Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories

James G. Thompson
Alan Jay Smith

Computer Science Division
Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

For the class of replacement algorithms known as *stack algorithms*, existing analysis techniques permit the computation of memory miss ratios for all memory sizes simultaneously, in one pass over a memory reference string. We extend the class of computations possible by this methodology in two ways. First, we show how to compute the effects of copy-backs in write-back caches. (The key observation here is that a given block is clean for all memory sizes less than or equal to C blocks and is dirty for all larger memory sizes.) Our technique permits efficient computations for algorithms or systems using periodic write-back and/or block deletion. The second extension permits stack analysis simulation for sector (or sub-block) caches, in which a sector (associated with an address tag) consists of sub-sectors (or sub-blocks) which can be loaded independently. (The key observation here is that a sub-sector is present only in caches of size C or greater.) Load forward prefetching in a sector cache is shown to be a stack algorithm and is easily simulated using our technique. Running times for our methods are only slightly higher than for a simulation of a single memory size using non-stack techniques.

*The material presented here is based on research supported in part by the National Science Foundation under grant CCR-8202591, by the IBM Corporation under a computer equipment grant to UC Berkeley, and by the State of California, the Hewlett Packard Corporation, the IBM Corporation, and the Signetics Corporation under the MICRO program.

1. Introduction

Analysis of memory system performance is one of the most important aspects of computer system design. Frequently this analysis is done using the technique of trace-driven simulation, where a trace of memory references from a similar system is used as input to a simulation of the system under study. This technique has been applied to all levels of the memory hierarchy, from microprocessor caches to file system design.

Trace driven simulation became much more practical with the discovery by Mattson, Gecsei, Slutz, and Traiger [Matt70] that, for certain replacement policies, the performance of *all* memory sizes could be determined with a single pass through the trace file. Their *stack analysis technique* relies on the inclusion property of these policies, such that the contents of any size memory is a subset of the contents of any larger memory. Thus, the contents of all memories can be represented by a stack, where the top k items in the stack are the contents of a memory of size k. Policies which obey this property are known as *stack algorithms*. An equivalent characteristic of stack algorithms is that each possess a total priority ordering of all blocks at any instant in time which is independent of memory size.

Until now, stack analysis has not been known to apply to some important situations, forcing the designer to fall back on the one-size-at-a-time method. One example of this is the write-back policy, where a write to a block causes the block to be marked "dirty" in the cache, but the write to secondary storage is delayed until some later time. Write-back is particularly desirable where memory bandwidth may be a limiting resource, such as in a shared-bus multiprocessor. The alternative policy, write through, where all modifications go directly to secondary storage, severely restricts the performance improvement due to caching. Even with write-back, in many cases a write can cause twice the memory accesses of a read; one to fetch the block prior to modification and another to rewrite the block (copy-back). However, discussions of stack analysis have either ignored writes altogether, or considered only write through.

The problem with stack analysis of write-back is that it appears to violate the inclusion property. For example, suppose that a dirty block which is at level k in the stack is read. It must come to the top of the stack, but it is now clean for some sizes and dirty for others. We will show that by maintaining a "dirty level" for each block that the stack analysis technique can be extended to analyze write-back. This dirty level is the smallest memory in which the block is dirty. This is the lowest level in the stack to which the block has been pushed since its last write, or infinity if the block has never been written.

Stack analysis can be similarly extended to analyze sector or sub-block caches [Hill84, Lipt68]. In a microprocessor cache, access time, on-chip data path widths, and pin-counts favor small blocks, whereas the size of associative lookup circuitry and tags favor fewer, larger blocks. A possible compromise is to break each block into independent sub-blocks, any or all of which may be present. Again, we will show that stack analysis can be applied by maintaining additional data with each block.

The remainder of this section is a review of previous stack analysis techniques and definition of terms. Section 2 presents the stack algorithm for write-back, followed in Section 3 by several simple extensions to handle deletions, periodic write-back, and cache flush. Section 4 similarly presents the algorithm for sector caches, including an extension for a useful form of prefetch. Section 5 presents a comparison of the time required to perform analysis using the stack technique. Finally we will show how stack analysis makes possible other useful measurements, exemplified by a computation of the probability of a write-back as a function of memory size.

1.1. Cache Memories

Analysis of memory systems has been an important part of the design of computer systems. The early concentration on virtual program memory [Bela66] has been replaced by recent emphasis on high-speed processor caches, and by file-system buffering or caching. In reality, a typical memory system can be viewed as a hierarchy of memory types, where the upper levels of the hierarchy are faster, but generally more costly. The goal of the memory system designer is to find the proper mix of memory types and sizes to provide a desired effective access time, subject to cost and design constraints. Although many of the analysis techniques discussed generalize to multi-level hierarchies, for the remainder of this paper we restrict our discussions to a two-level hierarchy, and refer to the top level as the *cache*.

Caches are effective because of the *principle of locality* [Denn72]. This principle says that the items most likely to be referenced next are those "near" the items which have been recently referenced. The two aspects of locality are "temporal" locality and "spatial" locality. Temporal locality implies that an item which has been recently referenced has a good chance of been referenced again in a short time. Spatial locality implies that items close to a referenced item are also likely to be referenced. This is particularly evident in the sequential reference behavior observed in instructions and within files.

There are a large numbers of design parameters to any cache, most of which must be considered in any analysis of that design. We briefly present definitions of a number of these. For more detail see [Smit82].

Blocking

The cache may be divided into fixed-size *blocks*, or variable-size *segments*. Unless otherwise specified we assume blocks, although most of the algorithms presented can be generalized to analyze segment caches. Blocks are also referred to as *pages* in the context of virtual memory, and *lines* or *sectors* in the context of processor cache. The cache block or line size may be equal to the amount of data retrievable in one memory cycle, or several memory cycles may be required to fetch a block. A larger block size reduces per-block overhead and provides a form of prefetch, discussed below.

Replacement Policy

In fixed memory-space systems, the replacement policy determines which block to remove when the cache is full and a new block must be fetched. Commonly used policies include the Least Recently Used (LRU) policy, First-In First-Out (FIFO), Least Frequently Used (LFU), and Random (RAND). An optimal policy, MIN, exists, but is unrealizable in practice because it requires knowledge of the future [Matt70]. The MIN policy does not consider writes or deletes, and is known to be non-optimal if writes are considered [Yu76]. There are also variable space algorithms, where the replacement policy more generally determines when a block is to be removed. We are concerned here only with fixed space policies.

Write Policy

The write policy determines when a modification is presented to the secondary storage. Writes may always go directly to the secondary storage using the *write through* policy. Alternatively, the write may go to the cache to be written at some later time, usually when the block is about to be replaced, which is called *write-back* or *copy-back*. Clearly write-back can never cause more accesses than write through, and usually far fewer. On the other hand, since it deals in blocks rather than words, write-back may increase the number of bytes written. It also requires that cache consistency be considered if several caches can share access to secondary storage, as is the case in a multi-processor bus. In addition, dirty blocks may remain in the cache for a long time, leading to reliability issues in large volatile caches. The decrease in memory traffic from write-back makes it very valuable in systems with limited memory bandwidth, such as shared-bus multiprocessor systems.

Fetch on Write

If a block becomes resident in the cache due to a write (which it may for either write-back or write through), it may be necessary to first fetch the block from secondary storage. This *fetch on write* is needed, for example, if only a portion of the block is being modified by the write. The

alternative is to keep track of the portion(s) of the the cache block which are "valid", which becomes difficult when several disjoint portions of the block are written. Even when fetch-on-write is in effect, there are situations where it can be avoided. Two examples are when the entire block is being overwritten, or when the contents of the rest of the block are predictable, such as when the block is a "new" block in a file system.

Prefetch

Because of spatial locality, a reference to a block often implies that the next physical block will soon be referenced. It is possible to take advantage of this anticipated reference and to *prefetch* the next block in advance. This reduces the delay when the next block is actually referenced. Prefetch is advantageous when it can be overlapped with processing of other references, or when two or more blocks can be fetched in much less time than all of them individually, as is the case with disk secondary storage. While it reduces the delay, prefetch will increase memory traffic unless all prefetched blocks are referenced before they are replaced. It may also result in *memory pollution*, where a soon-to-be-referenced block is displaced to make room to prefetch an unnecessary block [Smit78b].

1.2. Metrics

The performance of a memory system can be measured in several ways. Perhaps the most common is the *miss ratio*, which is the fraction of references which were not satisfied by the cache. Conversely, the hit ratio is the fraction which were satisfied by the cache. The miss ratio is a latency metric since it determines the apparent access time of the memory system. For a multi-level hierarchy, the effective access time is given by $\sum t_i h_i$, where t_i is the access time to the ith level, and h_i is the hit ratio to the ith level. The access time to each level includes any queuing delays.

The actual computation of miss ratios during simulation varies with the parameters of the cache. Let N be the total number of references, and $m(C)$ be the number of misses in a cache of size C. If all references are assumed to be reads, then the miss ratio for a cache of size C is given by

$$MR_R(C) = m(C)/N \tag{1.1}$$

hence the name.

With write through, where every write is a "miss" (i.e. causes an access to secondary storage), the miss ratio is

$$MR_{WT}(C) = (m_r(C)+W)/N \tag{1.2}$$

where $m_r(C)$ is the number of reads which "miss", and W is the number of write

references.

When write-back is used, this becomes

$$MR_{WB}(C) = (m_r(C)+dp(C))/N \tag{1.3}$$

where $dp(C)$ is the number of dirty blocks "pushed" from a cache of size C.

If write fetch is also considered, a write could result in two accesses to secondary storage, one to fetch the block and another later to write it. The miss ratio is now given by

$$MR_{WBWF}(C) = (m(C)+dp(C))/N \tag{1.4}$$

where we have used the fact that a write fetch is actually just a read reference and occurs if the block reference "misses".

All of the expressions so far assume that the processor must wait for the write to secondary storage to complete before continuing. It is often reasonable to buffer the writes so that the processor can continue almost immediately. In this case delay occurs only if there are enough accesses to create contention. (In [Smit79], it is observed that when memory bandwidth is sufficient, four store-through buffers are sufficient to largely eliminate queuing for writes.) Under this assumption, the write-back miss ratio with write fetch is again simply

$$MR_{WF}(C) = m(C)/N. \tag{1.5}$$

A related metric is the *traffic ratio*, which is the ratio of traffic between cache and secondary storage compared to the traffic which would be present without a cache [Hill84]. The traffic ratio is increasingly important for analyzing shared bus systems such as multi-processor architectures or a network file system. Although buffering may eliminate write-back from consideration in the miss ratio, the write traffic is not eliminated, so writes must be considered in the traffic ratio. Also, prefetch may result in increased traffic, since some prefetched blocks may not actually be referenced.

The traffic ratio is dependent on the same factors as the miss ratio and, in addition, depends on the size of data blocks transferred. Suppose that the processor accesses B_p bytes per average memory reference. The traffic without a cache is then B_p times the number of references. Frequently the cache block size, B_c is larger than B_p. We assume that each cache miss causes B_c bytes to be transferred. Then a large cache block size may act as a form of prefetch and reduce the miss ratio, but it may also increase the amount of traffic.

The general form of the traffic ratio computation is

$$TR(C,B_c) = [m_r(C)+m_w(C)*Wf+f(C)+dp(C)]*B_c/N*B_p \quad (1.6)$$

where Wf is 1 if write fetch is used, 0 otherwise; $m_w(C)$ is the number of write misses (i.e. write fetches); and $f(C)$ is the number of prefetches. Notice that the traffic ratio is identical to the miss ratio when there is no prefetching, no write buffering, and the cache block size is the same as B_p.

A third important metric is the *transfer ratio*, which is the ratio of secondary storage accesses with and without cache. This metric has also been called the *transaction ratio* [Gibs86], and the *swapping ratio* [Kubo75]. The amount of data transferred per access may be fixed (e.g. one double-word per memory cycle) or variable (e.g. disk I/O). Thus, for example, a disk cache can fetch two blocks with one I/O, potentially increasing the traffic, but without increasing the number of accesses. The transfer ratio is similar to the traffic ratio, but is more appropriate for those systems where there is a relatively high cost for a memory access, but a small incremental cost per unit of data transferred. This is typically the case for disk and local area network accesses, within certain limits on message size. The transfer ratio also affects the access time if there are enough transfers to create contention, particularly in multiple processor systems with shared memory. A general expression for the transfer ratio is

$$T(C) = [m_r(C)+m_w(C)*Wf + dp(C)]/N \quad (1.7)$$

which is almost proportional to the traffic ratio with constant block sizes.

1.3. Trace Driven Simulation

One common method to calculate these metrics is to use trace-driven simulation. Memory references are gathered from a system assumed to be similar to the system being modeled. These events are then used to drive a simulation of the system under study with varying design parameters. To the extent that the traces apply to the modeled system, simulation is a relatively simple way to observe the effect of changes to the memory hierarchy. Unfortunately, it could take a large number of simulations if only a single combination of memory sizes could be simulated at a time.

In a classic paper, Mattson, et al. showed that for certain replacement policies, the miss ratios for all memory sizes could be calculated in a single pass of the reference trace [Matt70]. These policies are collectively known as *stack algorithms*. The technique depends on the *inclusion property* of these policies. For these algorithms, the memory at any time can be represented as a stack, with the most recently referenced on top. The upper k elements of the stack are the blocks present in a memory of size k. The current stack level of any block is therefore the minimum memory capacity for which the block is resident. If a

block is referenced while at level k, it is a "hit", and therefore resident, for all sizes k and larger. The level at which the block is found is referred to as its *stack distance*. Using stack analysis, it is possible to compute the miss ratio of equation (1.1) for all sizes by recording the hits to each level. The miss ratio for a cache size C is:

$$MR_R(C) = (N - \sum_{i=1}^{C} hits(i))/N \tag{1.8}$$

where N is again the total number of references. Notice that, since $hits(i)$ is never negative, this is a non-increasing function of memory size. All stack algorithms possess this characteristic, whereas non-stack algorithms may show points at which performance declines with increased memory [Bela69].

The simplest example of a stack algorithm is the Least Recently Used (LRU) policy. The stack always contains the blocks in order of last reference, with the most recently referenced block on the top. For any memory size C, the LRU block for that memory size is the block at level C in the stack. When a block at level k is referenced, it is not in any memory smaller than k, and therefore it must be fetched. The block which must be removed from any memory of size j, j smaller than k, is the block at level j. The stack is updated by simply "pulling" the referenced block out of the stack and placing it on top. All blocks down to level k are effectively "pushed" down one level. Since the referenced block was in all memories k or larger, all blocks below level k remain unchanged. Figure 1.1 illustrates these operations for the case where the referenced block is at stack level 4, and the case where the block is not currently in the stack.

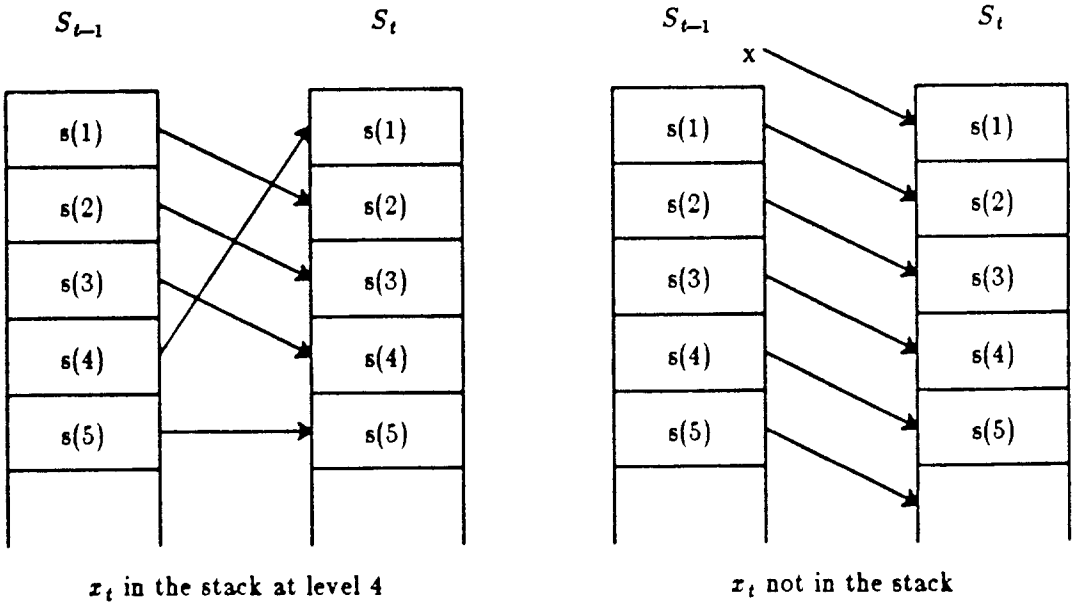


Figure 1.1 Examples of stack maintenance using LRU replacement.

More generally, Mattson, et al. [Matts70] showed that any stack algorithm possesses a "priority function" which imposes a total ordering on all blocks at any given time, independent of memory size. Notice that LRU imposes such an ordering based on the time of last reference. However, in the more general case the relative priority of two blocks may change without either of them being referenced. It is no longer the case that the block at level j is necessarily the one to be pushed from that size memory. This complicates the stack update procedure, but only slightly. First, the referenced block must still be pulled to the top of the stack, and the block which was at the top must be pushed. In the memory of size two, either the block at level two or the block pushed from level one will be pushed, depending on their relative priorities. (Ties are broken by some arbitrary rule.) In the three-block memory, either the block at level three or the one pushed from size two will be pushed. The block which was not pushed from level two can not have the lowest priority of the three blocks, since is is known to have a higher priority than the one which was pushed. Similar logic applies for all levels down to level k, the original level of the referenced block; only the block currently at the level and the one pushed from above need to be compared to find the block to be pushed. The contents of all sizes larger than k are again unchanged. Figure 1.2 illustrates the operations required to maintain the stack.

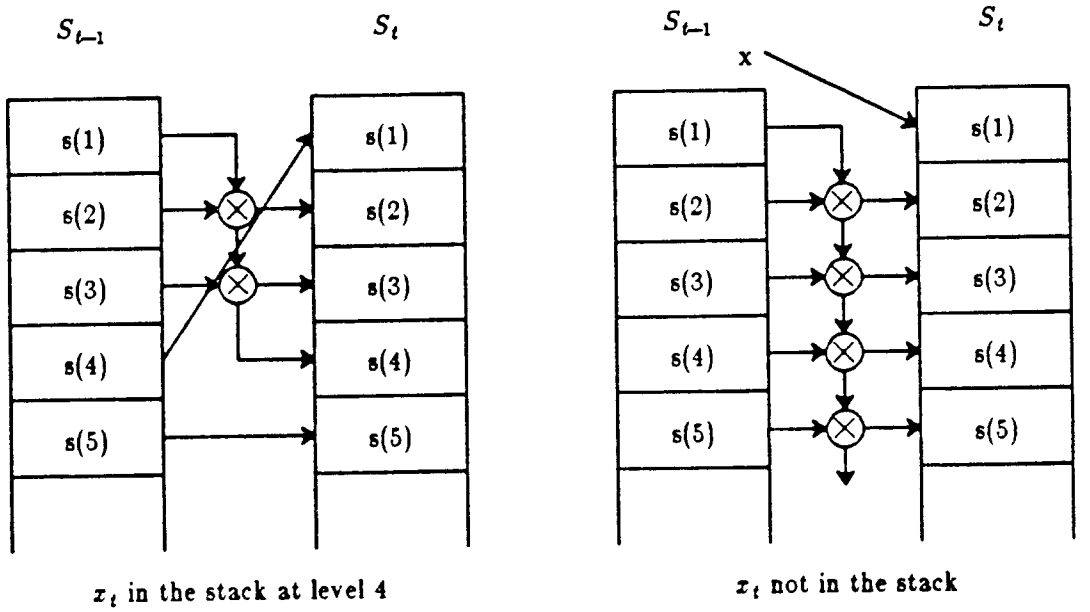


Figure 1.2 Examples of stack maintenance using a stack replacement algorithm.

The stack analysis algorithm is formally presented below. This algorithm will be used as the basis for the extensions in Sections 2 and 4. Let:

X = $x_1, x_2 \ldots x_N$ be a trace, where x_t is the reference at time t.

S_t = the memory stack, with $s_t(C)$ = the block at stack level C.
$s_0(C) = \phi$ for all C.

Δ = the stack distance of x_t, that is, $s_{t-1}(\Delta)=x_t$

$y_t(C)$ = the block pushed from memory of size C.

$rh_t(C)$= a count of the number of hits to level C.

ALGORITHM 1: GENERAL STACK ANALYSIS ALGORITHM

1.	FOR $1 \le t \le N$ DO	*For all events*
2.	IF $x_t \notin S_{t-1}$ THEN $\Delta=\infty$	*If not referenced before.*
3.	ELSE DO	
4.	FIND Δ SUCH THAT $s_{t-1}(\Delta)=x_t$	*Find the stack distance*
5.	$rh_t(\Delta)=rh_{t-1}(\Delta)+1$	*Update the read hits*
6.	IF $\Delta \neq 1$	*If the stack needs updating*
7.	$y_t(1)=s_{t-1}(1)$	*Calculate push set.*
	FOR $2 \le i < \Delta$ DO $y_t(i)=pmin[y_t(i-1), s_{t-1}(i)]$	
	FOR $i \ge \Delta$ DO $y_t(i)=\phi$	
8.	$s_t(1)=x_t$	*Establish new stack.*
	FOR $i>1$ DO $s_t(i)=s_{t-1}(i) + y_t(i-1) - y_t(i)$	

NOTES:

1. In step 5, all counts which are not incremented are assumed to remain unchanged at the next time interval, i.e. $rh_t(i)=rh_{t-1}(i), i \neq \Delta$.
2. In step 7, *pmin* returns the block with the lowest priority, as defined by the replacement algorithm.
3. In step 8, plus and minus have the intuitive meaning of adding a member to a set, or removing a member. In this context, adding a member which is already present, or subtracting a member which is not present, have no effect. The same is true of adding or subtracting ϕ.

Notice that, in practice, it is possible to search the stack for the referenced block and update the stack simultaneously, since the priority function can not depend on where (or even if) the referenced block is in the stack. The update stops when the referenced block is found. The block being pushed takes the place of the referenced block, which is inserted on top of the stack.

As an example, consider the application of the Least Frequently Used policy to the reference string {**AAABBCCDB**}. The contents of the stack after each reference are shown in Figure 1.3, where the number beside each block is the

priority, i.e. the number of uses of the block. Notice that a block may be pushed several levels because of a reference, as seen at time 8. Note too that blocks below the level where the referenced block is found are unchanged, even though they may have higher priorities, as seen after the last reference.

Time	1	2	3	4	5	6	7	8	9
Reference String	A	A	A	B	B	C	C	D	B
	A1	A2	A3	B1	B2	C1	C2	D1	B3
Memory				A3	A3	A3	A3	A3	A3
Stack						B2	B2	B2	D1
								C2	C2

Figure 1.3 Memory contents using Least Frequently Used policy

1.4. Non-Stack Algorithms

The prohibition against a priority function which depends on memory size prevents some otherwise simple policies from being stack algorithms, such as the First-In First-Out (FIFO) rule [Matt70]. Another common technique which is not a stack algorithm is the use of *prefetch*. Suppose that the prefetch policy is to fetch the following block along with any fetched block, but not to prefetch if the referenced block is already present. This is typical of a *demand fetch* policy, where no fetch should take place unless the referenced block is missing. Assume an arbitrary stack algorithm for replacement. It is easy to construct counter-examples which violate inclusion, because the priority of a prefetched block depends on when it is fetched, which varies with memory size. For example, consider the examples of Figure 1.4, where the contents of a larger memory is clearly not a subset of a smaller memory after the final reference.

It is possible to construct prefetch policies which are stack algorithms. For example, the policy which always prefetches the next block, regardless of whether the referenced block is resident, is a stack algorithm. This policy is a form of One Block Lookahead, or OBL [Smit78b]. From the point of view of the stack this is equivalent to the insertion of a reference to the next block after each reference. However, this is only a practical policy when it is possible to test for whether the next block is in the cache simultaneously with the reference to the current block. Otherwise the cache must do double the work of searching and updating the cache.

Time		1	2	3
Reference		A	C	A
	Size			
	1	A	C	A
Memory	2	AB	CD	AB
Contents	3	AB	CDA	ACD

(a)

Time		1	2	3	4	5
Reference		A	B	C	A	D
	Size					
	1	A	B	C	A	D
Memory	2	AB	BA	CD	AB	DE
Contents	3	AB	BA	CDB	ABC	DEA
	4	AB	BA	CDBA	ACDB	DACB

(b)

Figure 1.4 Memory contents using one block prefetch always, showing lack of inclusion.

A practical prefetch policy which obeys inclusion is described by Horspool and Huberman [Hors83]. Their algorithm adds the condition that the following block (x_t+1) is only prefetched if its stack distance is less than that of the referenced block (x_t). This prevents the loss of inclusion seen in Figure 1.4(a) above. It has the added benefit of reducing memory pollution, since it ensures that x_t+1 was referenced after the prior reference to x_t, increasing the chances that it will be referenced after the current reference to x_t. In addition, their algorithm adjusts the priority of x_t+1 whether or not x_t is fetched, preventing the anomaly seen in Figure 1.4(b). Their algorithm also allows prefetched blocks to "age" down the stack at a rate k times faster than referenced blocks, where k is some small constant. They refer to their class of prefetch policies as OBL/k policies, and also discuss variable-space counterparts, VOBL/K. The combination of factors decreases the miss ratio by 10-30% compared to LRU, with far fewer prefetches than OBL.

1.5. Extensions to Stack Analysis

There have been several important extensions to the stack analysis technique. Mattson, et. al. [Matt70] showed how the hit ratio can be computed for an arbitrary number of levels, assuming a common block size and replacement policy. Gecsei [Gecs74] showed how it could be generalized to multiple levels with different block sizes for LRU and certain related policies. Traiger and Slutz

[Trai73] showed that it is possible to compute miss ratios for variable block sizes, and variable associativity, in a single pass. See also [Shed76] and [Slut72].

Coffman and Randell [Coff71] investigated the "extension problem", that is, to predict the performance of memory sizes greater than C, given only the misses from memory size C instead of a full trace. For LRU, a trace of "pushes" and "pulls" was sufficient; for other stack algorithms, the priority ranking for the block pushed and all blocks not in the memory of size C was also required. A trace of misses only was shown to be sufficient to provide good approximations to the performance of larger memories in [Smit77].

A more recent extension by Silberman [Silb83] showed that stack analysis can be applied to a "delayed-staging hierarchy" in which the processor directly accesses several levels of the memory hierarchy. When a referenced block is not in a higher level cache, it is supplied to the processor (at the speed of the highest level cache to contain the block) and begins "migrating" into the higher caches. The time elapsed until it becomes "staged" (resident) in a higher cache is equal to the sum of the access times of the caches below it. Further, the displacement of a block in the higher level cache is also delayed, creating a situation where the stack level of a block may be a function of the size of several lower level caches, and the time since last reference of one or more *other* blocks. Silberman showed that stack analysis can be applied to this class of hierarchy by maintaining the time and cache depth of last "migration" for each block. This information is used at the time of each reference to compute the stack distance of the block for different sizes of each level, considering the delayed staging times. This idea of maintaining additional information about each block will be seen again in our write-back algorithm.

2. Write-Back Stack Algorithm

We turn now to the development of a stack analysis algorithm for write-back. We begin by discussing the problems with write-back stack analysis, then present a general non-stack algorithm for computing the write-back ratios. We then prove that the algorithm obeys a form of inclusion, and derive a corresponding stack algorithm.

2.1. The Write-Back Problem

In write-back, a write access to the secondary storage occurs whenever a dirty block is "pushed". The main problem with write-back is maintaining the "state" (clean or dirty) of each block in the stack. A single dirty bit is sufficient in the real cache, however it clearly is not for the simulation stack. Consider a read to a dirty block at level k. For sizes k and larger the block is still dirty, since it has not been written; for sizes 1 to k it is clean. The inclusion property is

violated since the contents of the larger cache is "different" in the sense that the block has different attributes in some larger sizes. A second problem is accounting for the "dirty pushes". Each miss from a memory of size C causes a push from each smaller memory; that pushed block may be dirty. On first inspection, this suggests that counts need to be maintained and updated for every memory size from which a dirty block is pushed. We will show that a surprisingly simple technique solves both of these problems.

2.2. A Non-stack Algorithm

We begin by assuming that write-back is not a stack algorithm, and imagining a general algorithm for computing write-back miss or transfer ratios. The algorithm is based on the stack analysis algorithm from Section 1.2, but maintains a separate set of dirty blocks for each cache size in order to solve the problem of the non-inclusion of dirty bits. In addition to the symbols defined in Section 1.3, let:

w = $w_1, w_2 \ldots w_N$ where $w_t = \begin{cases} x_t & \text{if } x_t \text{ is a write} \\ \phi & \text{otherwise} \end{cases}$.

$D_t(C)$ = the set of dirty blocks in a memory of size C.
= $\{x : x \text{ is dirty in memory of size } C\}$.

$p_t(C)$ = the dirty block pushed from a memory of size C
= $\begin{cases} y_t(C) & \text{if } y_t(C) \in D_{t-1}(C) \\ \phi & \text{otherwise} \end{cases}$

$dp_t(C)$= the number of blocks written back from a memory of size C.

When a block is written it must be added to each dirty set. A block is removed from a set if and only if a dirty block is pushed from memory. We define Algorithm 2 by adding steps 7A and 8A to Algorithm 1. Note that if write fetch is not used then line 5 of Algorithm 2 must be conditioned on a read, i.e. IF $w_t = \phi$ THEN $rh_t(\Delta) = rh_{t-1}(\Delta) + 1$.

ALGORITHM 2. GENERAL NON-STACK WRITE-BACK ALGORITHM

```
1.   FOR 1≤t≤N                                          For all events
2.     IF x_t ∉ S_{t-1} THEN Δ=∞                        If not referenced before.
3.     ELSE
4.       FIND Δ SUCH THAT s_{t-1}(Δ)=x_t                Find the stack distance
5.       rh_t(Δ)=rh_{t-1}(Δ)+1                          Update the read hits
6.     IF Δ≠1                                           If stack needs updating
7.       y_t(1)=s_{t-1}(1)                              Calculate push set.
         FOR 2≤i<Δ DO y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]
         FOR i≥Δ DO y_t(i)=φ
7A.      FOR 2≤i≤Δ DO                                   Calculate dirty push set
           IF y_t(i)∈D_{t-1}(i) THEN                    If block is dirty
             p_t(i)=y_t(i)                              Include in dirty push set
             dp_t(i)=dp_{t-1}(i)+1                      Count dirty pushes
           ELSE p_t(i)=φ
8.       s_t(1)=x_t                                     Establish new stack.
         FOR i>1 DO s_t(i)=s_{t-1}(i) + y_t(i-1) - y_t(i)
8A.      FOR i≥1 DO D_t(i)=D_{t-1}(i) + w_t - p_t(i)    Establish new dirty set.
```

2.3. Dirty Set Inclusion Property

The inclusion property of stack algorithms states that if a block is present in memory of size C then it is present in size $C+1$, and therefore in all larger sizes. This can be formally stated as $M_t(C) \subseteq M_t(C+1)$, for all t and C. We now show that a similar condition applies to dirty sets, that is, if a block is dirty in a memory of size C then it is dirty in all larger sizes.

PROPOSITION 2.1: $D_t(C) \subseteq D_t(C+1)$, for all t and C.

PROOF: Choose an arbitrary C. The condition certainly applies at the start of the simulation. Assume it to be true at time $t-1$. We will operate on the sets in ways which preserve the subset relation and show that it holds at time t.

$$D_{t-1}(C) \subseteq D_{t-1}(C+1)$$

Adding the possibly null block w_t to both sets does not affect the subset relation.

$$D_{t-1}(C)+w_t \subseteq D_{t-1}(C+1)+w_t$$

Similarly, the relation holds if the block p_t is removed from the smaller set.

$$D_{t-1}(C)+w_t-p_t(C) \subseteq D_{t-1}(C+1)+w_t$$

Finally, removing the same block from both sets preserves the subset relation.

$$D_{t-1}(C)+w_t-p_t(C)-p_t(C+1) \subseteq D_{t-1}(C+1)+w_t-p_t(C+1)$$

Note that the right-hand side is exactly $D_t(C+1)$ as computed by line 8A, while the left-hand side differs from $D_t(C)$ only by the term $p_t(C+1)$. There are three

possible values for $p_t(C+1)$ none of which affect the set on the left-hand side:

a) if $y_t(C+1)$ is not dirty then $p_t(C+1)=\phi$;

b) if $y_t(C+1)=y_t(C)$ then $p_t(C+1)=p_t(C)$;

c) if $y_t(C+1)\neq y_t(C)$ then $p_t(C+1)\neq p_t(C)$. However, it must be true that $y_t(C+1)=s_{t-1}(C+1)$, that is, the block pushed from size $C+1$ was the block at level $C+1$. But $s_{t-1}(C+1)\notin M_{t-1}(C)$, and therefore $p_t(C+1)\notin D_{t-1}(C)$, so again it has no effect.

Removing this term gives

$$D_{t-1}(C)+w_t-p_t(C)\subseteq D_{t-1}(C+1)+w_t-p_t(C+1)$$

which is exactly equal to

$$D_t(C)\subseteq D_t(C+1)$$

With these facts we can simplify the algorithm considerably. First, $D_t(C)\subseteq D_t(C+1)$ implies that there is a minimum size at which a block is dirty (if it is dirty at all). Intuitively, this is the smallest memory from which the block has not been pushed since its last write reference, and therefore the smallest memory size in which it is still dirty. This is also the largest stack distance the block has attained since it was last written. Therefore the separate $D_t(C)$ can be replaced by a single array. Let $dl(x)$ be the *dirty level* of block x; infinity if the block has never been written. A block at level k (i.e. $s(k)=x$) is dirty if and only if $dl(x)\leq k$. We can set the dirty level to 1 when a block is written and update it as the block is pushed.

2.4. Writes Avoided

Before defining the new algorithm, let us also reconsider the way dirty pushes are counted. In Algorithm 2, dp is updated as each block is pushed. Also, recall that the purpose of write-back is to avoid the physical write to secondary storage for each write reference which is required when using write through. We can count the number of write-backs required in two ways. One is to count them directly. The other is to count the total number of writes, and then to subtract the number of times that no *additional* write-back is required, since the block was already dirty or is being deleted. When a write does not require a write back, we increment the count of *writes avoided*. This is analogous to the way reads are computed in the basic stack analysis algorithm, where a read is avoided for all sizes larger that the current stack distance.

Ignoring deletes for now, a write is avoided only when a dirty block is overwritten, since both the previous and current modification can be written by

the next physical write (copy-back). Therefore we can say that the previous write has been avoided for all sizes equal to or greater than the current dirty level. Notice that we now only care about the dirty level for the block being referenced and therefore we only need to adjust dl for the referenced block. If it is found at level Δ which is below its dirty level (i.e. $\Delta > dl(x_t)$), we can reason that the block has been pushed (while dirty) from all levels between $dl(x_t)$ and Δ, therefore the proper value for $dl(x_t)$ is Δ.

We now define $wa(C)$ to be the *writes avoided* at level C, that is, the number of writes for which the referenced block was still dirty in memory sizes C and larger. The combined algorithm is shown below.

ALGORITHM 3: WRITE-BACK STACK ALGORITHM

1.	FOR $1 \le t \le N$ DO	*For all events*
2.	IF $x_t \notin S_{t-1}$ THEN $\Delta = \infty$	*If not referenced before.*
3.	ELSE	
4.	FIND Δ SUCH THAT $s_{t-1}(\Delta) = x_t$	*Find the stack distance*
5.	$rh_t(\Delta) = rh_{t-1}(\Delta) + 1$	*Update the read hits*
6.	IF $dl(x_t) < \Delta$ THEN $dl(x_t) = \Delta$	*Set the "real" dirty level.*
7.	IF $\Delta \neq 1$	*If stack needs updating*
8.	$y_t(1) = s_{t-1}(1)$	*Calculate push set.*
	FOR $2 \le i < \Delta$ DO $y_t(i) = pmin[y_t(i-1), s_{t-1}(i)]$	
	FOR $i \ge \Delta$ DO $y_t(i) = \phi$	
9.	$s_t(1) = x_t$	*Establish new stack.*
	FOR $i > 1$ DO $s_t(i) = s_{t-1}(i) + y_t(i-1) - y_t(i)$	
10.	IF $w_t \neq \phi$ THEN	*Skip if a read.*
11.	IF $dl(x_t) \neq \infty$ THEN	
	$wa_t(dl(x_t)) = wa_{t-1}(dl(x_t)) + 1$	*Count writes avoided.*
12.	$dl(x_t) = 1$	*Block is dirty.*
13.	$W_t = W_{t-1} + 1$	*Count of write references.*

For the special case of LRU, this algorithm is particularly simple. As in the standard stack analysis algorithm for LRU, updating the stack is a matter of removing the referenced block and inserting it at the top of the stack. The fact that only the referenced block affects the statistics is particularly useful for this case, since no work needs to be done while searching for the referenced block.

2.5. Dirty Push Computation

Using Algorithm 3, the number of dirty pushes which have occurred by time t for a memory of size C is given by

$$dp_t(C)=W_t-\sum_{i=1}^{C} wa_t(i) - |D_t(C)| \tag{2.1}$$

where the count of write references by time t is

$$W_t = \sum_{i=1}^{t} (1: w_i=x_i)$$

and the count of dirty blocks resident in the cache of size C is the size of the set

$$D_t(C) = \{x : x=s_t(\Delta), \Delta \leq C, dl(x) \leq C\}$$

The first two terms of (2.1) are obvious, but we should elaborate on the need for the third term. It should be clear that each block which is still dirty has avoided the most recent write for all sizes in which it is still dirty and should therefore be subtracted from the count of writes. This argument applies at any point during the trace, and at the end of the simulation. Since the relevant metrics are those gathered during the trace period, regardless of any activity which occurs after the trace ends, we should consider each remaining dirty block as having avoided a write. To simplify the computations, we make a final scan of the memory stack and update $wa(dl(x))$ for each dirty block x. Of course, the effect of this should be small if the total number of trace events is large.

Using this expression for the number of dirty pushes leads to a simple recurrence for computing the traffic ratio. Recall that equation (1.7) for computing the transfer ratio from Section 1.2 is

$$T(C) = [m_r(C)+m_w(C)*Wf + dp(C)]/N$$

Assuming write-fetch, the first two terms can be replaced by the stack analysis computation of the miss ratio given by (1.8), giving

$$T(C) = [[N - \sum_{i=1}^{C} rh(i)] + dp(C)]/N$$

Substituting (2.1) for $dp(C)$, assuming that the final scan has updated wa, this simplifies to

$$T(C) = [N - \sum_{i=1}^{C} rh(i)] + [W_t - \sum_{i=1}^{C} wa(i)]/N$$

$$T(C) = [(N+W_t) - \sum_{i=1}^{C} (rh(i)+wa(i))]/N \tag{2.2}$$

or

$$T(0) = (N+W_t)/N \tag{2.3}$$

$$T(C) = T(C-1)-[(rh(C)+wa(C))/N]$$

Notice that since $rh(i)$ and $wa(i)$ are both non-negative, this function also decreases as memory size increases, just as the miss ratio does.

2.6. Warm Start

If the simulation results are gathered starting from an empty stack, the results can be biased by the fact that many of the early references will be misses in all cache sizes. In fact, until the memory contains k blocks there is no chance of a hit at level k, producing a higher than expected miss ratio. In some situations this *cold-start* miss ratio is appropriate, for example when a single-program address trace is used to derive multi-programming metrics [East78]. In other situations, the desired metrics are those for a system in steady-state. In these cases it is common to *warm start* the simulation to reduce startup effects. A warm start consists of allowing the simulation to run until it is assumed to be in steady-state, often either for a fixed number of events or until the memory contains a fixed number of blocks, then stopping. Without changing the state of the simulation, all statistics are cleared. The simulation then resumes from its current state. The final metrics are those gathered after the warm start.

Warm start using the write-back algorithm can produce an anomaly in the transfer ratio. This is caused by the final scan of memory which considers all dirty blocks as having avoided a write — a write which may have occurred before the warm start. Suppose, for example, that the write-back simulation is warm started, and suppose that W_t and wa are zeroed. Then immediately after warm start, the value of $dp(C)$ calculated using (2.1) may be negative for some values of C, as shown in Figure 2.2, where the number in parentheses is the dirty level of the block. Of course, a "negative push" is meaningless. We can keep the numbers positive by setting W_t to the number of dirty blocks in the cache at warm start, but then dp is immediately non-zero for some cache sizes. Another alternative would be to zero both wa and dl, but then it will be a long time before any dirty block could be pushed from large sizes — in conflict with the reason to warm start in the first place.

Level	Stack	*wa*	*dp*
1	A(1)	0	-1
2	B(4)	0	-1
3	C(∞)	0	-1
4	D(4)	0	-3
5	E(5)	0	-4

Figure 2.2 Count of dirty pushes after zeroing W_t and *wa*.

Since the third term of (2.1) increases with C, the second term of (2.1), the sum of *wa*, must decrease for larger C if we want the computed value of *dp* to be zero immediately after warm start. This can only happen if some *wa* are negative. The solution we use is to zero *wa* at warm start, then *decrement* $wa(dl(x))$ for all dirty blocks x. With this solution $dp(C)$ is zero immediately after warm start for all C, as it intuitively should be. See Figure 2.3a. Now suppose that a total miss causes all blocks to be pushed (Figure 2.3b). The result is that $dp(C)$ is zero except for those sizes from which a dirty block is pushed — consistent with the result from a simulation of a single cache size, or a real cache

Level	Stack	*wa*	*dp*
1	A(1)	-1	0
2	B(4)	0	0
3	C(∞)	0	0
4	D(4)	-2	0
5	E(5)	-1	0

(a)

Level	Stack	*wa*	*dp*
1	F(∞)	-1	1
2	A(1)	0	0
3	B(4)	0	0
4	C(∞)	-2	1
5	D(4)	-1	1
6	E(5)	0	0

(b)

Figure 2.3 Count of dirty pushes (a) after warm start with W_t=4 and, (b) after all blocks are pushed one level.

Note, however, the unexpected result that the transfer ratio due to dirty pushes is no longer a monotone decreasing function of size. In fact, if the warm start of Figure 2.3(a) were followed by the unlikely event of five total misses, the resulting transfer ratio would be *increasing* with cache size. It seems that the rate of dirty pushes may be exaggerated for larger cache sizes by the fact that there are more dirty blocks in the larger cache. (There may also be a higher probability that blocks pushed from larger caches are dirty. See Section 5.2) This "error" for large sizes is bounded by the number of dirty blocks in the stack divided by the number of references after warm start. It can therefore be made arbitrarily small by increasing the number of references after warm start. In most

cases, locality will cause the write-back traffic ratio to assume its normal decreasing form.

3. Extensions

In addition to write-back, several intermediate and related policies can be analyzed using our technique.

3.1. Write Through

This policy is trivially included in the algorithm by setting $dl(x_t)$ to infinity instead of one after a write. If fact, since the total number of write requests is known, both the write-back and write through transfer and traffic ratios are available simultaneously. It is also possible to simulate a combination of policies, provided the choice of policy is not a function of memory size. For example, some blocks could be write through and others write-back, a scheme used in some real caches, for example the Fairchild CLIPPER processor [Cho86] and the NEC disk cache [Toku80]. An example of an algorithm for such a cache is given as Algorithm 4 below.

ALGORITHM 4: MIXED WRITE-BACK/WRITE-THROUGH STACK ALGORITHM

1.	FOR $1 \le t \le N$ DO	*For all events*
2.	IF $x_t \notin S_{t-1}$ THEN $\Delta=\infty$	*If not referenced before.*
3.	ELSE	
4.	FIND Δ SUCH THAT $s_{t-1}(\Delta)=x_t$	*Find the stack distance*
5.	IF $w_t=\phi$	*If this is a read*
6.	$rh_t(\Delta)=rh_{t-1}(\Delta)+1$	*Update the read hits*
7.	IF $dl(x_t)<\Delta$ THEN $dl(x_t)=\Delta$	*Set the "real" dirty level.*
8.	IF $\Delta \neq 1$	*If stack needs updating*
9.	$y_t(1)=s_{t-1}(1)$	*Calculate push set.*
	FOR $2 \le i < \Delta$ DO $y_t(i)=pmin[y_t(i-1), s_{t-1}(i)]$	
	FOR $i \ge \Delta$ DO $y_t(i)=\phi$	
10.	$s_t(1)=x_t$	*Establish new stack.*
	FOR $i \ge 1$ DO $s_t(i)=s_{t-1}(i) + y_t(i-1) - y_t(i)$	
11.	IF $w_t \neq \phi$ AND BLOCK IS WRITE-BACK THEN	*If write-back.*
12.	IF $dl(x_t) \neq \infty$ THEN	*Update dirty pushes.*
	$wa_t(dl(x_t))=wa_{t-1}(dl(x_t))+1$	*Count writes avoided.*
13.	$dl(x_t)=1$	*Block is dirty.*
14.	$W_t=W_{t-1}+1$	
15.	ELSE $dl(x_t)=\infty$	*Write-through or read*

3.2. Periodic Write-Back

With large caches, there may be a very long delay before a block is removed by replacement. We have mentioned that reliability considerations may dictate that a dirty block be written before this time. Suppose that all dirty blocks are written every n seconds instead. An example of this is the UNIX file system policy of writing all dirty file system buffers to disk every 30 seconds. Alternatively, suppose only certain blocks are written, for example by a policy to write a block after it has been unreferenced for n seconds. These policies are all stack algorithms, provided that the write happens for all memory sizes where the block is dirty, in order to maintain inclusion in the dirty set.

A forced write-back is implemented in the algorithm by setting $dl(x)$ to infinity for each written block. It has no effect on writes avoided, except that the write which made the block dirty can not subsequently be avoided. The effect of this is to increase the calculated number of dirty pushes. Consider the third term in (2.1) for any C where the block is dirty: the block was dirty and included in $D_t(C)$; it is now clean and not in the term; the net increase to $dp_t(C)$ is 1.

3.3. Deletions

A important consideration in file system studies is the existence of deletions in the reference string. If a file is deleted, the blocks of that file should be removed from the cache without write. With a write-back cache and short file lifetimes, it is likely that file blocks will be created and deleted without ever being written to the next level [Oust85]. Deletions also occur in processor caches when blocks are invalidated, but generally not without writing the block first if it is dirty. This case is discussed is Section 3.4.

Deletion of blocks from the cache was discussed by Mattson et al. [Matt70] in the context of a "call back" hierarchy, where cache blocks may be invalidated by a write directed to a lower level. The example used by Mattson is a virtual memory system in which all I/O occurs to blocks residing in an "I/O Subsystem", not the CPU memory. If an I/O is addressed to a block which is in CPU memory, that block must be invalidated. Greenburg [Gree74] also discusses deletions, and implemented an algorithm to approximate the effect of deletion. Olken [Olke81] proposes an exact algorithm, and discusses implementation using various data structures. None of these consider the effect of write back.

If a deleted block were simply deleted from the stack, the stack level for all lower blocks would be reduced. This would have the undesirable effect of calling these blocks back into a memory from which they had been pushed. Instead, what Mattson called a "marker" block is inserted in the stack replacing the deleted block.

We refer to the marker blocks as "gaps" in the stack, corresponding to a vacant block in all larger caches. The next push from above the gap will fill it with the pushed block. Thus a gap will stop the sequence of pushes, just as finding the referenced block stops the pushes in the normal case. However, since the referenced block must still be pulled to the top, it may have to be replaced in the stack by another gap. Thus, a reference to a block below the first gap will seem to "migrate" the gap down the stack. As an example, consider the sequence of Figure 3.3. After block **D** is deleted, a gap is left at level 4. References above level 4 will not affect the gap. However, a reference below level 4 will "migrate" the gap to a lower level. From the point of view of the "real" cache, the gap represents the same vacant block, which was in all memory sizes 4 or larger. Since block **F** is already resident in memories of size 6 or larger, the reference to **F** has not fetched any block to fill the gap. Therefore the gap still exists in these sizes.

Stack Level	Initial Stack	Delete D	Reference B (above gap)	Reference F (below gap)
1	A	A	B	F
2	B	B	A	B
3	C	C	C	A
4	D	γ	γ	C
5	E	E	E	E
6	F	F	F	γ

Figure 3.3 The migration of gaps in the stack.

The effect of deletions on the transfer ratio is to introduce another way in which a write can be avoided, particularly evident in large cache sizes. If a block is written then deleted before it is pushed, the copy-back is avoided. The write is avoided for all sizes greater than the current dirty level, thus satisfying the inclusion property. It is therefore a simple matter to increment the appropriate $wa(dl(x_t))$ on deletion. In addition, the count of read hits must exclude deletes, since a deleted block is never fetched. This is seen in lines 6 and 7 below.

The complete, though somewhat complicated, algorithm for write-back with deletions is given as Algorithm 5 below. Let:

γ = a gap marker in the stack.

Γ = the level of the first gap in the stack.

Δ' = $\min(\Delta,\Gamma)$, the level at which pushes stop.

ALGORITHM 5: WRITE-BACK STACK ALGORITHM WITH DELETES

```
1.  FOR 1≤t≤N DO                                   For all events
2.     IF x_t ∉ S_{t-1} THEN Δ=∞                   If not referenced before.
3.     ELSE
4.        FIND Δ SUCH THAT s_{t-1}(Δ)=x_t          Find the stack distance
5.        IF dl(x_t)<Δ THEN dl(x_t)=Δ              Set the "real" dirty level.
6.        IF x_t IS A DELETE THEN
             wa_t(dl(x_t))=wa_{t-1}(dl(x_t))+1     Count writes avoided.
             s_t(Δ)=γ                              Store a gap in the stack.
             BREAK                                 Process next reference
7.        ELSE rh_t(Δ)=rh_{t-1}(Δ)+1               Update the read hits
8.     Γ=min(i : s_{t-1}(i)=γ)                     Level of the first gap.
9.     Δ'=min(Δ,Γ)                                 Level where pushes stop.
10.    IF Δ'≠1                                     If stack needs updating
11.       y_t(1)=s_{t-1}(1)                        Calculate push set.
          FOR 2≤i<Δ' DO y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]
          FOR i≥Δ' DO y_t(i)=φ
12.       FOR i>1 DO s_t(i)=s_{t-1}(i) + y_t(i-1) - y_t(i)   Establish new stack.
13.    s_t(1)=x_t                                  Pull reference to top.
14.    IF Δ'=Γ THEN s_t(Δ)=γ                       Migrate the gap
15.    IF w_t≠φ THEN                               Skip if a read.
16.       IF dl(x_t)≠∞ THEN
             wa_t(dl(x_t))=wa_{t-1}(dl(x_t))+1     Count writes avoided.
17.       dl(x_t)=1                                Block is dirty.
```

3.4. Flush Back

In some situations a block should be written and removed from the cache before it is a candidate for replacement. An example is the wholesale flush of a local processor cache in a multiprocessor system. A more selective example is where an individual block is flushed from a private cache on a multi-processor bus so that another processor can acquire the block [Katz85]. Flushing the cache periodically is also used in some processor simulations to approximate multiprogramming effects [Smit82]. It should be clear that this can be simply implemented as a periodic write-back followed by a delete. The contents of wa is unchanged.

4. Sector Cache Simulation

We now consider the study of sub-block or sector caches, and show that they too can be simulated using stack analysis by a technique similar to that used for write-back.

4.1. Background

A typical cache consists of blocks or sectors of data, each with associated tags identifying the virtual addresses contained in the block. See Figure 4.1(a). If smaller blocks are used the total space for tags increases (Figure 4.1(b)), since each of the blocks requires its own tags. Regardless of size, each block also requires a valid bit indicating whether the block contains valid data.

An alternative arrangement is the sub-block or sector cache. In a sector cache each cache block/sector is divided into a fixed number of sub-blocks/sub-sectors. (Throughout this section we will use IEEE-proposed terminology for such caches, referring to *sectors* and *sub-sectors*.) Tags are associated with the sector as a whole. See Figure 4.1(c). Transfers between the cache and secondary storage are done in units of sub-sectors. In addition, there must be a valid bit for each sub-sector to indicate whether or not the sub-sector data is present.

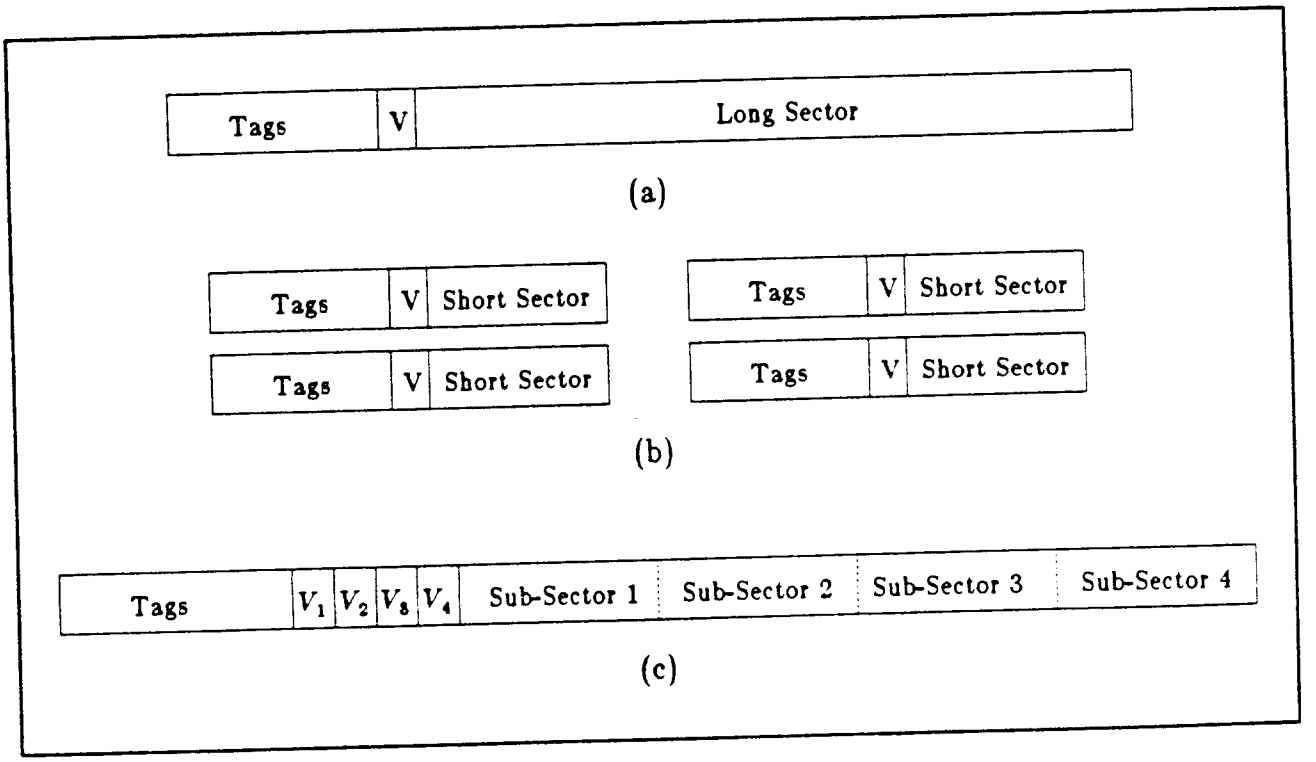


Figure 4.1 Alternative layouts for a block/sector in a cache.

Sector caches are motivated by two factors. The first is a need to reduce the number of tags to be searched. This was the motivation when it was first used in the IBM 360/85 cache [Lipt68]. The reduced number of tags also reduces the chip area needed for tags in a VLSI cache. A second reason for sub-sectors is to reduce the size of each data transfer. On a cache chip with limited pins for parallel data transfer, a large sector size would require multiple cycles, where a smaller sub-sector could be transferred in one parallel access. Similarly, on-chip data path

widths favor a small sector. The smaller sub-sectors may also be used to reduce memory bus traffic when the bus is a potential constraint [Good83].

A sector cache tends to have a higher miss ratio than the same size cache with sub-sector-sized blocks because the of the rigidity in the assignment of sub-sectors. It may also have a higher miss ratio than the same size cache using sector-sized blocks because each sub-sector can cause a fault. However, misses that fetch smaller sectors may "cost" less than larger sectors, is some cases. At the same time, the sector cache reduces the traffic ratio compared to the non-sector cache with large blocks by not loading sub-sectors which are not needed, as would be the case if the entire sector were loaded. The performance of sub-sector processor cache is studied by Hill and Smith [Hill84].

The disk cache in the IBM 3880 Control Unit is also a form of sector cache [Gros85]. The sector size is a full track, with a variable number of sub-sectors — one for each disk record. This organization was chosen so that the cache could be a physical and logical copy of the disk contents, while offering the advantages of caching. To avoid holding up the processor waiting for a full track to be transferred, the disk is positioned to the requested record which is then transferred to both the processor and controller cache. After signaling completion of the requested I/O, the controller continues to read to the end of the track into cache, anticipating further sequential requests. This form of prefetch is called *load forward*, and is discussed in Section 4.3.2.

4.2. The Stack Simulation Problem

The problem with stack simulation of a sector cache is that the valid bits do not obey inclusion. For example, suppose sub-sector 1 of a sector is referenced and becomes valid. Now suppose that the sector is pushed to level k in the stack, then sub-sector 2 is referenced. The entire sector must be pulled to the top of the stack in order for sub-sector 2 to become valid in all cache sizes, but sub-sector 1 is valid for some sizes (k and larger), and invalid for others.

Our solution is to replace the valid bit with a *valid level* for each sub-sector. The valid level is the minimum memory size for which the sub-sector is still valid; infinity if the sub-sector has never been referenced. As in the case of write-back, the valid levels only need to be updated when the sector is referenced. A reference to any sub-sector will pull the entire sector to the top of the stack. Since valid levels are only updated for the sector being referenced, it is possible for valid levels to be less than the stack level of the sector as a whole. If a sub-sector has a valid level less than the current stack level, the valid level is set to the current level, since the sector as a whole is invalid in smaller cache sizes. The formal algorithm is similar to the one for write-back, and is presented below. The terms are somewhat different from those used previously.

Let:

X = $(x_1,y_1),(x_2,y_2)...(x_N,y_N)$ be a series of references, where (x_t,y_t) is a reference to sector x, sub-sector y at time t.

$(x,*)$ = any sub-sector of sector x.

B = number of sub-sectors per sector.

vl_t = the valid level of sub-sector (x,y).

ALGORITHM 6: SUB-SECTOR STACK ALGORITHM

1.	FOR $1 \le t \le N$ DO	*For all events*
2.	IF $(x_t,*) \notin S_{t-1}$ THEN $\Delta=\infty$	*If sector not in stack*
3.	ELSE	
4.	FIND Δ SUCH THAT $s_{t-1}(\Delta)=(x_t,*)$	*Find the stack distance*
5.	FOR $1 \le j \le B$ DO	
	$vl_t(x_t,j)=max(vl_{t-1}(x_t,j),\Delta)$	*Fix valid levels*
6.	$\Delta_y=vl_t(x_t,y_t)$	*Stack distance for (x,y)*
7.	$rh_t(\Delta_y)=rh_{t-1}(\Delta_y)+1$	*Update the read hits*
8.	IF $\Delta \neq 1$	*If the stack needs updating*
9.	$y_t(1)=s_{t-1}(1)$	*Calculate push set.*
	FOR $2 \le i < \Delta$ DO $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$	
	FOR $i \ge \Delta$ DO $y_t(i)=\phi$	
10.	$s_t(1)=(x_t,y_t)$	*Establish new stack.*
	FOR $i>1$ DO $s_t(i)=s_{t-1}(i) + y_t(i-1) - y_t(i)$	
11.	$vl_t(x,y)=1$	*(x,y) valid in all sizes.*

Notice that the miss and transfer ratios are based on the valid level of the referenced sub-sector — not the stack level of the sector as a whole. For example, in Figure 4.2, a reference to sub-sector **A1** is a hit at level 4, since the sub-sector is not present in sizes smaller than 4. On the other hand, a reference to sub-sector **B2** is a hit at level 2, since the sector as a whole is absent from size 1.

Stack Level	Sector	Sub-sector 1	2	3
1	A	4	1	∞
2	B	2	1	∞
3	C	1	1	1
4	D	1	∞	∞

Figure 4.2 Valid levels in a sector cache.

4.3. Extensions

4.3.1. Write Back

The first obvious extension is to consider write back with a sector cache. Since these are independent they can be combined by maintaining a dirty level in addition to the valid level. The dirty level could be associated with the entire sector, however since part of the motivation for the sub-sector cache is to reduce bus traffic, it is more logically associated with each sub-sector. The algorithm is similar to those already presented.

4.3.2. Load Forward

Load forward is a form of prefetch associated with sector caches [Hill84]. After loading a requested sub-sector, successive sub-sectors are loaded until the end of the sector. As with any prefetch, this reduces the miss ratio because of the strong probability of sequential references. However, unlike normal prefetch, there is no chance that load forward can cause memory pollution [Smit78b] by displacing a soon-to-be-referenced sector.

Under certain conditions, load forward is a stack algorithm. We will present a formal algorithm assuming it is not a stack algorithm, then develop the conditions under which it is a stack algorithm. Again, imagine a non-stack algorithm which keeps a separate memory set $M_t(C)$ for each size C. For simplicity we ignore writes. In addition to symbols previously defined, let:

$(x,y)^+ =$ (x,y) plus the set of all sectors/sub-sectors prefetched with sub-sector (x,y). For the moment we will not restrict it to load forward.

$M_t(C) =$ the set of valid sub-sectors in memory of size C.
$= \{(x,y):(x,y)$ in memory of size C at time $t\}$

$s_t(C) =$ the set of valid sub-sectors of the sector at stack level C. Note that there can be sub-sectors which are valid at larger levels but not at C.
$= \{(x,y):(x,y) \in M_t(C)\ ,\ (x,i) \notin M_t(j)\ ,\ 1 \le i \le B\ ,$ for all $j<C\}$

ALGORITHM 7: GENERAL NON-STACK LOAD-FORWARD ALGORITHM

```
1.FOR 1≤t≤N DO                                          For all events
2.   IF (x_t,*)∉S_{t-1} THEN Δ=∞                        If sector not in stack
3.   ELSE
4.      FIND Δ SUCH THAT (x_t,*)∈s_{t-1}(Δ)             Find the sector distance
5.      FIND Δ_y=min(C:(x,y)∈M_t(C))                    Find the sub-sector distance
6.      rh_t(Δ_y)=rh_{t-1}(Δ_y)+1                       Update the read hits
7.   IF Δ≠1                                             If the stack needs updating
8.      y_t(1)=s_{t-1}(1)                               Calculate push set.
        FOR 2≤i<Δ DO y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]
        FOR i≥Δ DO y_t(i)=φ
9.   FOR 1≤i<Δ_y DO M_t(i)=M_{t-1}(i) + (x,y)^+ - y_t(i)   Establish new memory.
```

We want to show inclusion, that is, $M_t(C) \subseteq M_t(C+1)$ for all t and C. We can immediately think of a situation where this will be violated. For example, suppose (x,y) prefetches (x,z), and these sub-sectors are valid at the levels shown in Figure 4.3(a). Now let (x,y) be referenced. For all sizes less than k, (x,y) is fetched, prefetching (x,z). For sizes greater than k, neither sub-sector is fetched. The valid levels, which are shown in Figure 4.3(b), violate inclusion since (x,z) is not present for sizes between k and l.

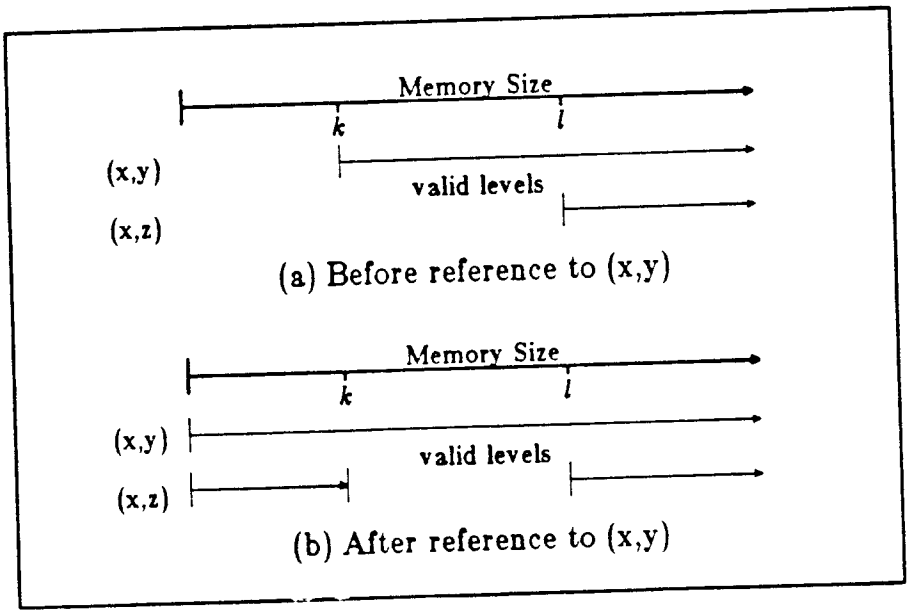


Figure 4.3 A situation where load-forward can violate inclusion

However, suppose the initial configuration is reversed, as shown in Figure 4.4(a). The result of a reference to (x,y) is that (x,z) is prefetched for all sizes less that l (although it only needs to be accessed from secondary storage for sizes less than k), resulting in both sub-sectors becoming valid in all sizes. See Figure

4.4(b). Therefore a necessary condition for inclusion is that the first configuration can never occur.

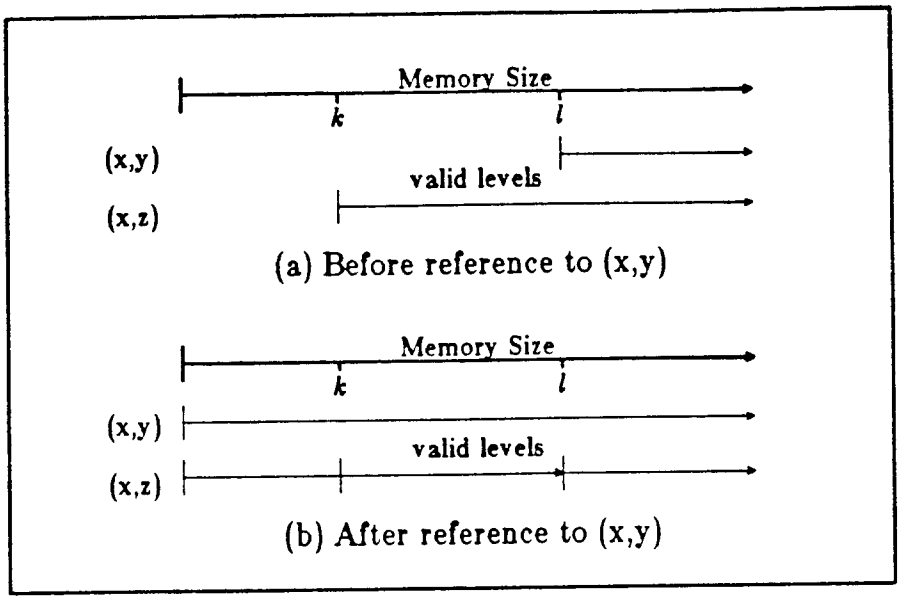


Figure 4.4 A situation where load-forward preserves inclusion.

Stated formally, this leads to the following proposition.

PROPOSITION 4.1: Using Algorithm 7, if $(x,z) \in (x,y)^+$ and $(x,y) \in M_t(C)$, then $(x,z) \in M_t(C)$ and inclusion holds.

PROOF: Choose an arbitrary size C. It is certainly true at the start when the cache is empty. Assume the induction hypothesis that the condition holds at time t-1. We will show that it holds after the reference at time t. Consider the possible configurations of (x,y) and (x,z) which could lead to (x,y) present at time t.

Case 1:

Neither (x,y) nor (x,z) present at time t-1, and (x,y) is referenced. Both sub-sectors are fetched, and the condition holds.

Case 2:

Sub-Sector (x,z) present, but not (x,y), and (x,y) referenced. Both are again fetched, and the condition holds.

Case 3:

Both present, and (x,y) referenced. As shown earlier, both become valid in all sizes, and the condition holds.

Case 4:

Both present, and (x,z) referenced. Sub-sector (x,z) will become valid in all sizes. Although (x,y) is unchanged, the condition still holds.

Case 5:

Both present and another sector referenced. If sector x is not pushed from size C, then the condition still holds. If the sector is pushed from size C, then neither (x,y) not (x,z) will be present, and the condition holds.

Case 6:

A more subtle case, where both are present and sub-sector (x,w) is referenced. If neither (x,y) nor $(x,z)\in(x,w)^+$ then neither is affected, and the condition holds. If $(x,z)\in(x,w)^+$, or if both are, then the condition holds. However, if $(x,y)\in(x,w)^+$, but not (x,z), then the new configuration violates the condition. Similarly, if neither is present, or (x,z) alone is present, there is no problem unless $(x,y)\in(x,w)^+$ and (x,z) is not.

The proposition is therefore true if the prefetch sets obey the transitive condition that if $(x,z)\in(x,y)^+$ and $(x,y)\in(x,w)^+$ then $(x,z)\in(x,w)^+$. This condition is satisfied by load forward if it loads the entire rest of the sector (but not if it loads just the next sub-sector, say).

We can now show that Algorithm 7 satisfies inclusion.

PROPOSITION 4.2: If Algorithm 7 is used then $M_t(C)\subseteq M_t(C+1)$ for all t and C.

PROOF: It is certainly true at the start. Assume it is true at time $t-1$ for an arbitrary size C. Again we perform operations on each set which preserve the subset relation to show that it is true at time t.

$$M_{t-1}(C)\subseteq M_{t-1}(C+1)$$

$$M_{t-1}(C)-y_t(C)\subseteq M_{t-1}(C+1)$$

$$M_{t-1}(C)-y_t(C)-y_t(C+1)\subseteq M_{t-1}(C+1)-y_t(C+1)$$

By an argument similar to the one used to prove Proposition 2.1, $y_t(C+1)$ can have three possible values, none of which affect the subset relation:

a) $y_t(C+1)=\phi$

b) $y_t(C+1)=y_t(C)$

c) $y_t(C+1)\neq y_t(C)$, in which case $y_t(C+1)=s_{t-1}(C+1)$, which is not in $M_{t-1}(C)$. Therefore

$$M_{t-1}(C)-y_t(C)\subseteq M_{t-1}(C+1)-y_t(C+1)$$

$$M_{t-1}(C)+(x,y)^+-y_t(C)\subseteq M_{t-1}(C+1)+(x,y)^+-y_t(C+1)$$

For $C<\Delta_y-1$, these are exactly the computations from line 10. For $C\geq\Delta_y$, the proposition is true by the induction hypothesis, since the contents of memory are unchanged. For $C=\Delta_y-1$, the algorithm uses $(x,y)^+$ on the left-hand side, but not on the right. But $(x,y)^+\subseteq M_t(C+1)$ by the prior proposition, therefore the right-

hand side is $M_t(C+1)$, giving

$$M_t(C) \subseteq M_t(C+1)$$

as was to be shown.

Because of inclusion, we can convert Algorithm 6 to a load forward algorithm using valid levels, as follows.

ALGORITHM 8: STACK ALGORITHM FOR LOAD FORWARD

1.	FOR $1 \le t \le N$ DO	*For all events*
2.	IF $(x_t,*) \notin S_{t-1}$ THEN $\Delta=\infty$	*If sector not in stack*
3.	ELSE	
4.	FIND Δ SUCH THAT $s_{t-1}(\Delta)=(x_t,*)$	*Find the sector distance*
5.	FOR $1 \le j \le B$ DO $vl_t(x_t,j)=max(vl_{t-1}(x_t,j),\Delta)$	*Fix valid levels*
6.	$\Delta_y=vl_t(x_t,y_t)$	*Find sub-sector distance*
7.	$rh_t(\Delta_y)=rh_{t-1}(\Delta_y)+1$	*Update the read hits*
8.	IF $\Delta \neq 1$	*If the stack needs updating*
9.	$y_t(1)=s_{t-1}(1)$	*Calculate push set.*
	FOR $2 \le i < \Delta$ DO $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$	
	FOR $i \ge \Delta$ DO $y_t(i)=\phi$	
10.	$s_t(1)=(x,y)^+$	*Establish new stack.*
	FOR $i>1$ DO $s_t(i)=s_{t-1}(i) + y_t(i-1) - y_t(i)$	
11.	FOR $y \le i \le B$ DO $vl_t(x,i)=1$	$(x,y)^+$ *valid in all sizes.*

5. Experimental Results

5.1. Run-Time Comparison

As we stated earlier, the chief advantage of stack analysis is that it allows the desired metrics to be calculated for all cache sizes in a single pass of the trace data. Although the overhead of maintaining the memory stack usually makes stack analysis take longer than the simulation of a single cache size, it should take only a fraction of the time required to produce a reasonable curve using several single-size simulations. We have used our write-back stack algorithm in the analysis of a variety of trace data, and find that this time savings does not always occur. For example, file system traces typically exhibit much poorer locality than single program address traces. This results in excessive run times using the straight-forward implementation of the stack simulator. In a companion paper we present several techniques to reduce the execution time of stack analysis by using a tree-based representation of the stack [Thom87].

Table 5.1 shows the execution times for simulations using several traces of various types. The trace files are described in more detail in Section 5.2. All

simulations use LRU replacement. The first column shows the time required to compute a point on the the miss or transfer ratio curve using a simple simulation of a single cache size. The second column is the time for stack analysis to compute the entire curve using a naive implementation which searches the simulation stack from the top to find the stack distance of the referenced block. The third column shows the time required using the best stack implementation presented in [Thom87]. In most cases this implementation uses a hash table to locate the referenced block and a binary tree to determine the stack distance. All times are in seconds for simulations of approximately 500,000 events running on a VAX 11/750. We see that the stack algorithm takes on the average 22% more time for memory address traces and twice as long for the file system traces, as compared to the single-size simulation. However, it reduces the execution time by as much as 90% for the program address traces, and at least 80% for the file system traces, when compared to the time required to approximate the miss/transfer ratio curves using ten non-stack simulations.

Comparison of Execution Times (CPU seconds)			
Trace	Single Size Simulation	Simple Stack Simulation	Best Stack Simulation
Program Address Traces			
FGO1	351	487	429
FGO2	309	357	357
MVS	369	878	460
LISPCOMP	328	512	428
RISCR	225	248	225
SPICE	277	342	342
VAXIMA	359	737	475
UNIX File System Traces			
ERNIE	598	20,787	1,317
ARPA	612	19,416	1,246
CAD	622	31,375	1,111

Table 5.1 Comparison of execution times for stack analysis and single-size simulations, in CPU seconds.

5.2. Write-Back Probability

Before the discovery of our write back stack algorithm, an attempt was made to estimate write-back traffic in the following way. Each time a miss occurs (ignoring gaps) a block is pushed from all cache sizes. The write-back traffic should be approximately equal to the miss ratio times the probability that the block pushed from cache is dirty [Smit85a]. Smith estimated for program data addresses that half of all pushes are dirty, but with wide variations between programs. It was also reasoned that the probability that a push was dirty increases

with cache size, since the pushed block will have been resident longer and hence have a higher probability of having been written.

Write-back stack analysis allows us to compute the probability of a dirty push directly and to compare this to the previous approximation. Although the probability of a dirty push is not needed directly in the computation of transfer ratios using equation (2.5), it is useful in its own right, for example as a parameter of a queuing model of a memory system.

5.2.1. The Trace Data

The traces used in these comparisons consist of instruction and data addresses from the execution of programs on one of several machines. They represent a variety of different applications in three different languages. The traces are: FGO1 (IBM 370, Fortran execution, factor analysis), FGO2 (IBM 370, Fortran execution, analysis of satellite data), MVS (standard MVS operating system workload at Amdahl Corp.), LISPCOMP (VAX, LISP compiler, written in LISP), SPICE (VAX, Spice circuit simulator, written in Fortran), VAXIMA (VAX symbolic algebraic manipulation program derived from Macsyma, written in LISP), and RISC (simulated execution of a C compiler for a RISC-architecture processor). All traces except MVS represent the execution of a single program. Most have been used in previous studies [Smit85a]. We used a 16-byte block size for all traces, as used in [Smit85a], and as is shown to be a good choice in [Smit86]. The simulations using these traces considered only data caching; instruction fetches were ignored because they are never writes, and to compare the write back results to those of Smith [Smit85a].

Two sets of experiments were done with each program address trace. First, each was simulated as if it were a stand-alone program. This provides a characterization of each program, but generally gives an optimistic prediction of the actual performance of the program in a multi-programming environment [East78]. In the second experiment we used the technique proposed by Smith [Smit82] to approximate the effects of multiprogramming by writing all dirty blocks and flushing the cache after a fixed time quantum, in our case every 20,000 memory references. The simulations without flushing were warm started; the others were not, producing some variation in the number of read/write events shown in Figure 5.2 for the same trace file.

By way of comparison, we also ran simulations using UNIX 4.2 BSD file system traces generated on three university research computers. The traces are identified by machine: ARPA is a VAX 11/780 used for operating system research and development and text processing; ERNIE is a VAX 11/780 used by staff and graduate students for program development and text processing; CAD is a VAX 11/750 used for computer aided design research. All three machines are also used

General Trace File Characteristics							
File	Number of Events	Types of Events Read	Write	Unique Blocks	Mean Stack Size	Dirty Blocks	Mean Dirty
Program Address Traces							
FGO1N	233727	66.5%	33.5%	2675	970.03	1341 (50.1%)	501.21 (51.7%)
FGO2N	182290	79.4%	20.6%	1049	614.51	660 (62.9%)	403.03 (65.6%)
MVSN	244292	63.4%	36.6%	4972	2672.71	3499 (70.4%)	1688.25 (63.2%)
LISPCOMPN	224856	62.1%	37.9%	1764	1370.19	660 (37.4%)	423.03 (30.9%)
RISCRN	45975	81.3%	18.7%	902	675.92	500 (55.4%)	310.02 (45.9%)
SPICEN	190460	63.2%	36.8%	602	555.75	347 (57.6%)	322.11 (58.0%)
VAXIMAN	238237	65.1%	34.9%	4326	3035.69	1263 (29.2%)	717.78 (23.6%)
Average	194262	68.7%	31.3%	2327	1413.54	1181 (50.8%)	623.63 (44.1%)
Program Address Traces with Flushing							
FGO1	234696	65.8%	34.2%	5293	154.91	3018 (57.0%)	88.56 (57.2%)
FGO2	182839	79.3%	20.7%	4530	133.45	2106 (46.5%)	60.90 (45.6%)
MVS	244168	63.4%	36.6%	17174	424.75	8921 (51.9%)	221.79 (52.2%)
LISPCOMP	237867	62.1%	37.9%	11875	317.33	2889 (24.3%)	78.67 (24.8%)
RISCR	50336	81.2%	18.8%	4357	107.46	1186 (27.2%)	29.60 (27.5%)
SPICE	194174	63.3%	36.7%	5642	164.37	2253 (39.9%)	66.67 (40.6%)
VAXIMA	238211	65.4%	34.6%	14011	353.99	3556 (25.4%)	91.79 (25.9%)
Average	197470	68.6%	31.4%	8983	236.61	3418 (38.0%)	91.14 (38.5%)
UNIX File System Traces							
ERNIE	475471	74.7%	25.3%	85119	8879.06	69024 (81.1%)	4021.80 (45.3%)
ARPA	492040	72.0%	28.0%	93930	7254.39	81002 (86.2%)	3471.87 (47.9%)
CAD	489962	56.8%	43.2%	103488	11370.72	87180 (84.2%)	4554.18 (40.1%)
Average	485824	67.8%	32.2%	94179	9168.06	79068 (84.0%)	4015.95 (43.8%)

Table 5.2 General characteristics of trace files.

extensively for electronic mail. These traces were analyzed in detail by Ousterhout et al. [Oust85]. The trace events show logical file creation, deletion, opens, closes, and seeks. Actual reads and writes are not recorded, however each close or seek event includes the range of bytes read or written since the last positioning event for the file. The simulator recreates reads and writes in block size units based on this information. These traces tend to overestimate the miss ratio, since some of the simulated reads/writes were actually several small requests. For these simulations we used a block size of 4096 bytes, consistent with common UNIX 4.2BSD usage. There is no information on program paging, or file system overhead activity such as directories. All files are identified by a logical identifier; there is no data on physical location [Oust85].

5.2.2. General Characteristics

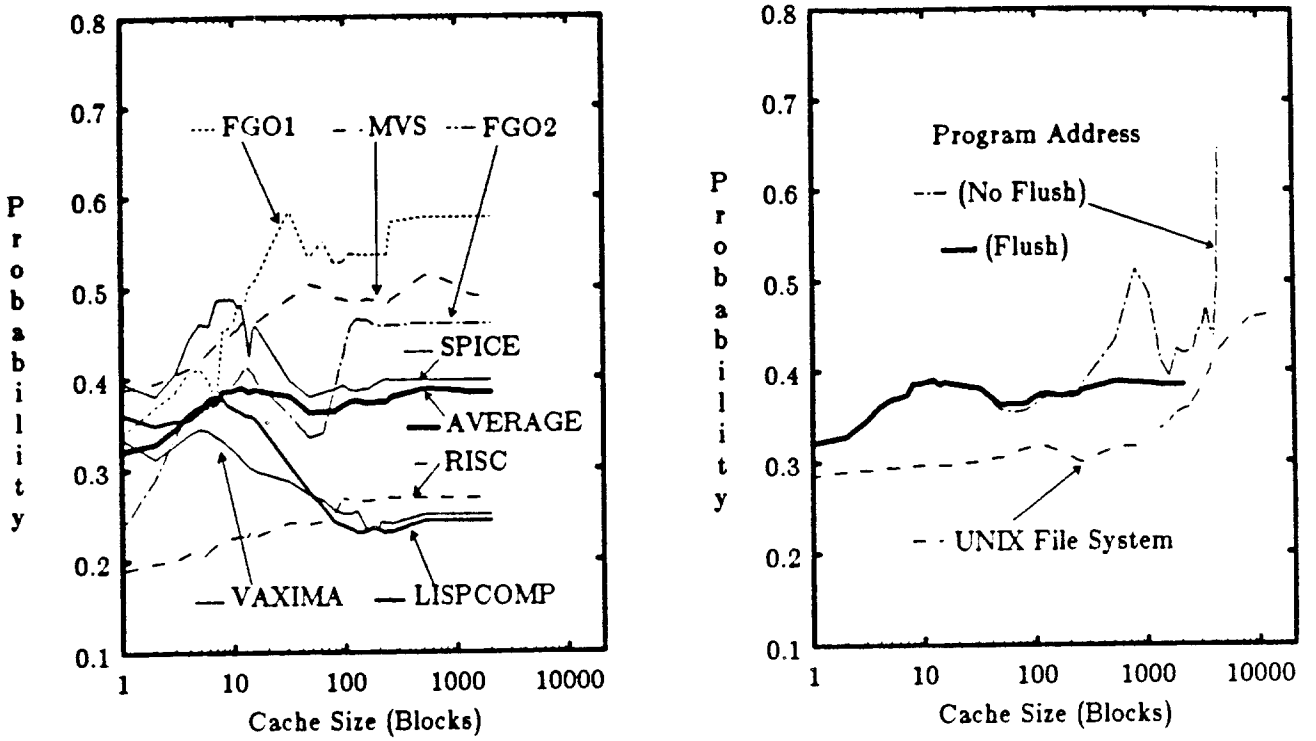
Table 5.2 shows the general characteristics of the traces. We processed approximately 500,000 events from each file after warm start, but only counted the data references, ignoring instruction fetches for the program address traces.

We initially speculated that writes would be a more significant percentage of the file system traces. However when instruction fetches are excluded from the traces to simulate a data-only cache the fraction of writes are comparable. We conclude that writes are a factor which should not be overlooked in any cache design.

The fourth column shows the number of *unique blocks* seen in each trace file. This number is exaggerated by cache flushing because a block reloaded after a flush is considered a new block. It is clear, however, that the file system traces come from a much larger population of blocks. The next column shows the *mean stack size*, that is, the stack distance of the least recently used block averaged over all trace references. These two columns together indicate the range of interesting cache sizes for study. For example, program address traces with flushing seldom use more than a few hundred blocks, whereas 10,000 blocks may be too few for a file system simulation. Notice that deletions and the resulting gaps tend to reduce the mean stack size compared to the number of unique blocks for the file system traces.

The final two columns indicate the extent of write activity. The column labeled *dirty blocks* shows the number of blocks which are ever written. This figure is also shown as a percentage of the number of unique blocks. The fraction of the blocks in the cache which are dirty will obviously affect the chances that a block must be written when it is pushed. The file system traces have far more dirty blocks (84% compared to 50%). The final column shows the *mean number of dirty* blocks in the cache, shown also as a percentage of the mean stack size. Although there is a wide variation between the individual program address traces, we find that on the average, the fraction of the cache which is dirty is about the same, near 44%, for both programs and files. The reason for the relatively low value for the file system traces, compared to the fraction of written blocks, is that most blocks are deleted before they are pushed very far down the stack, and most of the deleted blocks have been written. The blocks which "survive" seem to be equally likely to have been written.

In Figure 5.1 we show the probability that a pushed block is dirty as a function of size. We see that on the average the projected increasing trend holds, although the probability for our traces was closer to 40%. However, the trend for individual traces is not consistent, and some show a distinct *downward* trend. Tables 5.3-5.5 show the same data, as well as the percent of references which are writes and the percent of blocks which are ever written. Notice that the two traces which show the strongest downward trend are the two LISP traces, LISPCOMP and VAXIMA. These are also the only two which have a higher percent write than percent dirty. We believe there is a relation between these observations, as explained below.



(a) Program Address Traces (Flushing)

(b) Average by Type of Trace

Figure 5.1 Probability that a pushed block is dirty, by memory size.

Probability of Dirty Push Program Address Traces With Flushing								
File Name	FGO1	FGO2	LISPCOMP	MVS	RISC	SPICE	VAXIMA	Avg
Percent Writes	34.19	20.75	37.94	36.59	18.79	36.75	34.63	31.38
Percent Dirty	57.02	46.49	24.33	51.94	27.22	39.93	25.38	38.90
Cache Size								
1	33.83	23.57	36.10	38.73	18.93	39.55	33.36	32.01
2	36.88	28.98	34.86	39.50	19.81	38.06	31.18	32.75
4	40.71	36.40	35.41	41.14	20.80	44.40	33.80	36.09
8	45.31	38.77	37.77	44.35	22.29	48.77	33.09	38.62
16	50.99	39.76	35.44	46.34	22.75	45.87	29.83	38.71
32	58.58	36.02	30.33	48.95	24.16	39.78	28.62	38.06
64	55.09	33.99	25.80	49.99	24.29	38.39	26.55	36.30
128	53.67	46.68	23.03	48.96	26.53	38.57	25.42	37.55
256	57.26	45.86	23.06	48.65	26.84	39.77	23.99	37.92
512	57.76	46.03	24.37	51.33	26.84	39.80	25.09	38.75
1024	57.76	46.03	24.37	50.07	26.84	39.80	25.06	38.56
2048	57.76	46.03	24.37	49.07	26.84	39.80	25.06	38.42

Table 5.3 Probability that a pushed block is dirty as a function of cache size.

Probability of Dirty Push Program Address Traces Without Flushing								
File Name	FGO1	FGO2	LISPCOMP	MVS	RISC	SPICE	VAXIMA	Avg
Percent Writes	33.55	20.63	37.94	36.62	18.74	36.80	34.86	31.31
Percent Dirty	50.13	62.92	37.42	70.37	55.43	57.64	29.20	51.87
Cache Size								
1	33.44	23.50	36.15	38.77	18.87	39.59	33.64	31.99
2	36.46	28.88	34.91	39.52	19.65	38.12	31.46	32.71
4	40.57	36.25	35.51	41.16	20.66	44.45	34.08	36.10
8	45.29	38.61	37.81	44.39	22.24	48.77	33.44	38.65
16	50.77	39.53	35.43	46.38	22.35	45.83	30.13	38.63
32	58.48	35.36	30.22	48.98	23.65	39.26	28.94	37.84
64	53.79	32.54	25.29	50.13	22.51	36.72	26.90	35.41
128	51.06	54.33	21.83	48.52	23.49	36.09	24.63	37.14
256	51.93	56.27	20.82	47.85	32.90	36.40	22.66	38.40
512	49.40	65.61*	22.80	53.07	55.60*	35.10*	22.25	43.40
1024	49.67	80.00*	31.15*	61.28	0.00	0.00	20.25	48.47
2048	44.50*	0.00	0.00	62.24	0.00	0.00	18.88*	41.87
4096	0.00	0.00	0.00	74.54	0.00	0.00	24.78*	49.66
* Based on fewer than 500 pushes.								

Table 5.4 Probability that a pushed block is dirty as a function of cache size.

Probability of Dirty Push UNIX File System Traces				
File Name	ARPA	CAD	ERNIE	Avg
Percent Writes	26.79	39.14	24.05	29.99
Percent Dirty	86.24	84.24	81.09	83.86
Cache Size				
1	27.42	33.52	24.42	28.45
2	27.88	33.97	24.93	28.93
4	28.14	34.44	25.06	29.21
8	28.37	35.06	25.21	29.55
16	28.83	34.68	25.36	29.62
32	30.02	34.32	25.76	30.03
64	31.24	34.69	27.00	30.98
128	31.47	33.57	29.79	31.61
256	28.56	30.74	30.55	29.95
512	30.22	30.11	34.14	31.49
1024	32.47	29.92	34.62	32.34
2048	37.19	33.52	36.52	35.74
4096	48.52	36.19	40.42	41.71
8192	50.70	44.39	42.32	45.80
16384	0.00	28.36	0.00	28.36

Table 5.5 Probability that a pushed block is dirty as a function of cache size.

First, notice that the probability of a dirty push from a single-block cache is close to the probability of a write, for all traces. This is certainly reasonable since most blocks are pushed from the single-block cache shortly after they are referenced. For other small cache sizes (in the range 2-15), the chance that a block has been written does increase as predicted, and most of the traces exhibit an upward trend. However, some blocks are never written, and the chance that a clean block will ever be written decreases as it is pushed down the stack.

At the other extreme, notice that without flushing (Table 5.4) the number of pushes eventually reaches zero for cache sizes which hold all the blocks of the program. Therefore, when flushing is used (Table 5.3), all of the pushes from large cache sizes are due to flushing. The probability that a block flushed from a large cache is dirty should be very close to the fraction of blocks which are dirty, as it is.

Since we can predict the probability of a dirty push from both small and large caches, we naturally expect that the trend should be from the percentage of writes to the percentage of dirty blocks. This explains the observed results, but suggests that they may be an artifact of the flushing methodology. We believe this is not the case and offer another explanation.

In Table 5.6 we classify all blocks into one of four classes: read only, read/write (in no particular order), write only, and write once/read (e.g. a variable which is initialized and subsequently only read). The latter class we expect to be small in program address traces and larger in the file system traces. The table also classifies all events as to the class of block they reference. Both of the LISP traces show a surprisingly large fraction of read-only blocks. At the same time these blocks receive a relatively low fraction of references. Therefore a few "dirty" blocks are receiving most of the references, and therefore are more likely to stay near the top of the stack. Thus the blocks being pushed from larger caches are more likely to be from the large class of clean, read-only blocks. This generally explains the decline in the probability of a dirty push, independent of whether we use flushing. We can not say whether this phenomena is characteristic of LISP programs in general.

In passing we note that the UNIX file system traces also show an increasing trend in the probability that a pushed block is dirty, although not nearly as much as might be predicted based on the fact that 85% of all blocks are eventually written. The difference again is caused by deletions, leaving about half of the remaining blocks dirty. We also note that the class of write once/read is very common for file system blocks, as predicted.

Percentage of Blocks and Reference By Class								
File	Blocks				References			
	Read Only	Read/ Write	Write Once/ Read	Write Only	Read Only	Read/ Write	Write Once/ Read	Write Only
Program Address Traces								
FGO1	49.87	5.27	0.07	44.78	15.14	82.40	0.15	2.31
FGO2	37.08	53.77	0.76	8.38	47.49	51.76	0.02	0.74
MVS	29.63	28.30	0.24	41.84	16.69	78.90	0.03	4.38
LISPCOMP	62.59	37.14	0.11	0.17	19.51	80.42	0.00	0.06
RISCR	44.57	53.55	0.00	1.88	34.02	64.20	0.00	1.78
SPICE	42.36	51.83	0.17	5.65	12.93	85.96	0.00	1.11
VAXIMA	70.80	28.18	0.16	0.85	24.05	75.89	0.01	0.05
Average	48.13	36.86	0.22	14.79	24.26	74.22	0.03	1.49
Program Address Traces (With Flushing)								
FGO1	42.98	21.31	0.11	35.59	16.85	80.91	0.03	2.21
FGO2	53.51	30.50	0.71	15.27	48.89	48.70	0.09	2.32
MVS	48.06	34.00	0.52	17.42	19.76	75.05	0.12	5.06
LISPCOMP	75.67	23.36	0.27	0.70	21.46	78.32	0.04	0.18
RISCR	72.78	23.97	0.41	2.84	37.22	60.49	0.22	2.07
SPICE	60.07	25.90	1.13	12.91	14.63	83.10	0.10	2.17
VAXIMA	74.62	23.45	0.23	1.71	26.00	73.72	0.05	0.22
Average	61.10	26.07	0.48	12.35	26.40	71.47	0.09	2.03
UNIX File System Traces								
ERNIE3	18.91	7.89	32.66	40.54	48.77	21.42	20.56	9.25
ARPA5	13.76	6.06	40.24	39.94	33.09	28.25	28.33	10.32
CAD4	15.76	6.56	36.73	40.95	28.36	36.61	22.62	12.41
Average	16.14	6.84	36.54	40.48	36.74	28.76	23.84	10.66

Table 5.6 Percentage of blocks and references by block class.

6. Conclusions

In this paper we have shown how stack analysis can be extended to two important new areas. The ability to collect transfer ratios for all memory sizes in a single pass reduces simulation time by as much as 90%, and makes this metric much more reasonable to collect. The transfer ratio is increasingly important in the study of shared-memory systems, particularly for file systems. Equally important, the ability to easily simulate sector caches, including writes and a form of prefetch, opens up a variety of new cache designs to efficient analysis.

7. Acknowledgements

We wish to express our appreciation to Mark Hill who observed that our write-back technique also applied to sector caches, and provided valuable comments on early drafts of this paper. Frank Olken also provided useful comments and advice on the paper, and was a major contributor to the referenced

companion paper on simulation techniques. The material presented here is based on research supported in part by the National Science Foundation under grant CCR-8202591, by the IBM Corporation and Digital Equipment Corporation under computer equipment grants to UC Berkeley, and by the State of California, the Hewlett Packard Corporation, the IBM Corporation and the Signetics Corporation under the MICRO program.

8. References

[Bela66] Belady, L. A., A Study of Replacement Algorithms for Virtual Storage Computers, *IBM Systems Journal 5* , 2 (1966), 78-101.

[Bela69] Belady, L. A., R. A. Nelson and G. S. Shedler, An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine, *Communications of the ACM 12*, 6 (June 1969), 349-353.

[Cho86] Cho, J. and H. Sachs and A. J. Smith, The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER Processor, *UC Berkeley CS Division Technical Report UCB/CSD 86/289*, March, 1986. Submitted for publication..

[Coff71] Coffman, E. G. and B. Randell, Performance Prediction for Extended Paged Memories, *Acta Informatica 1*, 1 (1971), 1-13.

[Denn72] Denning, Peter J., On Modeling Program Behavior, *Proc. SJCC*, 1972, 937-944.

[East78] Easton, M. C., Computation of Cold-start Miss Ratios, *IEEE Trans. Comp. TC-27*, 5 (May 1978), 404-408.

[Gecs74] Gecsei, J., Determining Hit Ratios in Multilevel Hierarchies, *IBM J. Res. and Dev. 18*, 4 (July 1974), 316-327.

[Gibs86] Gibson, G., Personal communication.

[Good83] Goodman, J. R., Using Cache Memory to Reduce Processor-Memory Traffic, *Proc. Tenth Int'l. Symp. on Computer Architecture*, Stockholm, Sweden, June 1983, 124-131.

[Gree74] Greenberg, Bernard S., An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory, *MAC Technical Report-127*, Cambridge, MA, Jan. 1974.

[Gros85] Grossman, C. P., Cache-DASD Storage Design for Improving System Performance, *IBM System Journal 24*, 3/4 (1985), 316-334.

[Hill84] Hill, Mark D. and A. J. Smith, Experimental Evaluation of On-chip Microprocessor Cache Memories, *Proc. 11'th Ann. Symp. on Computer Architecture*, Ann Arbor, Michigan, June 1984, 158-166.

[Hors83] Horspool, R. N. and R. M. Huberman, Demand Prepaging Algorithms with the Memory Inclusion Property, *Proc. 16'th Ann. Hawaii Int'l. Conf. on System Sciences*, 1983, 138-145.

[Katz85] Katz, R., S. Eggers, D. A. Wood, C. Perkins and R. G. Sheldon, Implementing a Cache Consistency Protocol, *Proc. 12'th Int'l. Symp. on Computer Architecture*, June 1985, 276-283.

[Kubo75] Kubo, Hidehito and Makoto Kobayashi, A Cost-Oriented Approach to Optimal Page Size, *Proc. Second USA-Japan Computer Conference*, 1975, 258-263.

[Lipt68] Liptay, J. S., Structural Aspects of the System/360 Model 85, Part II: The Cache, *IBM Systems Journal 7 (1)* (1968), 15-21.

[Matt70] Mattson, R. L., J. Gecsei, D. Slutz and I. L. Traiger, Evaluation Techniques for Storage Hierarchies, *IBM Syst. J. 9*, 2 (1970), 78-117.

[Olke81] Olken, F., *Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies*, Master's Report, University of Cal., Berkeley, Cal., May 1981.

[Oust85] Ousterhout, John K., Herve' DaCosta, David Harrison, John A. Kunze, Mike Kupfer and James G. Thompson, A Trace-Driven Analysis of the UNIX 4.2BSD File System, *Proc. Eleventh Annual Symposium on Operating Sys. Principles*, Dec. 1985, 15-24.

[Shed76] Shedler, G. S. and D. R. Slutz, Derivation of Miss Ratios for Merged Access Streams, *IBM J. Res. Dev. 20*, 5 (Sept. 1976), 505-517.

[Silb83] Silberman, G. M., Stack Processing Techniques in Delayed-Staging Storage Hierarchies, *Communications of the ACM 26*, 11 (Nov. 1983), 999-1007.

[Slut72] Slutz, D. R. and I. L. Traiger, Determination of Hit Ratios for a Class of Staging Hierarchies, *IBM Res. Rep RJ 1044*, May 1972.

[Smit77] Smith, A. J., Two Methods for the Efficient Analysis of Memory Trace Data, *IEEE Trans. on Softw. Engr. SE-3*, 1 (January 1977), 94-101.

[Smit78a] Smith, A. J., A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory, *IEEE Trans. Softw. Engr. SE-4*, 2 (March 1978), 121-130.

[Smit78b] Smith, A. J., Sequential Program Prefetching in Memory Hierarchies, *Computer 11*, 2 (Dec. 1978), 7-21.

[Smit79] Smith, A. J., Characterizing the Storage Process and its Effect on the Update of Main Memory by Write-through, *J. ACM 26*, 1 (Jan 1979), 6-27.

[Smit82] Smith, A. J., Cache memories, *Computing Surveys 14*, 3 (Sept. 1982), 473-530.

[Smit85a] Smith, A. J., Cache Evaluation and the Impact of Workload Choice, *Proc. 12'th Ann. Symp. on Computer Architecture*, Boston, Mass., June 1985, 64-73.

[Smit85b] Smith, A. J., Disk Cache - Miss Ratio Analysis and Design Considerations, *ACM Trans. on Comp. Sys.*, August 1985, 161-203.

[Thom87] Thompson, J., A. J. Smith and F. Olken, Efficient Methods for Calculating the Miss Ratio for Disk and Processor Caches, To appear, 1987.

[Toku80] Tokunaga, T., Y. Hirai and S. Yamamoto, Integrated Disk Cache Systems with File Adaptive Control, *Proc. IEEE Computer Society Conf.*, Washington, DC, Sept. 1980, 412-416.

[Trai71] Traiger, I. L. and D. R. Slutz, One Pass Techniques for the Evaluation of Memory Hierarchies, *IBM Research Report RJ 892*, Yorktown Heights, NY, July 1971.

[Yu76] Yu, F. S., *Modeling the Write Behavior of Computer Programs*, PhD Thesis, Stanford University, Palo Alto, CA, May 1976.