NASA Contractor Report 189571

ICASE Report No. 91-83

1

ICASE

MASSIVELY PARALLEL ALGORITHMS FOR TRACE-DRIVEN CACHE SIMULATIONS

David M. Nicol
Albert G. Greenberg
Boris D. Lubachevsky

Contract No. NAS1-18605
November 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Massively Parallel Algorithms for Trace-Driven Cache Simulations

David M. Nicol [*]
College of William and Mary

Albert G. Greenberg
Boris D. Lubachevsky
AT & T Bell Laboratories

Abstract

Trace-driven cache simulation is central to computer design. A trace is a very long sequence, x_1, ..., x_N, of references to lines (contiguous locations) from main memory. At the t^{th} instant, reference x_t is hashed into a *set* of cache locations, the contents of which are then compared with x_t. If at the t^{th} instant x_t is not present in the cache, then it is said to be a *miss* and is loaded into the cache set, possibly forcing the replacement of some other memory line, and making x_t present for the $(t+1)^{st}$ instant. The problem of parallel simulation of a subtrace of N references directed to a C line cache set is considered, with the aim of determining which references are misses and related statistics.

A simulation method is presented for the Least-Recently-Used (LRU) policy, which regardless of the set size C runs in time $O(\log N)$ using N processors on the exclusive read, exclusive write (EREW) parallel model. A simpler LRU simulation algorithm is given that runs in $O(C \log N)$ time using $N/\log N$ processors. We present timings of the second algorithm's implementation on the MasPar MP-1, a machine with 16384 processors. A broad class of *reference-based* line replacement policies are considered, which includes LRU as well as the Least-Frequently-Used and Random replacement policies. A simulation method is presented for any such policy that on any trace of length N directed to a C line set runs in time $O(C \log N)$ time with high probability using N processors on the EREW model. The algorithms are simple, have very little space overhead, and are well-suited for SIMD implementation.

[*] This research was supported in part by NASA grants NAG-1-1132 and NAS-1-18605, in part by NSF Grant ASC 8819373, and was initiated during a visit to AT&T Bell Laboratories.

1 Introduction

A cache is a high-speed memory on the access path to a larger, slower main memory. Cache performance is critical to the overall performance of computer systems [10], and consequently a tremendous amount of effort is put into the evaluation of cache designs. This is particularly true for RISC microprocessor designs, where the ratio of the time needed to access an off-chip cache to that needed to access the main memory can be as high as 10 [10], and the off-chip cache is typically at least 10 times smaller than the main memory. Trace-driven simulations, which evaluate cache performance on actual reference streams taken from characteristic programs, are the most reliable and widely used tools for cache design evaluation. These simulations require a great deal of computation, because of the many different design possibilities that are simulated, and because of the length of the reference traces that drive the simulation.~~[10]~~

Data is moved between main memory and the cache in contiguous blocks called *lines*. Every memory line is hashed to some fixed cache *set*, but may be placed in any one of the C physical cache lines in the set. In emerging computer designs, a microprocessor might be supported by a 1 Megabyte off-chip cache, with a line size of 128 bytes, and a set size $C = 4$. A *miss* occurs whenever a memory line is referenced, but is not found in its set. The cache hardware then fetches the desired line from main memory, overwriting another line in the same set if the set is full. The rule used to select which line to replace is called the *replacement policy*. An effective, widely used policy is Least-Recently-Used (LRU), which simply replaces the line accessed least recently. The objective of a trace-driven simulation is to determine which references in the trace are misses. Given the identities of the misses, statistics of chief interest in cache design are easily computed, such as the fraction of read misses, the fraction of write misses, and the number of write-backs (stores of modified lines) from cache to main memory.

Heidelberger and Stone [9] showed that it is valuable to simulate a long trace directed to a few sets, when cache miss statistics between sets are highly correlated.[1] High correlation removes the need to simulate all sets, but also removes the easy parallelism that might be exploited by simulating a large number of sets in parallel on different processing elements (PEs). A massively parallel method to handle the simulation of a long trace targeted to a single set allows more powerful, flexible solutions.

We consider the problem of determining the misses in a given reference trace, x_1 , ..., x_N, directed to a set of size C. An algorithm is presented (Section 3) that solves this problem in $O(\log N)$ time using $N/\log N$ PEs, on the exclusive-read, exclusive-write (EREW) model of a parallel machine. The algorithm and its complexity do not depend on C. The algorithm computes the stack distance Δ_t associated with each reference x_t [16]. If x_t is not a first reference to a line then Δ_t is the smallest set size for which x_t would be a hit; otherwise $\Delta_t = \infty$.

In Section 4, we present an alternative LRU simulation, with running time $O(C \log N)$ time using $N/\log N$

[1] Recent experiments (private communication from Harold Stone) have validated that high correlation exists between sets, but have also shown that special care must be taken when selecting the sets which are analyzed, as the measured miss ratio from an arbitrary set simulation may not be an accurate predictor of the overall miss ratio.

PEs on the EREW model. The algorithm computes the stack distance at level C, $\Delta_t(C)$, for each reference x_t. If x_t is not a first reference to a line then $\Delta_t(C)$ is the smallest set size $\leq C$ for which x_t would be a hit; otherwise $\Delta_t(C) = \infty$. The algorithm is simple and the implicit constant in the time bound is favorable. We report timings of this algorithm's performance on a MasPar [4] SIMD computer having 16384 PEs.

In Section 5, a broad class of *reference-based* replacement policies is considered. Roughly, the class contains all stack replacement policies where priorities controlling line replacement are static and can be computed efficiently in parallel. This class includes LRU as well as:

- OPT: Replace the line referenced most remotely in the future. This unrealizable policy provably minimizes the number of misses. Its simulation gives a baseline against which realizable policies can be measured.

- Least-Frequently-Used or LFU: Replace the line accessed least often in the past. Ties can be broken by, for example, giving higher priority to the reference that has been in the cache the shortest length of time.

- Random: Replace one of the C lines, chosen independently and uniformly at random. Random replacement is easy to implement; furthermore, there is evidence that if the total number of lines in the cache (not just the lines in one set) is sufficiently large, the policy works nearly as well as any other implementable policy[10].

In Section 5, an algorithm is presented for reference-based policy simulation. Given any trace of N references targeted to a C line set, the algorithm runs in time in $O(C \log N)$ with high probability using N PEs on the EREW model. (The algorithm is probabilistic, the choice of trace is not.) In Section 5.3, we extend the class of reference-based replacement policies to include an aging mechanism, whereby stale lines lose priority and tend to be flushed from the cache. Accommodating this mechanism increases the algorithm's running time to $O(C \log^2 N)$.

Our algorithms are simple, require at most $O(\log N)$ space per PE, and break the computation down into calls to a few primitive parallel subroutines. As a result the algorithms are well-suited for SIMD architectures, such as the Connection Machine [11] or MasPar [4]. The $O(C \log N)$ with high probability bound holds because we have assumed that a fast probabilistic parallel algorithm [18] is used to solve a certain trapezoidal decomposition problem (Sections 2, 5). Adopting the notation of [18], this algorithm runs in $\tilde{O}(\log N)$ time using N PEs, meaning that there is a constant k such that the time exceeds $km \log N$ with probability less than N^{-m} for any $m > 1$. In practice, simpler, deterministic methods may do better, while raising the asymptotic time bound to $O(C \log^2 N)$.

For simplicity, we have assumed the problem size N is comparable to the number of PEs, so that it is as if each PE handles a few references (up to $\log N$). However, a "supersaturated" setup [8] may be effective in practice, where a large block of consecutive references would be loaded in the local memory of each PE. Our algorithms generalize to that setup, by using efficient supersaturated implementations of the underlying parallel primitives (cf. [12, 17]). Indeed, our implementation of the LRU algorithm is a supersaturated one, with complexity $O(C(N/P + \log P))$ for a reference trace with N elements on an architecture with P processors.

Collecting the cache miss statistics mentioned above adds just $O(\log N)$ time. Moreover, by the nature of the replacement policies and the simulation methods, statistics for each set size up to C can be computed at this cost. All of our algorithms can be adapted for efficient simultaneous simulation of many sets, by the simple device of initially sorting the references on the basis of their set identifiers.

Heidelberger and Stone [9] had the original insight that trace-driven simulation of an LRU cache set could be parallelized. Their algorithm is intended for a network of P MIMD processors, and requires $P \ll N$ for good speedup. Our work was motivated by theirs; our algorithms are different, apply to a larger class of replacement policies, and to a different class of architectures. Lin, Baer, and Lazowska have considered parallelizing cache simulations, in the context of multiprocessor cache protocols[15]. Their method assumes that each individual processor's cache is simulated on a different PE, so that the degree of parallelism is limited to the number of caches in the simulated system. An important and beautiful paper on cache simulation was published in 1970 by Mattson, Gecsei, Slutz, and Traiger[16]. Most of our notation is taken from that paper.

The practical utility of implementing trace-driven cache simulations on today's SIMD computers has yet to be shown, although our implementation proves the great promise of the approach. It seems likely that a very long reference trace will have to be partitioned into blocks, where one block is processed at a time. The I/0 problem is to move the blocks to the processors fast enough to keep them busy. An attractive alternative is to use a synthetic trace; for example Thiebáubt, Stone, and Wolf [20] recently proposed a simple method for random generation of realistic traces.

2 Preliminaries

2.1 Cache Notation

Henceforth, we focus on a single set cache, and treat its size C as a parameter. Let $B_t(C)$ denote the set of lines stored just after reference x_t. Each reference must be cached, so $x_t \in B_t(C)$ for all $C \geq 1$ and $t \geq 1$. (By convention, $B_t(0)$ is the empty set.) If the cache is full ($|B_t(C)| = C$) and x_t is a miss ($x_t \notin B_{t-1}(C)$) then x_t replaces a line in $B_{t-1}(C)$. We refer to this replaced line as y_t. All of the replacement policies we consider are *stack* policies [16], meaning if a reference is a hit given that the cache size is C then the reference will remain a hit if the cache size is increased to $C+1$. That is,

$$B_t(C) \subset B_t(C+1) \quad \text{for all } C \geq 0.$$

This inclusion allows us to order the lines of the cache by the least size needed for their appearance.

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$s_t(1)$	$\emptyset$	a	c	b	c	c	a	c	a	d	b	b	d	c	d	a	a	b	a
$s_t(2)$	$\emptyset$	$\emptyset$	a	c	b	b	c	a	c	a	d	d	b	d	c	d	d	a	b
$s_t(3)$	$\emptyset$	$\emptyset$	$\emptyset$	a	a	a	b	b	b	c	a	a	a	b	b	c	c	d	d
$s_t(4)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	b	c	c	c	a	a	b	b	c	c
Δ_t		∞	∞	∞	2	1	3	2	2	∞	4	1	2	4	2	4	1	4	2

Figure 1: An example of the LRU rule acting on an $N = 18$ line trace, with lines labeled $a - d$; $\emptyset$ is the empty line marker.

Define the i^{th} element of $B_t(C)$ as

$$s_t(i) = \begin{cases} B_t(i) - B_t(i-1) & \text{if } |B_t(i)| = i \\ \emptyset & \text{otherwise} \end{cases} . \tag{1}$$

The symbol $\emptyset$ is an empty line marker; $s_t(i) = \emptyset$ if fewer than i lines belong to $B_{t-1}(i)$. We assume the stack starts empty, and therefore let $s_0(i) = \emptyset$ for all $i \geq 0$.

Figure 1 gives an example, for the LRU replacement policy. The trace length $N = 18$, and the lines are labeled $a - d$. The first level, $s_t(1)$ coincides with the trace itself. Consider the first two levels; i.e., the cache contents given $C = 2$. The cache is initially empty. The first two references, $x_1 = a$, $x_2 = c$, miss, and as a result $B_1(2) = \{a\}$, $B_2(2) = \{a, c\}$. The third reference, $x_3 = b$, also misses, forcing the replacement of $y_3 = a$, yielding $B_3(2) = \{b, c\}$. The fourth reference, $x_4 = c$ hits, so $B_4(2) = B_3(2)$, and so forth.

Hit and miss statistics are easily extracted from stack distances, defined as follows. Let the level i stack distance $\Delta_t(i)$ denote the smallest cache size $\leq i$ such that x_t is a hit, or ∞ if x_t is a miss for cache size i. Thus, given $\Delta_t(C)$, for $t = 1, \ldots, N$, we can extract the hits for any cache size $c < C$. More generally, define the stack distance $\Delta_t = \lim_{C\to\infty} \Delta_t(C)$ to be the smallest cache size such that x_t is a hit, or ∞ if there is no prior reference to the same line ($x_s = x_t$ for some $s < t$). Stack distances are shown in Figure 1.

2.2 Parallel Processing Model

Our algorithms are well-suited for a wide variety of parallel architectures, because the algorithms transform data in simple ways using a small number of basic, highly parallelizable operations. However, to state precise time and processor requirements, we must choose a precise model of parallel computation. The EREW (exclusive read, exclusive write) model (cf. [14]) provides a nice blend of simplicity and realism. In this model, the PEs operate in lockstep. There is a global shared memory, supporting at unit cost any pattern of accesses except those where two PEs simultaneously access the same location.

We state the complexity of our algorithms with respect to the EREW model. We now list the parallel subroutines used in our algorithms, and their complexities on the EREW model.

- Merging: Two sorted lists each of length N can be merged in time $O(\log N)$ using $N/\log N$ PEs, via Batcher's odd-even merge algorithm [2].

- Sorting: A list of length N can be sorted in time $O(\log N)$ time using N PEs [5].

- 2d Ranking: Given points (x_i, y_i), $i = 1, \ldots, N$, compute for each point (x_i, y_i) its *rank*, the number of other points (x_j, y_j) strictly above and to the right: $x_i < x_j$ and $y_i < y_j$. In slightly different form, this is the problem of computing the empirical cumulative distribution function (ECDF), considered in [3]. The *multidimensional divide-and-conquer* serial algorithm given in [3] parallelizes easily to solve the problem in $O(\log^2 N)$ time using N PEs. Atallah et al. improved on this, lowering the time to $O(\log N)$ using N PEs.

- Closest Larger Right Neighbor (CLRN) Problem: Given input numbers $a_1, \ldots, a_N$, find, for each a_i, the index of the first larger number to the right; i.e., for each $i = 1, \ldots, N-1$, compute $b_i = \min\{j > i : x_j > x_i\}$ if there is some $j > i$ with $x_j > x_i$, $b_i = N + 1$ otherwise. The CLRN problem can be reduced to *trapezoidal decomposition* [18]: given a set of line segments and points, from each point, report the line segment first hit (if any) by a ray shot horizontally to the right. To make the reduction, consider the polygonal path connecting consecutive points (i, a_i), $i = 1, \ldots, N$. If $a_{i+1} > a_i$ then we know $b_i = a_{i+1}$. Otherwise, b_i is the height of the right end point a_j of the segment from $(j-1, a_{j-1})$ to (j, a_j) first hit by the ray shot horizontally to the right from (i, a_i), if there is an $a_j > a_i$, $j > i$. If not then $b_i = N + 1$. Reif and Sen give a probabilistic algorithm for trapezoidal decomposition. Applying that algorithm to the CLRN problems yields its solution in $\tilde{O}(\log N)$ time using N PES. Alternatively, the CLRN problem can be solved by a binary search-like algorithm, given in Section 5, in $O(\log^2 N)$ time using N PEs.

- Parallel Prefix (scan, segmented scan): Given inputs $a_1, \ldots, a_N$ and an associative operator $\circ$, compute the partial products $p_1, \ldots p_N$ where $p_i = a_1 \circ a_2 \circ \ldots \circ a_i$. Solutions to this *parallel prefix* problem [13] are commonly called scan computations. The problem can be solved in $O(\log N)$ time using $N/\log N$ PEs [12].

  A variation breaks the products over the indices $[1, N]$ into segments over these indices, with the segment boundaries also given as inputs. For example, an additional vector $b_1, \ldots, b_N$, is given where $b_1 = 0$, for $i > 1$, b_i is either 0 or 1, and the 0's mark the segments' left boundaries. Specifically, if $b_i = 0$ then $p_i = a_i$; otherwise, $p_i = a_j \circ a_{j+1} \circ \ldots \circ a_i$ where j is the largest index k, $1 \leq k < i$, such that $b_k = 0$. The segmented problem has the same complexity as the original. In the algorithms below, we use copy-scans defined by $\alpha \circ \beta = \alpha$, and add-scans where $\circ$ is addition.

None of the algorithms listed above requires more than $O(\log N)$ space per PE.

3 Fast Parallel LRU Simulation

In this section we present a fast parallel algorithm for computing stack distances under the LRU replacement policy.

LRU may be characterized as follows. Reference to $\alpha = x_t$ places α at the first level of the stack. Until α is referenced again, it can only move down in the stack. Specifically, after α has been pushed to level i it remains there until a reference is made either to α (moving α to level 1) or to a line not stored in levels 1 through $i-1$ (moving α to level $i+1$). As a result, the stack distance Δ_t is one greater than the number of distinct lines in the subtrace between t and the closest prior reference to α (or ∞ if there is no prior reference to α). For example, in Figure 1, consider the consecutive references to line b at $t = 3$ and $t = 10$. The stack distance $\Delta_{10} = 4$ because 3 distinct symbols belong to the subtrace $x_4, \ldots, x_9$. More generally, letting

$$prev(t) = \begin{cases} \max\{s < t : x_s = x_t\} & \text{if } x_s = x_t \text{ for some } s < t \\ 0 & \text{otherwise} \end{cases},$$

we obtain

$$\Delta_t = \begin{cases} 1 + \text{ number of distinct symbols in } x_{prev(t)+1}, \ldots, x_{t-1} & \text{if } prev(t) > 0 \\ \propto & \text{otherwise} \end{cases}.$$

Let us take a geometric view of this new problem of counting distinct symbols within subtraces. As illustrated in Figure 2, identify each reference x_t with the point $(t, next(t))$, where

$$next(t) = \begin{cases} \max\{s > t : x_s = x_t\} & \text{if } x_s = x_t \text{ for some } s > t \\ N+1 & \text{otherwise} \end{cases}.$$

Note that the last references to symbols within the subtrace $x_{prev(t)+1}, \ldots, x_{t-1}$ are identified by those points $(s, next(s))$ satisfying

$$prev(t) < s < t < next(s).$$

These are the points that lie strictly within the rectangle with lower left hand corner $(prev(t), t)$, lower right hand corner (t, t) and sides extending upwards to $(prev(t), N+1)$ and $(t, N+1)$. Again, see Figure 2. Counting these points reduces to 2d-ranking. Specifically, suppose we know the 2d-rank, $rank(u, v)$, of each point (u, v) in the union of sets $\{(t, next(t)) : 1 \le t \le N\}$ and $\{(t, t) : 1 \le t \le N\}$. Then, the stack distance

$$\Delta_t = \begin{cases} rank(prev(t), t) - rank(t, t) & \text{if } prev(t) > 0 \\ \infty & \text{otherwise} \end{cases}.$$

We see from Figure 2 that $\Delta_{10} = 4$ because $rank(3, 10) = 20$ and $rank(10, 10) = 16$.

Now, let us present the detailed simulation method. Suppose that the trace is initially stored in the N-vector $\mathbf{x}$. We use the additional N-vectors $\mathbf{p}$, $\mathbf{next}$, $\mathbf{prev}$, and $\boldsymbol{\Delta}$. Initially, let $\mathbf{p}_t = t$, so $\mathbf{x}_t$ identifies the line and $\mathbf{p}_t$ the trace index of reference x_t. Vectors $\mathbf{next}$, $\mathbf{prev}$, and $\boldsymbol{\Delta}$ will hold permuted copies of the vectors $next$, $prev$, and Δ, respectively. The algorithm is as follows.

1. [Compute *next* and *prev*.] Sort the tuples $(\mathbf{x}_t, \mathbf{p}_t)$ using $\mathbf{x}_t$ as the primary key and $\mathbf{p}_t$ as the secondary key: $(\mathbf{x}_t, \mathbf{p}_t) < (\mathbf{x}_s, \mathbf{p}_s)$ if either $\mathbf{x}_t < \mathbf{x}_s$ or $\mathbf{x}_t = \mathbf{x}_s$ and $\mathbf{p}_t < \mathbf{p}_s$. Thus, the data now in location t of $\mathbf{x}$ and $\mathbf{p}$ was in location $\mathbf{p}_t$ before the sort. For all $t = 1, \ldots, N$, set

$$\mathbf{prev}_t = \begin{cases} \mathbf{p}_{t-1} & \text{if } t > 1 \text{ and } \mathbf{x}_{t-1} = \mathbf{x}_t \\ 0 & \text{otherwise} \end{cases}$$

Reference Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
trace	*a*	*c*	*b*	*c*	*c*	*a*	*c*	*a*	*d*	*b*	*b*	*d*	*c*	*d*	*a*	*a*	*b*	*a*
prev	0	0	0	2	4	1	5	6	0	3	10	9	7	12	8	15	11	16
next	6	4	10	5	7	8	13	15	12	11	17	14	19	19	16	18	19	19

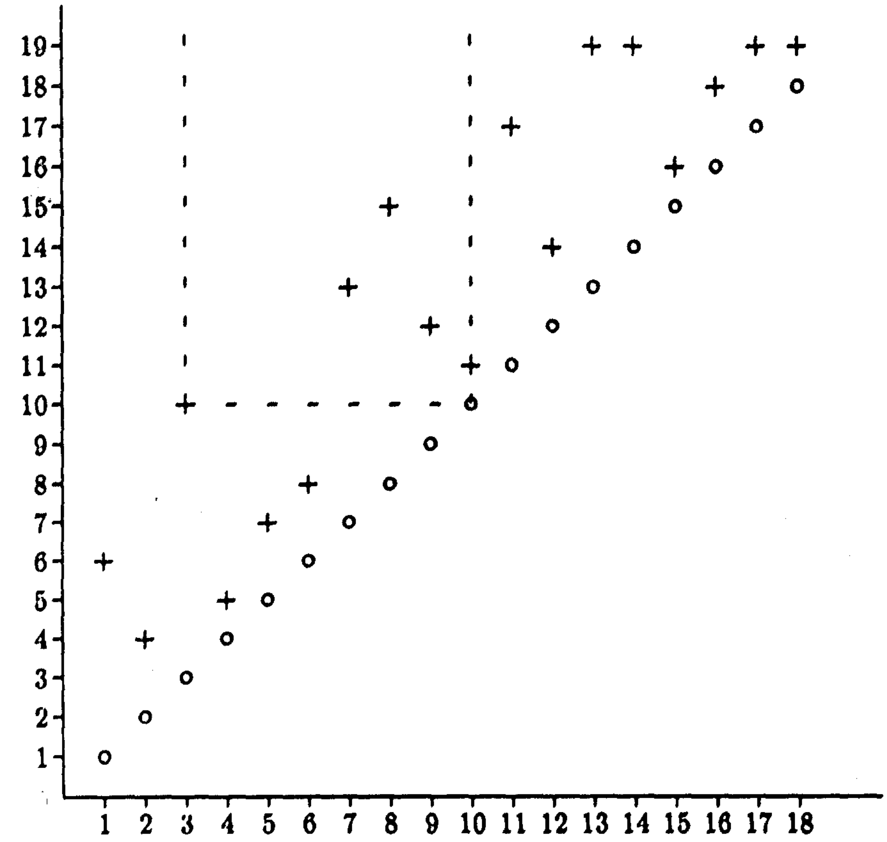

Figure 2: The trace of Figure 1 is repeated, along with corresponding *next* and *prev* values. The values $(t, next(t))$ are plotted as $+$'s, and the values (t,t) as o's. The number of points strictly within the rectangle indicated by dashed lines is one less than the stack distance of $x_{10} = b$.

$$\mathbf{next}_t = \begin{cases} \mathbf{p}_{t+1} & \text{if } t < N \text{ and } \mathbf{x}_{t+1} = \mathbf{x}_t \\ N+1 & \text{otherwise} \end{cases} .$$

At this point, the **prev** and **next** vectors hold permuted copies of the *prev* and *next* vectors discussed above.

2. [2d-rank.] Compute the 2d-ranks of the set of points

$$\{(\mathbf{p}_t, \mathbf{next}_t) : 1 \le t \le N\} \cup \{(t,t) : 1 \le t \le N\}$$

and set

$$\Delta_t = rank(\mathbf{prev}_t, \mathbf{p}_t) - rank(\mathbf{p}_t, \mathbf{p}_t).$$

As a result, $\Delta_t = \Delta_{\mathbf{p}_t}$, which completes the computation.

Sorting within the first step costs $O(\log N)$ time on N PEs, using the EREW model [5]. The **next** and **prev** computations may be done within the same time and processor bounds using segmented copy-scans, with changes in the **x** vector marking the segment boundaries. The 2d-ranking within the second step costs $O(\log N)$ time on N PEs [1]. Thus:

Theorem 1 *On the EREW model, given the trace $x_1, \ldots, x_N$, the associated stack distances $\Delta_1, \ldots, \Delta_N$ induced under the LRU replacement policy can be computed in $O(\log N)$ time using N PEs.*

Aiming for a simpler implementation and smaller implicit constants, we may sacrifice a $\log N$ factor in the running time. The natural parallelization of Bentley's multidimensional divide and conquer method [3] gives a 2d-ranking algorithm that runs in $O(\log^2 N)$ time using N PEs. Using, for example, Batcher's sorting method [2] requires time $O(\log^2 N)$ on N PEs.

4 Parallel Simulation of LRU Level by Level

An alternative approach is to simulate LRU level by level, at the i^{th} iteration computing the level i cache contents $s_1(i), \ldots, s_N(i)$ and stack distances $\Delta_1(i), \ldots, \Delta_N(i)$. Assuming a set size of C, the final results are the stack distances $\Delta_1(C), \ldots, \Delta_N(C)$.

Define reference x_t to be a *prior hit* (*prior miss*) *at level i* if x_t is a hit (miss) given that the cache size is $i-1$. That is, x_t is a prior miss at level i if $x_t \notin B_{t-1}(i-1)$. If x_t is a prior hit at level i then $\Delta_t(i-1) < i$; otherwise $\Delta_t(i) = \infty$. In Figure 3, we have marked the prior hits x_t at level 3 by underscoring the symbol at level 2 in column t. In studying this figure one should remember that an underscore on symbol $s_t(i)$ means that symbol x_t was a hit in a $(i-1)$-line cache, not that $s_t(i)$ was. The placement of underscores was chosen to highlight the propagation of a symbol across a sequence of prior hit positions, to be described below. For example, of the first ten references four are prior hits at level 3—x_4, x_5, x_7, and x_8—because c $(= x_4, x_5, x_7)$ is found in $B_3(2)$, $B_4(2)$ and $B_6(2)$, and symbol a (x_8) is found in $B_7(2)$.

Any prior hit at level $i-1$ is also a prior hit at level i (for example, x_5 in Figure 3). Under LRU, the other prior hits at level i are the references x_t satisfying $x_t = s_{t-1}(i-1)$; i.e., the references that hit at the last level of the size $i-1$ cache (for example, x_4 in Figure 3).

The key to the simulation method is that under LRU, for all $t \geq 1$ and $i > 1$,

$$s_t(i) = \begin{cases} s_{t-1}(i-1) & \text{if } x_t \text{ is a prior miss at level } i \\ s_{t-1}(i) & \text{otherwise} \end{cases}, \tag{2}$$

where

$$s_t(1) \equiv x_t, s_0(i) \equiv \emptyset.$$

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$s_t(1)$	$\emptyset$	a	c	b	c	c	a	c	a	d	b	b	d	c	d	a	a	b	a
$s_t(2)$	$\emptyset$	$\emptyset$	a	c	$\underline{b}$	$\underline{b}$	c	$\underline{a}$	$\underline{c}$	a	d	$\underline{d}$	$\underline{b}$	d	$\underline{c}$	d	$\underline{d}$	a	$\underline{b}$
$s_t(3)$	$\emptyset$	$\emptyset$	$\emptyset$	a	a	a	b	b	b	c	a	a	a	b	b	c	c	d	d
$\Delta_t(3)$	∞	∞	∞	∞	2	1	3	2	2	∞	∞	1	2	∞	2	∞	1	∞	2

Figure 3: LRU acting on an 18 line trace, assuming a cache size $C = 3$. Each prior hit x_t at level 3 is identified by underscoring $s_t(2)$.

To see this, suppose $x_t \notin B_{t-1}(i-1)$. The LRU rule puts x_t into level one, and shifts lines 1, 2, ..., $i-1$ down one level, which pushes $s_{t-1}(i-1)$ to level i. On the other hand, if x_t is a prior hit at level i, the cache update leaves level i unchanged. In Figure 3, we see that the 3^{rd} and the 6^{th} references are prior misses at level 3, and that the intervening references are prior hits. As a result, $s_2(2) = a$ enters level 3 at $t = 3$ and propagates over prior hits at level 3 until $t = 6$, where it is replaced with $s_5(2) = b$, which in turn propagates up through $t = 8$.

We now describe the simulation algorithm, taking special care with the details because similar methods are needed in Section 5. At the $(i-1)^{st}$ iteration we will overwrite vectors $\mathbf{s} = (\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_N)$ and $\mathrm{d} = (\mathrm{d}_1, \mathrm{d}_2, \ldots, \mathrm{d}_N)$ with the level i cache contents and stack distances, $(s_0(i), s_1(i), \ldots, s_N(i))$ and $(\Delta_1(i), \Delta_2(i), \ldots, \Delta_N(i))$, respectively. A vector $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)$ holds the trace $(x_1, x_2, \ldots, x_N)$, and another vector $\mathrm{u} = (\mathrm{u}_1, \ldots, \mathrm{u}_N)$ will hold a copy of $(s_0(i-1), s_1(i-1), \ldots, s_{N-1}(i-1))$. To initialize the computation, for $t = 1, \ldots, N$, set $\mathrm{d}_t = \infty$, and $\mathbf{s}_t = x_t$, $\mathbf{s}_0 = \emptyset$. For $i = 1, \ldots, C$, do as follows.

1. [Update the Level i Cache Contents via equation (2).] For $t = 1, \ldots, N$, set $\mathbf{u}_t = \mathbf{s}_{t-1}$. For $t = 1, \ldots, N$, if $\mathbf{d}_t \neq \infty$ then set $\mathbf{s}_t = \mathbf{s}_{t-1}$; otherwise, $\mathbf{s}_t = \mathbf{u}_t$. This is to be understood, but not implemented, as a serial update: first $\mathbf{s}_1$ is updated, then $\mathbf{s}_2$, and so forth.

2. [Update the Stack Distances.] For $t = 1, \ldots, N$, set $\mathbf{d}_t = i$ if $\mathbf{d}_t = \infty$ and $\mathbf{u}_t = \mathbf{x}_t$; otherwise leave $\mathbf{d}_t$ unchanged.

The right shift of $\mathbf{s}$ into $\mathbf{u}$ in step 1 and the update to the stack distances in step 2 are naturally parallel operations. The update of $\mathbf{s}$ is a segmented copy-scan, with the coordinates t with $\mathbf{d}_t = \infty$ marking the segment boundaries. Hence, the cost of both steps is just $O(\log N)$ time using $N/\log N$ PEs. As there are a total of C iterations to perform, we obtain:

Theorem 2 *On the EREW model, given the trace $x_1, \ldots, x_N$, the associated level C stack distances $\Delta_1(C), \ldots, \Delta_N(C)$ under the LRU replacement policy can be computed in $O(C \log N)$ time using $N/\log N$ processors.*

We implemented this algorithm on a MasPar MP-1 computer [4], with 16384 PEs. Each PE is a 4-bit processor with a clock cycle of 80 nanoseconds. A typical integer operation such as those common in our

	Trace Length											
C	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
4	3.7	3.4	3.9	5.0	7.3	11.4	20	37.1	71.1	139	275	548
8	8.6	8.4	9.6	12	16.7	25.8	44.3	81.3	155	303	599	1190
16	18.6	18.5	20.9	25.8	35.5	54.5	93	170	324	630	1245	2474
32	38.5	38.7	43.6	53.4	73.1	120	190	347	660	1286	2539	5043

Table 1: Execution time of the LRU algorithm on a MasPar MP-1 with 2^{14} PEs, in milliseconds, as a function of trace length and set size

algorithm requires a few ten's of clocks.

Our implementation supports "super-saturation" of the PE's, as described earlier. The PE memory size permit us to assign as many as 2048 references to each PE, thereby permitting the simultaneous simulation of a trace with over 33 million references. The performance data we present includes only the time spent in the solution phase of the algorithm. The traces were generated randomly. For a given trace length and cache set size we observed a 10-15% increase in running time between caches with a verv low hit ratio, and caches with a high hit ratio. This is likely due to the fact that long segments accompany high hit ratios, requiring greater inter-PE communication to implement the copy-scan. The timings presented are from traces with nearly perfect hit ratios, and so represent an upper bound on the timing one might expect from an actual trace.

A full implementation would have to spend time loading the trace; the I/O time required depends on the available I/O hardware and the organization of the trace on the I/O devices. In light of our timings, it is clear that moving the trace onto the machine may well be the most serious bottleneck an actual implementation would face.

Our experiments vary the length of the trace from 2^{14} to 2^{25}, and the set size C from 2 to 32. The presented timings are averages, given in milliseconds, taken by executing the solution loop many times in succession.

Observe that about five seconds of execution time were required to analyze the behavior of a 32-line set on a trace with $2^{25} = 33,554,432$ references. This is 710 times faster than the solution time (with trace generation costs subtracted off) of an optimized serial algorithm we implemented on a Sparc-1+ workstation. These timings demonstrate the remarkable promise of massive parallelism for trace-driven cache simulation.

5 Reference Based Replacement Policies

We now broaden the scope of our methods, to handle a large class of line replacement policies, which we term *reference-based.* In Section 5.3, we extend the class to handle policies that allow priorities associated with cached lines to "age" so that stale lines tend to be flushed.

Mattson et al. [16] show that a stack policy is obtained if a numerical *priority* $P(s_t(i))$ is assigned to each line $s_t(i)$ at reference x_t, and the line $y_t(C)$ chosen for replacement on loading x_t is the one with least priority among the members of $B_{t-1}(C)$. In the class of policies we now consider, a line's priority is established at the point it appears in the reference stream, after which it remains constant until the line is referenced again. Of course we must be able to calculate the priorities from the reference trace. Thus we limit attention to policies that support *efficient* parallel priority calculations. As practical policies seem to use very simple priority assignments, this limitation is mild. Here, "efficient" means within the resource bounds needed for the rest of our simulation method: $\tilde{O}(\log N)$ time using N processors. Recall (Section 2.2) that $\tilde{O}(\log N)$ means $O(\log N)$ with high probability.

Let us define the class of *reference-based* replacement policies as those stack policies induced by priorities satisfying the following conditions.

R1: All $P(x_t)$ values can be computed quickly in parallel: in $\tilde{O}(\log N)$ time using N PEs. For example, the priorities for LFU (Least Frequently Used) can be established with a sort on the reference tags, followed by a segmented sum-scan.

R2: A line's replacement priority does not change except when the line appears in the reference stream.

Several important replacement policies are reference-based, including

- LRU: $P(x_t) = t$.
- LFU: $P(x_t) = \text{Count}(x_t, t)$, the number of references $x_u = x_t$ for $u \leq t$. Ties can be broken, for example, by lexicographic ordering of the lines, or by giving higher priority to the line that has been in the cache the shortest length of time. ($P(x_t) = \text{Count}(x_t, t) - 1/(t+1)$ would serve the latter purpose.)
- OPT: $P(x_t)$ is the negation of the smallest index $u > t$ such that $x_u = x_t$.

In addition, the *Random replacement* (RR) policy shares most of the properties we need to quickly simulate reference-based policies, and we include it in this class as a special case. Under RR, priorities are chosen that determine a uniform random ranking of the cache contents; details are given below.

Figure 4 gives an example of the operation of LFU with ties broken by lexicographic ordering, $a < b < c < d$. A line's subscript equals the number of earlier references to the line. First, note that the stack order and the priority order may differ. A line with low priority can be buried in the middle of the stack order; for example, line d at $t = 13$. There are important departures from the behavior of the LRU policy. Under LRU, the replaced line is the one at the lowest stack level. Here, we see that if the cache size is 2 then at $t = 9$, line d misses and replaces line a at the first stack level, leaving line c in place at the second stack level. To illustrate the entry and propagation of lines across a given level, each prior miss x_t at level 3 is marked by underscoring $s_t(2)$. As in LRU, a line propagates across all prior hits. Unlike LRU, a line may propagate across some prior misses. For example, line a enters level 3 at $t = 9$ and propagates until $t = 15$, across the prior misses at $t = 10$ and $t = 12$.

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
trace	$\emptyset$	a	c	b	c	c	a	c	a	d	b	b	d	c	d	a	a	b	a
$s_t(1)$	$\emptyset$	a_0	c_0	b_0	c_1	c_2	a_1	c_3	a_2	d_0	b_1	b_2	d_1	c_4	d_2	a_3	a_4	b_3	a_5
$s_t(2)$	$\emptyset$	$\emptyset$	a_0	c_0	$\underline{b}_0$	$\underline{b}_0$	c_2	$\underline{a}_1$	$\underline{c}_3$	c_3	c_3	$\underline{c}_3$	c_3	$\underline{d}_1$	$\underline{c}_4$	c_4	$\underline{c}_4$	c_4	c_4
$s_t(3)$	$\emptyset$	$\emptyset$	$\emptyset$	a_0	a_0	a_0	b_0	b_0	b_0	a_2	a_2	a_2	a_2	a_2	a_2	d_2	d_2	a_4	b_3
$s_t(4)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	b_0	d_0	d_0	b_2	b_2	b_2	b_2	b_2	d_2	d_2
$\Delta_t(2)$	∞	∞	∞	∞	2	1	∞	2	2	∞	∞	1	∞	2	2	∞	1	∞	∞
$y_t(2)$	$\emptyset$	$\emptyset$	$\emptyset$	a	b	b	b	a	a	a	d	b	b	d	d	d	a	a	b

Figure 4: LFU; ties are broken by $a < b < c < d$. The subscripts count the number of earlier references, and underscores mark prior hits at level 3. Also shown are the stack distances $\Delta_t(2)$ at level 2 and the lowest priority lines $y_t(2) \in B_{t-1}(2)$ at level 2.

By convention, the priority of $\emptyset$, the empty line marker, is $-\infty$. As before, we assume $s_0(i) \equiv \emptyset$ for $i = 1, \ldots, C$. It follows from the analysis of [16] that a stack policy is induced by the rule stating that the line of least priority is selected for replacement. We refer to line $y_t(C)$ as a *replacee*, and define $y_t(C)$ to be the least priority line in $B_{t-1}(C)$ if $B_{t-1}(C)$ is full; i.e., $|B_{t-1}(C)| = C$. If $B_{t-1}(C)$ is not full then we let $y_t(C) = \emptyset$. In studying our notation it is important to remember that $y_t(C)$ refers to a line with a particular property in the cache after reference x_{t-1}, not after x_t. This convention follows Mattson et al. [16].

A simple recurrence determines the level i cache contents. Given two cached lines α and β, let $\text{maxp}\{\alpha, \beta\}$ select the one with higher priority. For all $t > 0$, $s_t(1) = x_t$. For all $i > 1$ and $t > 0$,

$$s_t(i) = \begin{cases} s_{t-1}(i) & \text{if } x_t \text{ is a prior hit at level } i \\ y_t(i-1) & \text{if } x_t \text{ is a prior miss at level } i \text{ and } s_{t-1}(i) = x_t \\ \text{maxp}\{y_t(i-1), s_{t-1}(i)\} & \text{otherwise} \end{cases} . \tag{3}$$

Notice that the only lines that ever enter level i are the least priority lines $y_t(i-1)$ from lower levels.

5.1 Parallel Simulation Level by Level

In this section, we present a rapid parallel simulation algorithm for any reference-based replacement policy. For any given $C > 0$, the objective is to compute the level C stack distances $\Delta_1(C), \ldots, \Delta_N(C)$. The algorithm works level by level, like the LRU simulation algorithm presented in Section 4. Specifically, we compute the cache contents $s_t(i)$, the replacees $y_t(i)$ and the stack distances $\Delta_t(i)$ at level i, given these same quantities at level $i-1$. As in the LRU simulation, just $O(1)$ space per reference is needed, with the results of the level i computation overwriting those for level $i-1$.

Level 1 is easy: $s_t(1) \equiv x_t$, $y_t(1) \equiv s_{t-1}(1)$, $\Delta_t(1) = 1$ if $s_t(1) = s_{t-1}(1)$, and $\Delta_t(1) = \infty$ otherwise. Two facts, which follow from equation (3), are crucial to our approach for computing the desired results for level $i > 1$:

- If a new line, say α, enters level $i > 1$ at time t (meaning $s_t(i) = \alpha$, $s_{t-1}(i) \neq \alpha$) then x_t must be a prior miss at level i and α must be the replacee $y_t(i-1)$.

- Assuming $\alpha = y_t(i-1)$ does enter level $i > 1$, it propagates until coming to the first reference x_u, $u > t$, where either $x_u = \alpha$ or x_u is a prior miss and $P(y_t(i-1)) < P(y_u(i-1))$. That is, $s_t(i) = ... = s_{u-1}(i) = \alpha$. If $u < N+1$ then replacee $y_u(i-1)$ enters level i at time u. If there is no such $u < N+1$ then the replacee $y_t(i-1)$ propagates on through time N. For every reference x_t let $u(t)$ denote the index so identified.

Consider the graph where the vertices are the indices t of all prior misses x_t and there is an edge from t to $u(t)$. This edge records the fact that if $y_t(i-1)$ enters level i then $s_t(i) = ... = s_{u(t)-1}(i) = y_t(i-1)$, whereupon it is replaced by $y_{u(t)}(i-1)$ (if $u(t) < N+1$). Observe that not all such $y_t(i-1)$ actually do enter level i—for instance, in Figure 4, $y_{10}(2) = d$ does not enter level 3, because $P(y_{10}(2)) < P(s_9(3))$. However, the set of references that do actually enter level i can be determined by following the maximal path through the graph, starting at vertex 1. Replacee $y_1(i) = \emptyset$ enters at time 1 and propagates until time $v = u(1) - 1$. If $v < N+1$ then replacee $y_{v+1}(i-1)$ enters level i, and propagates until time $w = u(v) - 1$. If $w < N+1$ then replacee $y_{w+1}(i-1)$ enters level i, and so forth.

Converting this serial process for simulating level i into a parallel one, we simulate level $i > 1$ as follows:

1. [Compute tentative propagation intervals.] For each prior miss x_t, compute

$$\begin{aligned} \mathit{stop}(t) &= \text{least } s > t \text{ such that } P(x_s) > P(x_t), \text{ and } x_s \text{ is a prior miss at level } i \\ \mathit{next}(t) &= \text{least } s > t \text{ such that } x_s = x_t, \end{aligned} \tag{4}$$

   where, by convention, $\mathit{stop}(t)$ and $\mathit{next}(t)$ equal $N+1$ if the index s above does not exist. Set $p(t) = \min\{\mathit{next}(t), \mathit{stop}(t)\}$. By earlier remarks, if $y_t(i-1)$ enters level i then $s_t(i) = \ldots = s_{p(t)-1}(i) = y_t(i-1)$.

2. [Follow propagation chain.] The pointers $u(t)$ determine the replacees that enter level i, namely, those with indices: 0, $u(0)$, $u(u(0))$, $\ldots$, with the sequence stopping at $v = u(\ldots u(0)\ldots) \neq N+1$, $u(v) = N+1$. Mark these replacees.

3. [Compute level i results.] For each marked replacee $y_t(i-1)$ and each v in the interval $[t, u(t)-1]$, set $s_v(i) = y_t(i-1)$. For every x_t set $y_{t+1}(i)$ to $\text{maxp}\{s_t(i), y_{t+1}(i-1)\}$. Following this, if x_t is a prior miss at level $i-1$ and if $s_{t-1}(i) = x_t$ set $\Delta_t(i) = i$. Set $\Delta_t(i) = \Delta_t(i-1)$ for prior hits.

Computing the $\mathit{stop}(t)$ values is an instance of the closest larger right neighbor (CLRN) problem, discussed in Section 2.2. It can be solved in $\tilde{O}(\log N)$ time using N PEs. A simpler $O(\log^2 N)$ time solution is described below. Computing the $\mathit{next}(t)$ values via sorting is described in Section 3; the time needed is $O(\log N)$ using N PEs. Marking the replacees on the chain of pointers from 0 to $N+1$ is a *pointer jumping* problem [14], which can be solved in $O(\log N)$ time using $N/\log N$ PEs. The final step of updating the level i cache

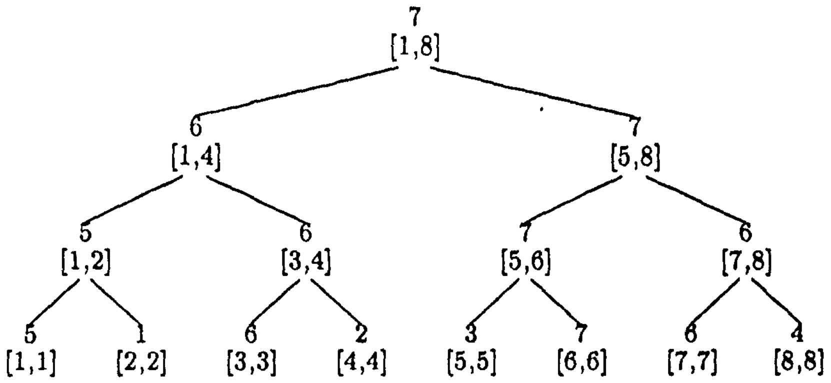

Figure 5: Search tree identifying maximum values over subintervals, which is used to solve the nearest right neighbor problem.

contents and stack distances is essentially the same as was done for LRU simulation in Section 4. We do not repeat the details. The time needed is $O(\log N)$ time using $N/\log N$ PEs. Summing up, the total time is $\tilde{O}(\log N)$ using N PEs. To produce the level C results, the computation must be repeated for each $i = 2$, ..., C. Thus,

Theorem 3 *On the EREW model, given the trace x_1, ..., x_N, the associated level C stack distances $\Delta_1(C)$, ..., $\Delta_N(C)$ induced by any reference-based replacement policy can be computed in time $\tilde{O}(C \log N)$ using N PEs.*

We close this section with a simple method for solving the CLRN problem. This method plays the key role in generalizing the simulation method to accommodate priority aging (Section 5.3).

Let a_1, ..., a_N be a sequence of N numbers, and, for simplicity, assume that N is a power of two and $a_N = +\infty$. For each a_i, $i = 1$, ..., $N-1$, we wish to find the closest right larger neighbor a_j; i.e., $j > i$ is as small as possible and $a_i < a_j$. Construct a binary tree over the inputs as illustrated in Figure 5. Each node is labeled with the maximum value of the inputs in its subtree and with the corresponding subrange of indices. To find the closest larger right neighbor a_j of a_i a two phase search is initiated. Phase one starts at the leaf node a_i, and progresses in steps up the tree. At each step we move from the present node to the nearest internal node at the next higher level whose span includes a node to the right. This phase ends upon visiting (i) an internal node which is rightmost at its level, or (ii) a node whose value is (strictly) greater than a_i. In the example of Figure 5, the first phase for a_3 visits nodes representing ranges $[3,4]$ and $[5,8]$. In general, the first phase stops at a node spanning an interval $[m+2^k, m+2^{k+1}]$, with $i \leq m+2^k$, which must contain the sought a_j. In phase two the search descends down to a_j, at each step moving to the left child if the left child's value exceeds a_i, or to the right child otherwise. On a concurrent read model, carrying out all N searches in parallel gives an $O(\log N)$ time solution. On an exclusive read model, standard methods [14] for resolving the read conflicts add an $O(\log N)$ factor, bringing the total time to $O(\log^2 N)$.

5.2 Random Replacement

The Random Replacement (RR) rule selects the line to replace on a cache miss independently and uniformly at random from the set of cached lines. Mattson et al. [16] observed that the selections can be made so as preserve the stack property ($B_t(i-1) \subset B_t(i)$; for all $t, i \geq 1$), by coupling the random decisions as follows. Suppose the members of $B_t(i-1)$ have been ranked randomly from 1 to $i-1$, with the understanding that the higher a line's rank the lower its priority. To build $B_t(i)$, insert $s_t(i)$ into the priority structure by randomly choosing an integer rank for it from $[1, i]$, say k. Members of $B_t(i-1)$ with ranks $\geq k$ have their ranks incremented by one in $B_t(i-1)$. Other members of $B_t(i-1)$ retain their rank. Thus, for each prior miss x_t at level i, we may decide the line $y_t(i)$ of least rank in $B_t(i)$ by a coin toss: with probability $1/i$, line $y_t(i) = s_{t-1}(i)$, and with the complementary probability $y_t(i) = y_t(i-1)$. Moreover, the outcome is completely independent of $s_{t-1}(i)$.

This independence can be exploited to considerably simplify the simulation of level i over that described above. Step 1 becomes: For each prior miss x_t, compute $next(t)$ as before, and use a coin toss to decide whether to label index t as a "stopper" (probability $1/i$) or leave the index unlabeled (probability $1-1/i$). Let $stop(t)$ be the least stopper $u > t$, or $N+1$ if no such u exists. Let $u(t) = \min\{stop(t), next(t)\}$ as before. Steps 2 and 3 remain the same.

Computing the $stop(t)$ values entails N independent coin tosses and a segmented copy-scan (cf. Section 2.2), operations that net $O(\log N)$ time using N PEs. As a result, we obtain

Theorem 4 *On the EREW model, given the trace* $x_1, \ldots, x_N$, *the associated level* C *stack distances* $\Delta_1(C), \ldots, \Delta_N(C)$ *induced by the RR policy can be computed in time* $O(C \log N)$ *using* N *PEs.*

Comparing with Theorem 3 we see that dropping the CLRN problem and the probabilistic algorithm used to solve it strengthens the running time bound from one that holds with high probability to one that holds deterministically.

5.3 Priority Aging

Under any reference-based replacement rule other than RR, a line's priority is fixed when it enters the cache. If the policy is a practical one, then it is likely that the priority is a simple function of the previous references. For example, under LFU a line's priority is the number of earlier references to the line. Past cache activity gives an imperfect indication of future cache activity. Under LFU a flurry of references to a small set of lines might lead to their long retention during a subsequent period when the lines are not needed. To counter this, it is natural to consider policies that allow a line's priority to *age*; i.e., to decrease monotonically while the line remains unreferenced. In this Section, we extend our reference-based simulation method to accommodate aging.

Let $\phi : \mathbb{R} \rightarrow \mathbb{R}$ be a monotonically decreasing operator, and let ϕ^d represent the d-fold application of ϕ. Let $P_t(\alpha)$ denote the priority of a line α held in the cache at the time of x_t. We consider replacement policies where the initial priority of line x_t is reference-based ($P_t(x_t)$ satisfies R1 of Section 5), but the line's priority "ages" to $P_{t+d}(x_t) = \phi^d(P_t(x_t))$ in the cache $B_{t+d}(C)$ if it remains unreferenced throughout time $t+1, \ldots, t+d$. As before, the replacement policy always selects the line with least priority. Some natural *aging operators* ϕ are $\phi(x) = x - \alpha$ for some fixed $\alpha > 0$ or $\phi(x) = \alpha x$ for some fixed $\alpha \in (0,1)$. We assume that for any $d = 1, \ldots, N-1$, $\phi^d(x)$ can be computed in $O(1)$ time.

Equation (3) describing the evolution of the stack levels continues to hold. A little thought shows that to adapt the simulation method to accommodate aging, we need only change the definition of *stop*(t) in equation (4) to

$$\mathit{stop}(t) = \text{least } s > t \text{ such that } P_s(x_s) > \phi^{s-t}(P_t(x_t)), \text{ and } x_s \text{ is a prior miss at level } i.$$

Computing these new values *stop*(t) can be posed as the following variant of the CLRN problem. Let $a_1, \ldots, a_N$ be a sequence of N numbers, and, for simplicity, assume that N is a power of two and $a_N = +\infty$. For each $i = 1, \ldots, N-1$, we wish to find the smallest $j > i$ such that

$$\phi^{j-i}(a_i) < a_j.$$

We now sketch how to extend the binary search solution given in Section 5 to solve the new problem. Let $\hat{\phi}$ denote the inverse of ϕ. Since $\hat{\phi}$ is monotone increasing, the inequality above implies to

$$\phi^u(a_i) < \hat{\phi}^v(a_j) \quad \text{for all nonnegative integers } u, v \text{ with } u + v = j - i.$$

Letting $[u,v]$ be any range of indices with $u \geq i$,

$$\phi^{j-i}(a_i) < a_j \quad \text{for all } j \in [u,v] \qquad \Leftrightarrow \qquad \phi^{u-i}(a_i) < \max\{a_u, \hat{\phi}(a_{u+1}), \ldots, \hat{\phi}^{v-u}(a_v)\}.$$

This equivalence reveals a way to determine whether a_i ages below some a_j, $j \in [u,v]$, and $i < u$; i.e., whether $\phi^{j-i}(a_i) < a_j$. For $j \geq u$ define $\hat{\phi}^{j-u}(a_j)$ to be the "rejuvenated" value of a_j with respect to index u (i.e., a_j's priority if "de-aged" back to position u). Let $r_{u,v} = \max\{a_u, \hat{\phi}(a_{u+1}), \ldots, \hat{\phi}^{v-u}(a_v)\}$ denote the maximum (over $j \in [u,v]$) rejuvenated value of any a_j with respect to u. To find out if a_i ages below some a_j with $j \in [u,v]$, we may simply compare $\phi^{u-i}(a_i)$ and $r_{u,v}$. Since a_i is arbitrary in this discussion, it is possible to use $r_{u,v}$ concurrently in many searches.

A "rejuvenation-max" tree can be built in parallel using the following observation: for any M (assumed to be a power of 2)

$$\begin{aligned}\max_{1 \leq i \leq M}\{\hat{\phi}^{i-1}(a_i)\} &= \max\{\max_{1 \leq i \leq M/2}\{\hat{\phi}^{i-1}(a_i)\},\ \max\{\max_{M/2+1 \leq i \leq M}\{\hat{\phi}^{i-1}(a_i)\}\} \\ &= \max\{\max_{1 \leq i \leq M/2}\{\hat{\phi}^{i-1}(a_i)\},\ \hat{\phi}^{M/2}(\max_{1 \leq i \leq M/2}\{\hat{\phi}^{i-1}(a_{M/2+i})\})\}\end{aligned}$$

This recursion shows we can build a rejuvenation-max tree over $a_1, \ldots, a_N$ in $\log N$ steps. At every step, all nodes at a given level of the tree are constructed. Leaves are understood to be at level $\log N$, the root is at level 0. A node spanning an interval $[u,v]$ is labeled with $r_{u,v}$. The recusion shows that to compute

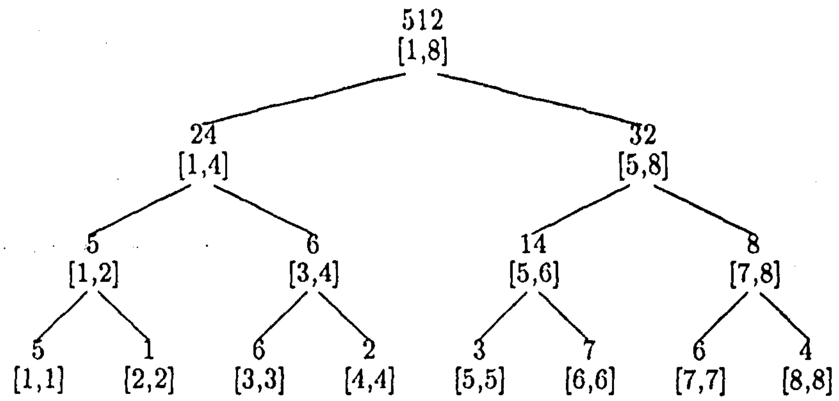

Figure 6: Rejuvenation-max tree, which is used to solve the nearest right neighbor problem when aging is permitted. In this example, the aging operator $\phi(x) = x/2$.

the label of a node at level k, one computes the maximum of (i) the label on the node's left-child, and (ii) the label on the node's right-child promoted by an operator $\hat{\phi}^c$, with $c = 2^{\log N - k}$. Thus, the parent of a left node with value a and right node with value b has label $\max\{a, \hat{\phi}^c(b)\}$. Figure 6 illustrates how the tree of Figure 5 is modified to accommodate the aging operator $\phi(x) = x/2$.

Given a_i we wish to find the closest a_j to the right such that $\phi^{j-i}(a_i) < a_j$. The same two phase strategy described in Section 5 works, replacing the comparison of the value a_i with the label of a node spanning an interval $[u, v]$ with the comparison of $\phi^{u-i}(a_i)$ with $r_{u,v}$, or an equivalent comparison. For example, consider a_3's search for the setup of Figure 5. In the first phase, the search moves up and to the right in the tree, looking over successively larger intervals for a value that a_3 ages below. First, we compare $a_3 = 6$ with $r_{3,4} = 6$, and since a_3 is not smaller continue the first phase. The next node visited represents $[5, 8]$. We compare $\phi^2(a_3) = 1.5$ with $r_{5,8} = 32$, and as a_3 is smaller, phase one stops at this node. In the second phase, the search moves down from $[5, 8]$ to locate the leftmost a_j in $[5, 8]$ that a_3 ages below. First, we branch left to $[5, 6]$, because $r_{5,6} = 14$ is larger than $\phi^2(a_3) = 1.5$. Second, we branch left again to $[5, 5]$ because $r_{5,5} = 3 > \phi^2(a_3) = 1.5$. Since $[5, 5]$ is a leaf, the search stops, having located the right match, a_5, for a_3.

Building the rejuvenation-max tree costs $O(\log N)$ time using N PEs. On the CREW model, we may assign one processor to the search for each input, and so obtain an $O(\log N)$ time solution using N PEs. This in turn implies an $O(\log^2 N)$ time solution using N PEs on the EREW model. The final result is:

Theorem 5 *On the EREW model, given the trace $x_1, \ldots, x_N$, the level C stack distances $\Delta_1(C), \ldots, \Delta_N(C)$ induced by any reference-based policy with aging can be computed in time $O(C \log^2 N)$ using N PEs.*

6 Summary

Trace driven cache simulation is an important tool used in the design of computer systems. Parallel processing offers the promise of reducing the time required to execute a cache simulation, and hence reduce the overall cache design time. We have shown how massively parallel SIMD architectures can be applied to this important problem area.

We note that the bottleneck problem in our simulation of reference-based replacement policies is the closest larger right neighbor problem. An improvement in the solution of this problem to $O(\log N)$ time (without using probabilistic methods) and N PEs on the EREW model appears possible [7]. This would improve the reference-based simulation method to deterministic $O(C \log N)$ time using N PEs.

A number of important issues remain. There is a class of "clock-based" stack algorithms which do not appear to fit within our framework. The classical clock algorithm [6] associates one bit with each physical line in the cache. The bit is set whenever a new line is written into the physical location. A clock counter determines replacement lines. On a hit the counter is untouched, but on a miss, the counter scans the set for a clear bit; the first clear bit found identifies the replacement line. The scan begins where the counter was last left, and any set bit encountered in the scan is cleared. Thus, at most one scan of the set is needed to find a clear bit. The clock algorithm is induced if we assign priority d to line r if d lines must be scanned by the clock before choosing r as the replacement. A line's priority ages as it sits in the cache, but in a highly state-dependent way. One object of our future research is to determine whether clock-based stack algorithms can be simulated in parallel.

We believe that the geometric methods used to obtain the fast, set size independent LRU simulation method of Section 3 might yield similar simulation methods for OPT and for general reference-based policies. Another important issue is whether these techniques can be extended to the simulation of multiprocessor caches. Yet another issue is the use of SIMD processors to generate synthetic cache traces. The method discussed in [20] is basically LRU "in reverse": given the stack distances, compute the reference string. We believe we can implement this method in poly-log time using ideas similar to those developed here. Given the promise of SIMD trace-driven simulation, a more comprehensive study of parallelized synthetic trace generation will be useful.

Acknowledgments

We are grateful to Sandeep Sen, who pointed out the reduction of the CLRN problem to trapezoidal decomposition mentioned in Section 2.2. We are indebted to Subhas Roy for implementing the LRU simulation algorithm of Section 4 on the MasPar MP-1. We also thank Philip Gibbons for several helpful discussions.

References

[1] M.J. Atallah, R. Cole, and M. Goodrich. Cascading divide-and-conquer. *SIAM Journal on Computing*, 18(3):499–532, June 1989.

[2] K.E. Batcher. Sorting networks and their applications. In *AFIPS 1968 Spring Joint Computer Conference*, pages 307–314, Atlantic City,NJ, May 1968. AFIPS Press; Montvale, NJ.

[3] J.L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23:214–219, 1980.

[4] T. Blank. The MasPar MP-1 architecture. In *Compcon Spring 1990*, San Francisco, CA, February 1990. IEEE Computer Society Press.

[5] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[6] R.A. Finkel. *An Operating Systems VADE MECUM.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[7] Philip Gibbons, 1992. personal communication.

[8] A. Gottlieb and C.P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *Journal of the ACM*, 31:193–209, 1984.

[9] P. Heidelberger and H. Stone. Parallel trace-driven cache simulation by time partitioning. In *1990 Winter Simulation Conference*, pages 734–737, New Orleans, LA, December 1990. IEEE.

[10] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Palo Alto, CA, 1990.

[11] W.D. Hillis and Jr. G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[12] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10), October 1985.

[13] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[14] F.T. Leighton. *An Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman, San Mateo, CA, 1992.

[15] Y-B. Lin, J.-L. Baer, and E.D. Lazowska. Tailoring a parallel trace-driven simulation technique to specific multiprocessor cache coherence protocols. In *Distributed Simulation 1989*, volume 21, pages 185–190. The Society for Computer Simulation, 1989.

[16] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 12(2):78–117, 1970.

[17] C.G. Plaxton. Load balancing, selection and sorting on the hypercube. In *1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, Santa Fe, New Mexico, June 1989.

[18] J.H. Reif and S. Sen. Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications. In *1991 ACM Symposium on Parallel Algorithms and Architectures*, pages 327–337, Crete Greece, July 1991.

[19] H. Stone. *High Performance Computer Architecture.* Addison-Wesley, Reading, MA, second edition, 1990.

[20] D. Thiebáubt, H. Stone, , and J. Wolf. Synthetic traces for trace-driven simulation of cache memories. Technical Report RC 14268, IBM Research Division, December 1988.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

[illegible]

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
November 1991

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
MASSIVELY PARALLEL ALGORITHMS FOR TRACE-DRIVEN CACHE SIMULATIONS

5. FUNDING NUMBERS
C NAS1-18605
WU 505-90-52-01

6. AUTHOR(S)
David M. Nicol
Albert G. Greenberg
Boris D. Lubachevsky

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 91-83

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING MONITORING AGENCY REPORT NUMBER
NASA CR-189571
ICASE Report No. 91-83

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report

Submitted to IEEE Trans. on Parallel and Distributed Systems (Journal)

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Trace-driven cache simulation is central to computer design. A trace is a very long sequence, $x_1, ...,$ x_N, of references to lines (contiguous locations) from main memory. At the t^{th} instant, reference x_t is hashed into a *set* of cache locations, the contents of which are then compared with x_t. If at the t^{th} instant x_t is not present in the cache, then it is said to be a *miss*, and is loaded into the cache set, possibly forcing the replacement of some other memory line, and making x_t present for the $(t+1)^{st}$ instant. The problem of parallel simulation of a subtrace of N references directed to a C line cache set is considered, with the aim of determining which references are misses and related statistics.

A simulation method is presented for the Least-Recently-Used (LRU) policy, which regardless of the set size C runs in time $O(\log N)$ using N processors on the exclu:ive read, exclusive write (EREW) parallel mo lel. A simpler LRU simulation algorithm is given that runs in $O(C \log N)$ time using $N/\log N$ processors. We present timings of the second algorithm's implementation on the MasPar MP-1, a machine with 16384 processors. A broad class of *reference-based* line replacement policies are considered, which includes LRU as well as the Least-Frequently-Used and Random replacement policies. A simulation method is presented for any such policy that on any trace of length N directed to a C line set runs in time $O(C \log N)$ time with high probability using N processors on the EREW model. The algorithms are simple, have very little space overhead, and are well-suited for SIMD implementation.

14. SUBJECT TERMS
parallel algorithms; caches; trace-driven simulation; SIMD; memory management; least-recently-used; stack policies

15. NUMBER OF PAGES
22

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

NASA-Langley, 1991