does not contain an item associated with p_i and the buffer is full, then (a) emit all marked items in timestamp order and unmark them, (b) purge the filter buffer, (c) emit (p_i, t_i) and enter it in the filter buffer as an unmarked item. This item (p_i, t_i) is considered to be the first item of a new *epoch* as well as the first item of a new phase.

FASTSLIM is easy to implement and needs only a single pass over the original trace. Note that FASTSLIM defines a partitioning of the trace into *epochs* and *phases*, which are central to the equivalence proof in Section 4. Each epoch is a trace segment that references exactly B distinct pages (without loss of generality we assume throughout this paper that the trace ends on an epoch boundary). Each epoch is further partitioned into B phases, each beginning at an item (p, t) representing the first reference to p in that epoch. Both variants of FASTSLIM retain at least the first item of each epoch and phase in the reduced trace, and thus the epoch and phase partitioning applies equally to the original trace and the reduced trace. Figure 4 depicts a FASTSLIM trace reduction based on the epoch-phase division.

Each item in the reduced trace is important for accuracy-preserving simulation of Class-A

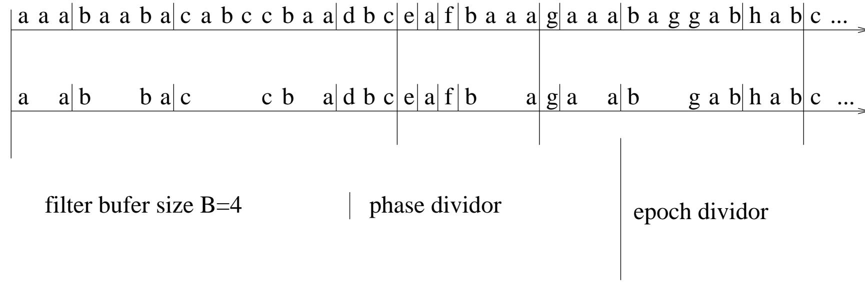

Fig. 4. A FASTSLIM trace reduction and its epoch-phase division.

prefetching and caching schemes as defined in Section 2.3. The reduced trace is accuracy-preserving if and only if it induces a qualifying scheduler to make exactly the same sequence of fetch and replacement choices at exactly the same times as the original trace. Thus the trace reduction algorithm must retain every trace item that could be considered by the scheduler in making its choices. In particular, the reduced trace must retain sufficient information to allow the scheduler to correctly identify:

1. *Holes.* The scheduler must identify the missing page references for a given memory size K and replacement policy (LRU(n) or OPT(n)). It can be shown that each hole must be the first item of a phase provided $B \leq K - n + 1$; FASTSLIM retains these items in the reduced trace.
2. *Replacement candidate sets.* For given memory contents and reference cursor RC, the DO-NO-HARM replacement candidates for a hole are the pages present in memory whose next access beyond the RC is after that hole (otherwise evicting the page would violate DO-NO-HARM). FASTSLIM emits all *marked* items in the filter buffer into the reduced trace at the end of each phase; these items represent the most recent references to any page that could be resident and could have been referenced again since it was brought into memory. Thus the reduced trace always includes the last reference to each page occurring before each hole. This is sufficient to determine DO-NO-HARM exclusions to the replacement candidates set.
3. LRU*(n) or* OPT*(n) replacement candidates.* It can be shown that if $B \leq K - n + 1$, then for any memory size K and any RC position, and any hole whose DO-NO-HARM replacement candidates set is not empty, the reduced trace includes sufficient information to identify and order the LRU(n) or OPT(n) replacement candidates.

Note that the marked items emitted on each phase boundary are needed only to identify DO-NO-HARM exclusions to the replacement candidates set in case the scheduler evicts a page to issue an early fetch. Since demand-paging systems do not replace until the RC reaches a hole, FASTSLIM-DEMAND may omit these items, preserving only the last reference to each page in each epoch, as depicted in Figure 5. Like FASTSLIM, FASTSLIM-DEMAND is simple and compatible with all LRU(n) and OPT(n) replacement policies. FASTSLIM-DEMAND is not prefetch-safe, but it yields better reduction ratios than FASTSLIM. Section 5 shows that FASTSLIM-DEMAND is competitive with SAD, a trace reduction algorithm recenly developed for demand paging algorithms with LRU and OPT.

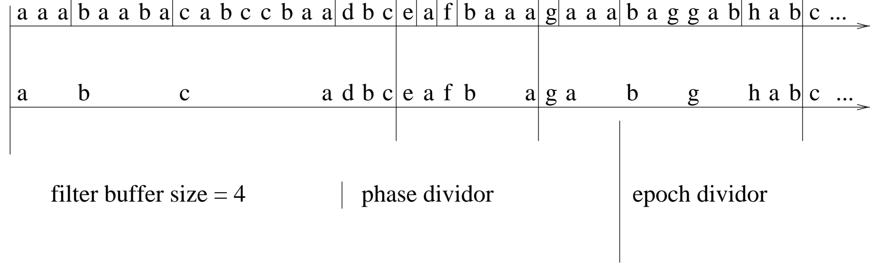

Fig. 5. FastSlim-Demand trace reduction for demand paging systems.

4. EQUIVALENCE PROOF

This section proves that the FastSlim trace reduction algorithm produces exact simulations for Class-A prefetch schedulers as defined in Section 2.2. We also show that FastSlim-Demand produces exact simulations for the subset of Class-A schedulers that use demand paging with either LRU(n) or OPT(n) replacement; we refer to this subset as Class A_d.

The following theorem states these claims formally.

Theorem **Equivalence Theorem**. *Consider a prefetch scheduler $\mathcal{P}$ managing a memory of size K. Let σ be a reference trace, and let σ_B be a reduced trace derived by applying* FastSlim *or* FastSlim-Demand *to σ using a filter buffer of size B. If $\mathcal{P}$ is a Class-A scheduler using* LRU*(n) or* OPT*(n) replacement, σ_B was reduced by* FastSlim*, and $K-n+1 \geq B$, then the action sequence chosen by $\mathcal{P}$ consuming σ_B is identical to the action sequence chosen by $\mathcal{P}$ consuming the original trace σ. If $\mathcal{P}$ is a Class-A_d scheduler using* LRU*(n) or* OPT*(n) replacement, σ_B was reduced by* FastSlim-Demand*, and $K - n + 1 \geq B$, then the action sequence chosen by $\mathcal{P}$ consuming σ_B is identical to the action sequence chosen by $\mathcal{P}$ consuming the original trace σ.*

Proof. The proof is by induction on the action sequence chosen by two identical prefetch schedulers, $\mathcal{P}_o$ and $\mathcal{P}_r$ using the same scheduling algorithm $\mathcal{P}$. $\mathcal{P}_o$ consumes the original trace σ, and $\mathcal{P}_r$ consumes the reduced trace σ_B. We show that if $\mathcal{P}_o$ and $\mathcal{P}_r$ are faced with the same *configuration* — the same initial memory contents, I/O system state, equivalent trace history, and reference cursor position RC, as defined by the t value on the current trace item (p, t) — then they make exactly the same sequence of fetch and eviction choices from that point forward. For the proof, we must assume that $\mathcal{P}$ makes deterministic choices based on the current configuration and the scheduler's internal state, e.g., $\mathcal{P}_o$ and $\mathcal{P}_r$ could be "random" schedulers, but they must use the same initial seed. If $\mathcal{P}$ is truly non-deterministic then the concept of equivalence is not meaningful. We further assume that $\mathcal{P}$ makes all timing decisions based on the t values in the trace items; no accuracy-preserving trace reduction is possible if $\mathcal{P}$ counts the items in the trace.

The base case for the induction is trivial. Assume that $\mathcal{P}_o$ and $\mathcal{P}_r$ face the same initial configuration: an empty memory and an idle I/O system. Since FastSlim always retains the first reference (p_0, t_0) of any trace, the first action of $\mathcal{P}_r$ is the same as $\mathcal{P}_o$, e.g., fetch p_0.

The crucial inductive step is to show that with the same configuration, $\mathcal{P}_o$ and $\mathcal{P}_r$ take the same next action, leaving them again in an identical configuration.

Overview. The proof has two steps. Step 1 shows that $\mathcal{P}_o$ and $\mathcal{P}_r$ take the same next fetch action. Step 2 shows that $\mathcal{P}_o$ and $\mathcal{P}_r$ take the same next eviction action. Step 2 must

be proved separately for LRU(n) and OPT(n) replacement.

The proof of the inductive step relies on three lemmas that define properties of the traces. The Missing Page Lemma states that any hole must be the first item of a phase. The Replacement Candidates Set Lemma states that $\mathcal{P}_o$ and $\mathcal{P}_r$ identify the same set of DO-NO-HARM replacement candidates when faced with the same configuration and the same holes. The Epoch Lemma concerns the location of the next and previous references to replacement candidates relative to the RC, for OPT(n) and LRU(n) replacement. For OPT(n), the Epoch Lemma states that the next reference (after the RC) to any candidate cannot be in the same epoch as the RC. For LRU(n), it states that the last reference (before the RC) to any candidate cannot be in the same epoch as the hole following the RC. These lemmas are formally stated and proved in Section 4.1.

Step 1. We show that the next fetch action taken by $\mathcal{P}_o$ and $\mathcal{P}_r$ fetches the same page at the same time. For a demand-paged (Class-A_d) scheduler, the fetch triggers when the RC advances to the next hole; by the Missing Page Lemma, the hole must occur at the first trace item of a phase, which FASTSLIM-DEMAND retains. Similarly, if $\mathcal{P}_o$ and $\mathcal{P}_r$ are Class-A prefetching schedulers facing the same configuration, then they see the same set of holes; any hole seen by $\mathcal{P}_o$ in σ must be the first item of a phase, and is retained in σ_B. According to the Replacement Candidates Set Lemma, $\mathcal{P}_o$ and $\mathcal{P}_r$ also identify the same set of DO-NO-HARM replacement candidates for each hole. Since $\mathcal{P}_o$ and $\mathcal{P}_r$ make deterministic fetch decisions considering only the missing pages (holes), I/O system state, and eviction opportunities, it follows that they must make the same next fetch decision at the same time, since these conditions are identical for $\mathcal{P}_o$ and $\mathcal{P}_r$.

Step 2. By the reasoning of the previous paragraph, $\mathcal{P}_o$ and $\mathcal{P}_r$ make their next eviction action at the same time, since $\mathcal{P}$ evicts pages only as opportunities arise and it needs memory to issue fetches. The following argument shows that $\mathcal{P}_o$ and $\mathcal{P}_r$ select the same victim page. By the Replacement Candidates Set Lemma $\mathcal{P}_o$ and $\mathcal{P}_r$ identify the same set of replacement candidates at the time of the eviction decision; moreover, this set must be nonempty or no eviction would occur. Suppose the size of the replacement candidates set is m. We are left to show that the items retained in the reduced trace σ_B are sufficient to induce $\mathcal{P}_r$ to identify the best $\min(m, n)$ LRU(n) or OPT(n) replacement candidates from the replacement candidates set, and to maintain proper LRU or OPT ordering among those pages. If this is so, then $\mathcal{P}_o$ and $\mathcal{P}_r$ must choose the same victim page, since they employ the same deterministic replacement policy on identical inputs.

This step of the proof is complicated by the likelihood that the replacement candidates set includes valid DO-NO-HARM candidates that are not among the n best LRU or OPT candidates. Let us call these pages *false candidates*. The *true candidates* are the $\min(m, n)$ pages from the replacement candidates set whose next reference (after the RC) is farthest in the future (for OPT) or whose previous reference (before the RC) is farthest in the past (for LRU). To complete the proof, we must show both that $\mathcal{P}_r$ correctly orders the true candidates, and that it ranks all true candidates before all false candidates. We handle the OPT(n) and LRU(n) cases separately.

OPT(n). We consider the following two cases for OPT(n): (1) the RC is in a previous epoch to the next hole, and (2) the RC is in the same epoch as the next hole.

(1) *The RC is in a previous epoch to the hole.* $\mathcal{P}_r$ assigns a correct OPT ordering to all m candidates. None of the m replacement candidates is referenced between the RC and

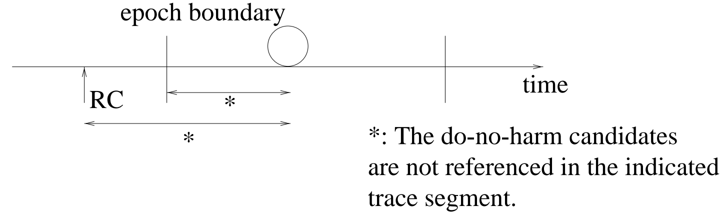

Fig. 6. The DO-NO-HARM replacement candidates are not referenced previously in the same epoch.

this hole, since that would violate DO-NO-HARM. Therefore, the next reference to each candidate after the RC must be the first reference to that page in the epoch in which it occurs. These references are retained in σ_B. See Figure 6 for illustration.

(2) *The RC is in the same epoch as the hole.* $\mathcal{P}_r$ assigns a correct OPT ordering to the true candidates, and ranks all true candidates before all false candidates. By the Epoch Lemma, the next reference (after the RC) to any true candidate cannot be in the same epoch as the RC. Therefore, the next reference to each true candidate after the RC must be in an epoch following the hole, and therefore must be the first reference to that page in the epoch in which it occurs. These references are retained in σ_B. The false candidates may include pages whose next reference is in a subsequent epoch to the RC, and pages whose next reference is in the same epoch. By the reasoning above, the former are correctly ordered. The latter are guaranteed to be ranked after all true candidates because both FASTSLIM and FASTSLIM-DEMAND preserve the last reference to any page in any epoch. All of these references occur before the next reference to any true candidate, which must be in a subsequent epoch.

LRU(n). For LRU(n), we consider Class A (prefetching) schedulers separately from Class A_d (demand paging) schedulers.

(1) FASTSLIM *on Class-A.* $\mathcal{P}_r$ assigns a correct LRU ordering to all m candidates. By the Missing Page Lemma, the next hole must be the first reference of a phase. Therefore, the last reference to each candidate occurring before the hole must occur in a previous phase. Since FASTSLIM always retains the last reference to each page in a phase, the last reference to each candidate is retained in σ_B. Moreover, none of the m replacement candidates is referenced between the RC and the hole, since that would violate DO-NO-HARM. Therefore, the last reference to each candidate before the hole also occurs before the RC.

(2) FASTSLIM-DEMAND *on Class A_d.* $\mathcal{P}_r$ assigns a correct LRU ordering to the true candidates, and ranks all true candidates before all false candidates. By the Epoch Lemma, the last reference to any true candidate cannot occur in the same epoch as the hole. For schedulers using demand paging, the RC points to the hole, so the last reference to each true candidate occurring before the hole must occur in a previous epoch to the RC. Therefore, the last reference to each true candidate occurring before the RC is retained in σ_B, since FASTSLIM-DEMAND preserves the last reference to any page in any epoch. The false candidates may include pages whose last reference is in a previous epoch to the RC, and pages whose last reference is in the same epoch as the RC. By the reasoning above, the former are correctly ordered. The latter are guaranteed to be ordered after

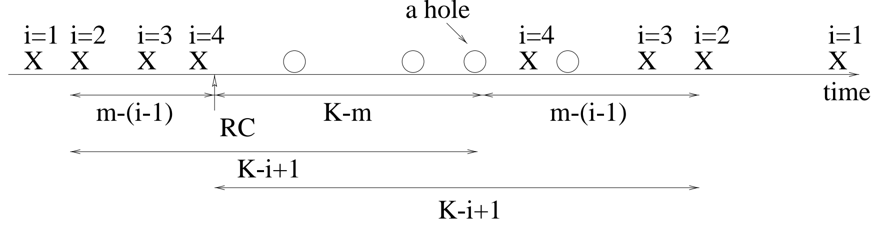

Fig. 7. The minimum distance.

all true candidates because FASTSLIM-DEMAND preserves the first reference to any page in any epoch. All of these references occur after the last reference to any true candidate, which must be in a previous epoch. □

4.1 Lemmas

We now formally state and prove the lemmas supporting the Equivalence Theorem. All of the lemmas except the Replacement Candidates Set Lemma are independent of FASTSLIM or FASTSLIM-DEMAND; they state fundamental properties of the traces and the behavior of Class-A or Class-A_d prefetch schedulers with LRU(n) or OPT(n) replacement on a memory of size K. The Replacement Candidates Set Lemma extends these properties to show that for any given configuration, the trace items that FASTSLIM (or FASTSLIM-DEMAND) retains in σ_B induce $\mathcal{P}_r$ to identify the same set of DO-NO-HARM replacement candidates as $\mathcal{P}_o$. All of the lemmas are based on a preliminary Minimum Distance Lemma.

LEMMA **Minimum Distance Lemma**. *For any page p that is a true* LRU*(n) or* OPT*(n) replacement candidate for any hole and any RC position, the following claim holds:*

LRU*(n). At least $K - n + 1$ distinct pages are referenced between the last reference to p (before the RC) and the hole.*

OPT*(n). At least $K - n + 1$ distinct pages are referenced between the RC and the next reference to p (after the RC).*

Proof. Let m be the size of the DO-NO-HARM replacement candidates set for this hole and this RC. Thus the number of true candidates is $min(m, n)$, and the number of distinct resident pages accessed between the RC and the hole is $K-m$ (by definition of the replacement candidates set). We consider LRU(n) and OPT(n) separately. Refer to the illustration in Figure 7.

LRU*(n).* Consider the i-th best true candidate page p, i.e., the page from the replacement candidates set whose *last* reference before the RC is the i-th farthest in the *past*, where $1 \leq i \leq min(m, n)$. The number of distinct pages referenced between p's last reference and the RC (not including the RC) is $m-(i-1)$. The number of distinct pages accessed between the last reference to p and this hole is therefore $m-(i-1)+(K-m) = K-i+1$, which is no less than $K-n+1$.

OPT*(n).* Consider the i-th best true candidate page p, i.e., the page from the replacement candidates set whose *next* reference after the RC is the i-th farthest in the *future*, where $1 \leq i \leq \min(m, n)$. The number of distinct pages referenced between this hole (not including the hole itself) and p's next reference is at least $m - (i - 1)$. The number of distinct pages accessed between the RC and p's next reference is therefore $m - (i-1) + (K - m) = K - i + 1$, which is no less than $K - n + 1$. □

The Epoch Lemma is a trivial extension of the Minimum Distance Lemma. Here an *epoch* is as previously defined; it is a partition of the trace that references exactly B distinct pages. In the lemma, B is further constrained so that $B \leq K - n + 1$.

LEMMA **Epoch Lemma**. *Consider a trace σ partitioned into epochs of size $B \leq K - n + 1$. For any page p that is a true* LRU*(n) or* OPT*(n) replacement candidate for any hole and any RC position, the following claim holds:*

LRU*(n). The last reference (before the RC) to p cannot be in the same epoch as this hole.*

OPT*(n). The next reference (after the RC) to p cannot be in the same epoch as the RC.*

For the Missing Page Lemma, a *phase* is as previously defined; it is a partition of an epoch such that for its first item (p, t), p is not previously referenced in the epoch, but for all its subsequent items (q, t), q is previously referenced in the epoch. Since each epoch references B distinct pages, it follows that each epoch contains B phases.

LEMMA **Missing Page Lemma**. *Consider a Class-A or Class-A_d scheduler $\mathcal{P}$ consuming a trace σ partitioned into epochs of size $B \leq K - n + 1$, with each epoch containing B phases. For any legal configuration of $\mathcal{P}$ consuming σ, every hole is the first item of a phase.*

Proof. Suppose there is some hole (p, t) that is not the first item of a phase (obviously $RC < t$). We show that this leads to a contradiction. There must exist an item (p, t') referencing p earlier in the epoch (so $t' < t$), or else (p, t) would start a new phase. Further, we know that $t' < RC < t$, i.e., p was brought into memory before the current RC; if this is not so, then by definition (p, t) is not a hole in this configuration, since (p, t') rather than (p, t) is the first reference to the missing page p after RC. Thus p must have been replaced after it was present in memory at time t', but on or before time RC; if this is not so, then p is not missing at time t and so (p, t) is not a hole. Say that p was replaced at some time RC', where $t' < RC' < t$. This means that p was a true replacement candidate for some hole (q, t'') that was visible at time RC' ($t' < RC' < t''$). We know that $t'' < t$, or else the presence of (p, t) would exclude p as a DO-NO-HARM replacement candidate for that hole (q, t''): thus $t' < t'' < t$. This means that (q, t'') is in the same epoch as (p, t') and (p, t). If this is so, then it follows directly from the Epoch Lemma that p could not have been a true replacement candidate for the hole (q, t''), a contradiction. □

The final lemma, the Replacement Candidates Set Lemma, extends the other lemmas to show that correct identification of DO-NO-HARM replacement candidate sets is preserved by FASTSLIM and FASTSLIM-DEMAND.

LEMMA **Replacement Candidates Set Lemma**. *Consider two instances $\mathcal{P}_o$ and $\mathcal{P}_r$ of a Class-A (or Class-A_d) scheduler $\mathcal{P}$ managing a memory of size K. Suppose that $\mathcal{P}_o$ consumes trace σ, and $\mathcal{P}_r$ consumes a reduced trace σ_B derived by applying* FASTSLIM *(or*

FASTSLIM-DEMAND) *to* σ *using a filter buffer of size* B. *Given the same configuration,* $\mathcal{P}_o$ *and* $\mathcal{P}_r$ *identify the same set of* DO-NO-HARM *replacement candidates for each hole.*

Proof. By the Missing Page Lemma, any hole (p, t) identified by $\mathcal{P}_o$ is also identified by $\mathcal{P}_r$: (p, t) is the first item of a phase, and therefore is retained in σ_B. Thus it suffices to show that $\mathcal{P}_o$ and $\mathcal{P}_r$ identify the same set of DO-NO-HARM replacement candidates for any hole that $\mathcal{P}_o$ identifies.

FASTSLIM-DEMAND *on Class* A_d*:*. For any algorithm in Class A_d, all pages in memory are DO-NO-HARM replacement candidates for any hole, since no pages are referenced between RC and the hole. Since $\mathcal{P}_o$ and $\mathcal{P}_r$ are in the same configuration, their memories hold the same set of pages. Therefore, they identify the same set of replacement candidates.

FASTSLIM *on Class* A*:*. Consider any hole (p, t). Each replacement candidate is a resident page that is not referenced between the RC and the hole at t ($RC \leq t$). Since $\mathcal{P}_o$ and $\mathcal{P}_r$ are in the same configuration, their memories hold the same set of pages. Consider any resident page q. If no trace item in σ references q between the RC and t, then $\mathcal{P}_o$ identifies q as a replacement candidate; $\mathcal{P}_r$ also identifies q as a replacement candidate, since σ_B contains no items not also present in σ. Otherwise, some trace item in σ references q between the RC and t ($RC \leq t' < t$). In this case, $\mathcal{P}_o$ does not identify q as a replacement candidate. It remains only to show that $\mathcal{P}_r$ does not identify q as a replacement candidate either.

By the Missing Page Lemma, (p, t) is the first item of a phase. Therefore, (q, t') is in a previous phase, since $t' < t$. Suppose without loss of generality that (q, t') is the last item to reference q in that phase. Then (q, t') occurs in σ_B as well as in σ, since FASTSLIM retains the last reference to each page occurring in each phase. □

The Replacement Candidates Set Lemma shows why prefetch-safe trace reduction must retain the last reference to each page occurring in each phase. For demand-paged schedulers, which never replace until a hole is encountered, it is sufficient to retain only the last reference to each page occurring in each epoch.

5. EFFECTIVENESS OF FASTSLIM

The previous section shows that FASTSLIM yields exact simulations for a wider range of prefetch policies and replacement policies than existing trace reduction schemes. The price of this generality is that the trace reduction is less effective; it must retain more of the references in the reduced trace to guarantee accurate simulations for the wider range of systems. This section quantifies the effectiveness of FASTSLIM and FASTSLIM-DEMAND trace reduction and the cost of their additional generality.

The figure of merit for trace reduction effectiveness is the *absolute reduction ratio*, defined as the number of items in the original trace divided by the number of items in the reduced trace. Since the execution time of a simulation is typically linear with the length of the trace, higher reduction ratios indicate that the trace reduction is more effective in reducing simulation costs as well as storage costs. We can directly determine the marginal benefit of using one trace reduction algorithm rather than another by measuring the ratio of the trace lengths produced by the two algorithms. For this purpose we define the *marginal ratio* (MR) as the length of the larger trace divided by the length of the smaller trace. The relative reduction ratio (RR) is defined as the MR relative to the simple reduction algorithm PREFETCH-SAFE BLOCKING,

which removes all consecutive references to a given block or page while preserving the first and last reference of each run.

To quantify the cost of prefetch-safety and generality of replacement algorithms, we compare the reduction ratios for FastSlim and FastSlim-Demand with sad [Kaplan et al. 1999], one of the most effective trace reduction algorithms known. We chose sad as a baseline because it is easy to implement and has been shown to yield reduction ratios almost as good as olr [Kaplan et al. 1999], which is optimal but limited to demand paging systems with pure lru replacement. Sad guarantees simulation accuracy for demand paging systems with lru or opt replacement.

It is easy to determine theoretical bounds on the relative effectiveness of sad, FastSlim, and FastSlim-Demand. All three algorithms take a filter size parameter (B) and generally yield higher reduction ratios with higher B values. It is easy to show by induction that sad retains at least B references per trace epoch as defined in this paper. FastSlim-Demand retains at most $2B$ references per epoch (the first and last reference to each of the B distinct pages in each epoch). Thus sad is at best twice as effective as FastSlim-Demand, i.e., the marginal benefit of using sad rather than FastSlim-Demand is bounded by an MR of 2. For prefetch-safety, FastSlim retains at most the first and last reference to each of the $i < B$ distinct pages in each phase i of each epoch, thus FastSlim retains at most $B(B+1)/2$ references per epoch. Therefore the marginal benefit of using FastSlim-Demand rather than FastSlim is bounded by an MR of $B/2$, and the marginal benefit of using sad rather than FastSlim is bounded by an MR of B.

The purpose of this section is to determine the relative performance of these trace reduction algorithms in practice on real applications. We selected a representative set of VOOC applications — virtual memory applications with large data set sizes — and measured the effectiveness of the three algorithms for reducing traces generated by those applications. We reduce traces "on-the-fly" by instrumenting Alpha executables with atom [Srivastava and Eustace 1994], injecting calls to trace reduction routines after each data reference. We then execute the programs, counting the references preserved and eliminated by the trace reduction algorithms.

For the applications we studied, sad typically yielded a marginal benefit below 20% (MR below 1.2) over FastSlim-Demand for useful B values, and its effectiveness relative to FastSlim ranged from an MR of five to ten. However, in some cases the gap is larger. For one application — *eigen* — sad is 80% more effective than FastSlim-Demand, and up to 150 times more effective than FastSlim, although this MR ratio was achieved only with high B values that constrain the range of memory sizes for systems simulated using the reduced trace.

5.1 Applications

We applied the three trace reduction algorithms to a representative set of VOOC programs that are diverse in data access patterns and application. Table 3 lists the applications, their storage demands, and their trace lengths. The test applications are drawn from three groups:

*Linear Algebra Programs (*mmm*, *qr *and *eigen*).* mmm implements outer-product matrix multiplication in C. qr and eigen use standard Fortran LAPACK block-computation (submatrix) routines for QR factorization and eigenvalue decomposition, respectively.

APPLICATIONS	DESCRIPTION	MEMORY SIZE (MB)	TRACE LENGTH
MMM	Outer Product Version	100.66	34,380,710,022
QR	LAPACK dgeqr2	33.64	17,223,728,902
EIGEN	LAPACK dgeev2	33.69	162,570,969,590
SORTING	Sorting by Elevation	240.95	3,271,893,292
SWEEPING	Updating Flow Accumulation Information	181.30	3,797,472,966
DES	Discrete Event Simulation	23.89	9,263,562,001
HT	Multiple Hypothesis Testing	133.99	1,032,000,000
RO	Route Optimization	50.78	7,775,621,991
SAR	Synthetic Aperture Radar	34.10	4,839,484

Table 3. VOOC test applications used for the empirical analysis.

APPLICATIONS	BLOCKING absolute ratio	B-RATIO ($B = 1024$)	FASTSLIM absolute ratio	FASTSLIM RR
MMM	1	8.33%	742.49	742.49
QR	1.50	25.94%	442.41	294.94
EIGEN	1.60	24.90%	84.45	52.78
SORTING	1.94	3.48%	3329.54	1716.26
SWEEPING	16.2	4.63%	151.79	9.37
DES	1.64	35.12%	421.87	257.24
HT	1.22	6.32%	2676.01	2193.45
RO	1.19	22.63%	102.39	86.04
SAR	9.81	24.60%	547.89	55.85

Table 4. Effectiveness of FASTSLIM ($B = 1024$) and PREFETCH-SAFE BLOCKING.

TCI Sorting and Sweeping Programs. TCI SORTING and TCI SWEEPING are two phases of a program to compute water flow accumulation and topographical convergence index (TCI) for each point in a terrain given its topographical map (a Digital Elevation Model) and precipitation data. TCI SORTING first sorts all points by elevation; TCI SWEEPING sweeps them in decreasing order, computing flow accumulation in each point, and updating flow into adjacent points. Accesses are not sequential because the map is stored in raster form and the sweep is ordered by elevation. We process a 10M element topographical map of the Sierra Nevada mountains with an approximate data size of 250MB.

*C3I Benchmarks (*DES, HT, RO *and* SAR*).* We selected four benchmarks from a suite representative of military C3I systems (Command, Control, Communication and Intelligence), provided by the U.S. Air Force Rome Laboratory. Each benchmark uses input data supplied with the benchmark; memory demands for these applications are modest, ranging from 24MB to 128MB.

5.2 Experimental Results

Table 4 summarizes the effectiveness of FASTSLIM on the sample applications, relative to PREFETCH-SAFE BLOCKING. The second column of Table 4 shows that BLOCKING typically reduces traces by less than a factor of two, motivating a more sophisticated approach. However, BLOCKING is more effective for applications that make sequential passes over a single

APPLICATION	FASTSLIM-DEMAND		SAD	
	RR	MR_{FASTSLIM}	RR	$MR_{\text{FASTSLIM-DEMAND}}$
MMM	2035.29	2.74	2039.29	1.00
QR	792.46	2.69	794.68	1.00
EIGEN	119.14	2.26	213.34	1.79
SORTING	3844.78	2.24	4085.28	1.06
SWEEPING	34.71	3.71	35.03	1.01
DES	2316.09	9.00	2700.55	1.17
HT	11435.53	5.21	11435.53	1.00
RO	821.51	9.55	951.37	1.16
SAR	114.95	2.06	115.00	1.00

Table 5. Effectiveness of FASTSLIM-DEMAND and SAD ($B = 1024$).

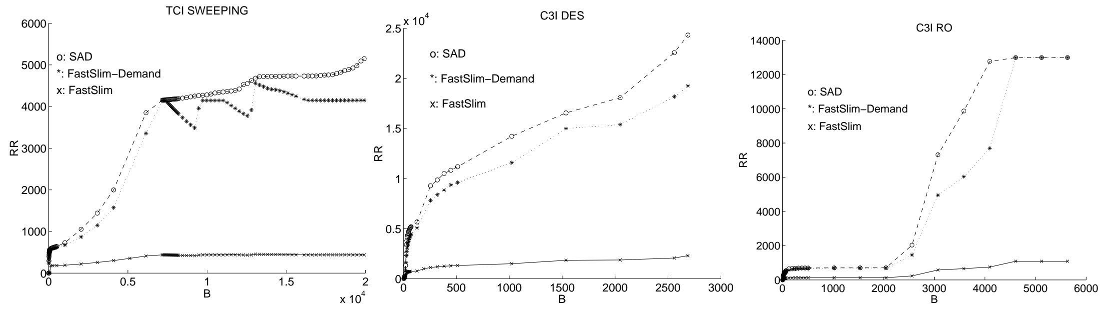

Fig. 8. Applications for which larger B values improve reduction ratios.

dataset; it reduces the SWEEPING and SAR traces by factors of 16.2 and 9.81 respectively.

The last two columns of Table 4 show that FASTSLIM is significantly more effective than BLOCKING. For these experiments we used a representative B value of 1024 8KB pages (total 8MB); the third column gives the percentage of the application's total memory demand represented by this B value. With $B = 1024$, FASTSLIM reduces traces by a factor of 10^2 to 10^3 for eight of the nine applications studied, with a peak absolute reduction ratio of 3329 (for SORTING); only EIGEN yields a ratio below 100. The last column of Table 4 gives relative ratio (RR) of FASTSLIM , showing that FASTSLIM is more effective than BLOCKING by factors ranging from 10 to 2193.

Although FASTSLIM is effective, Table 5 shows that the cost of prefetch-safety is significant. The second and fourth columns give the relative reduction ratios for FASTSLIM-DEMAND and SAD relative to BLOCKING, as in the last column of Table 4. The third and fifth columns give the marginal ratio of each algorithm relative to its closest competitor; the third column gives the MR of FASTSLIM-DEMAND relative to FASTSLIM, and the fifth column gives the MR of SAD relative to FASTSLIM-DEMAND. The cost of prefetch-safety is given by column 3, which shows that FASTSLIM-DEMAND reduces traces by as much as a factor of 9.55 more than FASTSLIM, although ratios of 2-3 are typical. The cost of generality with respect to replacement policies is much lower: the marginal benefit of using SAD rather than FASTSLIM-DEMAND is typically insignificant and is greater than 17% only for *eigen*.

Since effectiveness varies with the value of the B parameter for all of the algorithms, we compare reduction ratios for different values of B. Figures 8 and 9 show the relative reduc-

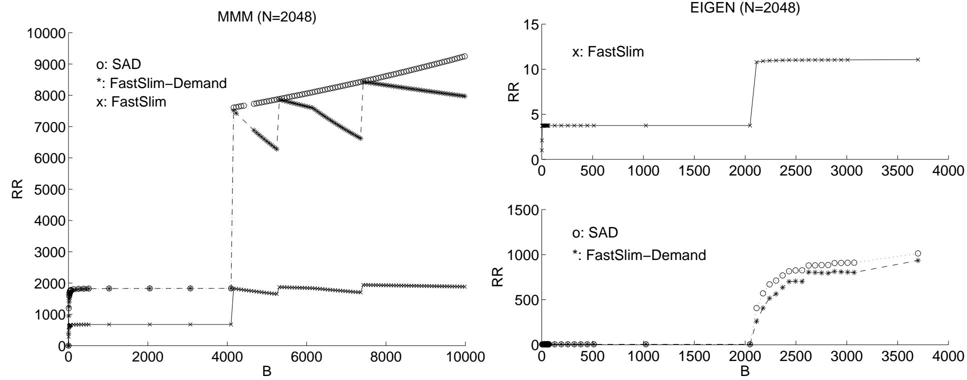

Fig. 9. Applications for which increasing B yields irregular improvements.

tion ratios as a function of B for a full range of B parameters ranging up to 90% of each application's total data size. There is no value to extending the B values beyond this range, since the reduced traces could be used only to simulate systems in which the application fits in memory. For these experiments we measured reduction ratios only for the first $250M$ references. We found that all the applications touch over 99% of their pages in the first $250M$ references (except for RO, whose prefix coverage is 72.99%), and that the reduction ratio for the first $250M$ references conservatively approximates the ratio for the entire trace.

We divide the applications into groups according to the effect of the B parameter on the reduction ratios achieved. Figure 8 shows that SWEEPING, DES, and RO yield reduction ratios that improve with B up to the limit of 90% of the application's memory demand. Figure 9 shows that MMM and EIGEN yield reduction ratios that increase suddenly at various thresholds. The remaining four applications (QR, SORTING, HT, and RO) are not shown because their reduction ratios are roughly constant at the values in Table 4 and 5, independent of B.

Note that these results give insight into the locality behavior of the applications. The applications in Figure 8 will generally do less I/O if supplied with more memory; pages brought into memory are referenced a larger number of times before eviction. On the other hand, applications whose reduction ratios do not improve with larger B values show poor locality; the application tends to touch a larger portion of its data before re-referencing any given page. The discontinuities in Figure 9 correspond to natural working set thresholds, in which the application makes repeated references to some set of its pages. B values larger than the working set size allow many of these references to be omitted from the trace, as they would never cause I/O in systems with memories large enough to retain the working set.

The results show that FASTSLIM-DEMAND is competitive with SAD across the full range of B values for all of the applications studied. However, the graphs show that for six of the applications the prefetch-safety constraint limits the benefit of larger B values for FASTSLIM. This is because FASTSLIM retains the last reference to each page before each phase boundary; while a higher B value reduces the number of epochs, it does not reduce the number of phases. Thus the performance gap between FASTSLIM and FASTSLIM-DEMAND (or SAD) tends to grow with larger B values.

To understand this effect better, Table 6 presents the marginal benefit of FASTSLIM-DEMAND relative to FASTSLIM for selected B values covering fixed percentages of each ap-

Application	10%	50%	90%
mmm	2.70	4.14	3.93
qr	3.00	3.02	3.21
eigen	1.50	1.51	84.50
sorting	3.28	2.84	2.85
sweeping	4.02	9.45	9.46
des	4.03	5.79	4.84
ht	2.60	2.61	4.92
ro	1.95	3.12	4.36
sar	2.06	2.06	2.06

Table 6. Marginal benefit of FastSlim-Demand relative to FastSlim for various B coverages.

plication's total memory demand. Table 6 shows that although FastSlim's reduction ratios grow more slowly with larger B values, the marginal benefit of using an algorithm that is not prefetch-safe — such as FastSlim-Demand — tends to stay constant across the full range of B values. However, two of the applications — *sweeping* and *eigen* — show diverging behavior with large B values in which prefetch-safety inhibits FastSlim from improving with larger B values. For *eigen*, the marginal benefit of FastSlim-Demand leaps to a factor of almost 85 as the B value reaches roughly 75% of the memory demand. This effect occurs only at B values large enough to significantly constrain the range of systems that can be studied, i.e., the highly reduced trace would contain only enough information to simulate systems in which at least 75% of the application's data fits in memory. For more representative B values, the cost of prefetch-safety, as measured by the marginal ratio, tends to be roughly constant for each application.

6. CONCLUSION

This paper presents FastSlim, a trace reduction algorithm that yields accurate simulations for a broad class of integrated prefetching and caching systems. Based on our analysis, FastSlim can be extended to systems with parallel disks and certain types of storage hierarchies such as network memory. Our experiments show further that the reduction ratios, with the prefetch-safe constraint, ranges from 10 to 10,000 for the applications we studied. In other words, FastSlim is accurate and effective.

A second contribution of our work with FastSlim is that it shows how trace reduction can handle a wider range of replacement policies easily. FastSlim is parameterized to accommodate progressively less effective lru(n) and opt(n) replacement schemes (using larger n values) with progressively less effective trace reduction (using smaller filter buffers). We introduce a variant of FastSlim, called FastSlim-Demand, to show that this contribution extends to the demand paging systems targeted by earlier trace reduction schemes. In particular, we show that FastSlim and FastSlim-Demand accommodate widely used lru approximations (e.g., fifo-with-second-chance).

Thus FastSlim is more general than existing trace reduction schemes along two dimensions: prefetching policy and replacement policy. The tradeoff of this generality is that FastSlim is less effective than the best existing schemes, i.e., it retains more references in the reduced trace in order to produce exact simulations for a wider range of target systems. Even so, prefetch-safe FastSlim reduces traces by *two* to *three* orders of magnitude for a representative set of

virtual out-of-core applications.

We conduct a quantitative comparison of FASTSLIM, FASTSLIM-DEMAND and SAD, the most effective trace reduction scheme known for demand paging systems with LRU or OPT replacement. While FASTSLIM may underperform SAD by an order of magnitude, factors of two to five are more typical for the applications we studied. On these traces, FASTSLIM-DEMAND is competitive with SAD while handling a wider range of replacement policies. This shows that while prefetch-safety is expensive, generality with regard to replacement policies is not.

REFERENCES

ALBERS, S., GARG, N., AND LEONARDI, S. 1998. Minimizing stall time in single and parallel disk systems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)* (New York, May23–26 1998), pp. 454–462. ACM Press.

CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. 1995. A study of integrated prefetching and caching strategies. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, May 1995), pp. 188–197. ACM Press.

CHANG, F. AND GIBSON, G. A. 1999. Automatic I/O hint generation through speculative execution. In *Operating Systems Design and Implementation (OSDI '99)* (1999), pp. 1–14.

COFFMAN, E. G. AND RANDELL, B. 1971. Performance predictions for extended paged memories. *Acta Informatica 1*, 1–13.

DRAVES, R. P. 1990. Page replacement and reference bit emulation in Mach. In *Proceedings of the Usenix Mach Symposium* (Nov. 1990).

FEELEY, M. J., MORGAN, W. E., PIGHIN, F. H., KARLIN, A. R., AND LEVY, H. M. 1995. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)* (Dec. 1995), pp. 201–212.

GLASS, G. AND CAO, P. 1997. Adaptive page replacement based on memory reference behavior. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Volume 25,1 of *Performance Evaluation Review* (New York, June 15–18 1997), pp. 115–126. ACM Press.

KAPLAN, S. F., SMARAGDAKIS, Y., AND WILSON, P. R. 1999. Trace reduction for virtual memory simulations. In *ACM SIGMETRICS* (May 1999).

KIMBREL, T. AND KARLIN, A. R. 1996. Near-optimal parallel prefetching and caching. In *37th Annual Symposium on Foundations of Computer Science* (Burlington, Vermont, 14–16 Oct. 1996), pp. 540–549.

KIMBREL, T., TOMKINS, A., PATTERSON, R. H., BERSHAD, B., CAO, P., FELTEN, E. W., GIBSON, G., KARLIN, A. R., AND LI, K. 1996. A trace-driven comparison of algorithms for parallel prefetching and caching. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), Seattle, WA* (Berkeley, CA, USA, Oct. 1996), pp. 19–34.

KROEGER, T. M. AND LONG, D. D. E. 1996. Predicting future file-system actions from prior events. In *Proceedings of the USENIX 1996 annual technical conference, San Diego, California, USA* (Berkeley, CA, USA, Jan. 1996), pp. 319–328.

MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2, 78–117.

MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. 1996. Automatic compiler-inserted I/O prefetching for out-of-core applications. In USENIX Ed., *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA* (Berkeley, CA, USA, Oct. 1996), pp. 3–17. USENIX.

PATTERSON, D., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record 17*, 3 (Sept.), 109–116.

PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995), pp. 79–95. ACM Press.

Puzak, T. R. 1985. *Analysis of Cache Replacement Algorithms.* Ph. D. thesis, University of Massachussetts, Department of Electrical and Computer Engineering.

Samples, A. D. 1989. Mache: No-loss trace compaction. In *ACM SIGMETRICS* (May 1989), pp. 89–97.

Smith, A. J. 1977. Two methods for the efficient analysis of address trace data. *IEEE Transactions on Software Engineering SE-3*, 1 (Jan.).

Srivastava, A. and Eustace, A. 1994. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices 29*, 6 (June), 196–205.

Trivedi, K. S. 1976. Prepaging and applications to array algorithms. *IEEE Transactions on Computers C-25*, 9 (Sept.), 915–921.

Vitter, J. S. and Krishnan, P. 1996. Optimal prefetching via data compression. *Journal of the ACM 43*, 5 (Sept.), 771–793.

Voelker, G. M., Anderson, E. J., Kimbrel, T., Feeley, M. J., Chase, J. S., and Karlin, A. R. 1998. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-98/PERFORMANCE-98)*, Volume 26.1 of *ACM Performance Evaluation Review* (New York, June 22–26 1998), pp. 33–43. ACM Press.