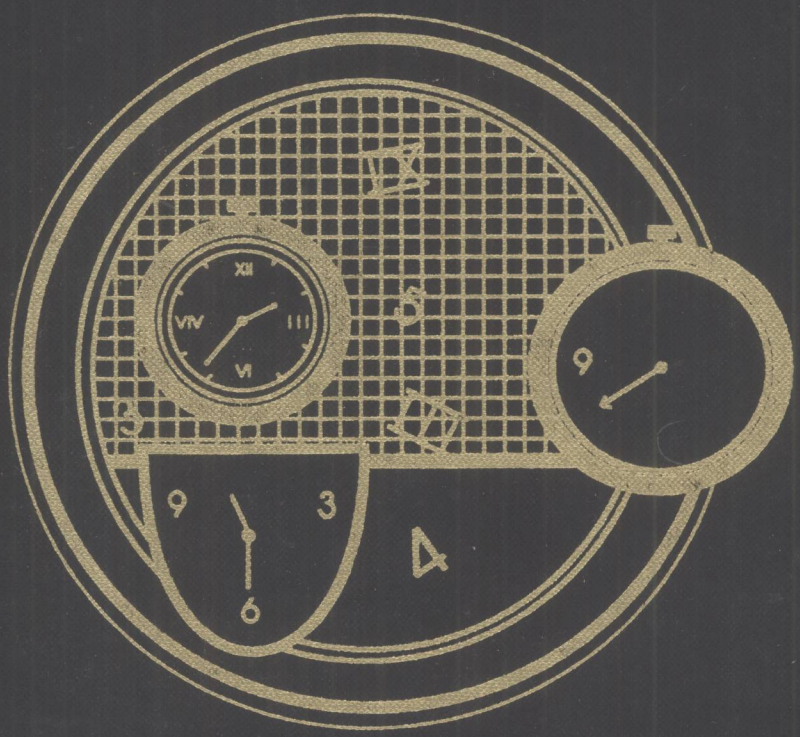
6TH WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION (PADS92)

Edited by
Marc Abrams
and
Paul F. Reynolds, Jr.



Simulation Series
Volume 24
Number 3

SCS A PUBLICATION OF THE SOCIETY FOR COMPUTER SIMULATION

6th Workshop on Parallel and Distributed Simulation (PADS92)

Proceedings of the 1992 SCS Western Simulation MultiConference
on Parallel and Distributed Simulation
20-22 January 1992
Newport Beach, California

Edited by
Marc Abrams
Virginia Polytechnic Institute and State University
and
Paul F. Reynolds, Jr.
University of Virginia

Simulation Series
Volume 24
Number 3

Sponsored by:
The Society for Computer Simulation
Institute of Electrical and Electronics Engineers
Association of Computing Machinery

Additional copies of the Proceedings are available from:

The Society for Computer Simulation
P. O. Box 17900
San Diego, CA 92177 U.S.A.

ISBN 1-56555-008-0

PRINTED IN THE UNITED STATES

6th Workshop on Parallel and Distributed Simulation (PADS92)

Titles in the *Simulation Series*

Vol. 1 No. 1	*Mathematical Models of Public Systems*	George A. Bekey, Editor	January 1971
Vol. 1 No. 2	*Systems and Simulation in the Service of Society*	David D. Sworder, Editor	July 1971
Vol. 2 No. 1	*The Mathematics of Large-Scale Simulation*	Paul Brock, Editor	June 1972
Vol. 2 No. 2	*Recent Developments in Urban Gaming*	Philip D. Patterson, Editor	December 1972
Vol. 3 No. 1	*Computer Simulation in Design Applications*	Sail Ashour and Marvin M. Johnson, Editors	June 1973
Vol. 3 No. 2	*Simulation Systems for Manufacturing Industries*	Marvin M. Johnson and Said Ashour, Editors	December 1973
Vol. 4 No. 1	*Annotated Bibliographies of Simulation*	Tuncer I. Oren, Editor	June 1974
Vol. 4 No. 2	*Spanning the Applications of Simulation*	Paul Brock, Editor	December 1974
Vol. 5 No. 1	*New Directions in the Analysis of Ecological Systems: Part 1*	George S. Innis, Editor	June 1975
Vol. 5 No. 2	*New Directions in the Analysis of Ecological Systems: Part 2*	George S. Innis, Editor	December 1975
Vol. 6 No. 1	*Toward Real-Time Simulation (Languages, Models and Systems), Part 1*	Roy E. Crosbie and John L. Hay, Editors	June 1976
Vol. 6 No. 2	*Toward Real-Time Simulation (Languages, Models and Systems), Part 2*	Roy E. Crosbie and John L. Hay, Editors	December 1976
Vol. 7 No. 1	*An Overview of Simulation in Highway Transportation: Part 1*	James E. Bernard, Editor	June 1977
Vol. 7 No. 2	*An Overview of Simulation in Highway Transportation: Part 2*	James E. Bernard, Editor	December 1977
Vol. 8 No. 1	*Simulation in Energy Systems: Part 1*	Kenneth E. F. Watt, Editor	June 1978
Vol. 8 No. 2	*Simulation of Energy Systems: Part 2*	Kenneth E. F. Watt, Editor	December 1978
Vol. 9 No. 1	*Simulation in Business Planning and Decision Making*	Thomas H. Naylor, Editor	July 1981
Vol. 9 No. 2	*Simulating the Environmental Impact of a Large Hydroelectric Project*	Normand Therien, Editor	July 1981
Vol. 10 No. 1	*Survey of the Apllication of Simulation to Health Care*	Stephen D. Roberts and William L. England, Editors	December 1981
Vol. 10 No. 2	*Computer Modeling and Simulation: Principles of Good Practice*	John McLeod, Editor	June 1982
Vol. 11 No. 1	*Peripheral Array Processors*	Walter J. Karplus, Editor	October 1982
Vol. 11 No. 2	*Computer Simulation in Emergency Planning*	John M. Carroll, Editor	January 1983
Vol. 12 No. 1	*Lumped-Parameter Models of Hydrocarbon Reservoirs*	Ellis A. Monash, Editor	March 1983
Vol. 12 No. 2	*Computer Models for Production and Inventory Control*	Haluk Bekiroglu, Editor	January 1984
Vol. 13 No. 1	*Aerospace Simulation*	Monte Ung, Editor	February 1984
Vol. 13 No. 2	*Simulation in Strongly Typed Languages: ADA, PASCAL, SIMULA...*	Ray Bryant and Brian W. Unger, Editors	February 1984
Vol. 14 No. 1	*All About Simulators, 1984*	Vince Amico and A. Ben Clymer, Editors	April 1984
Vol. 14 No. 2	*Peripheral Array Processors*	Walter J. Karplus, Editor	October 1984
Vol. 15 No. 1	*Emergency Planning*	John M. Carroll, Editor	January 1985
Vol. 15 No. 2	*Distributed Simulation 1985*	Paul F. Reynolds, Editor	January 1985
Vol. 16 No. 1	*Simulators*	John S. Gardenier, Editor	March 1985
Vol. 16 No. 2	*Aerospace Simulation II*	Monte Ung, Editor	January 1986
Vol. 17 No. 1	*Intelligent Simulation Environments*	Paul A. Luker and Heimo H. Adelsberger, Editors	January 1986
Vol. 17 No. 2	*Simulators III*	Bruce T. Fairchild, Editor	March 1986
Vol. 18 No. 1	*AI Applied to Simulation*	E. J. H. Kerckhoffs, G. C. Vansteenkiste and B. P. Zeigler, Editors	February 1986
Vol. 18 No. 2	*Multiprocessors and Array Processors*	Walter J. Karplus, Editor	April 1987
Vol. 18 No. 3	*Simulation and AI*	Paul Luker and Graham Birtwistle, Editors	July 1987
Vol. 18 No. 4	*Simulators IV*	Bruce T. Fairchild, Editor	October 1987
Vol. 19 No. 1	*Methodology and Validation*	Osman Balci, Editor	January 1988
Vol. 19 No. 2	*Aerospace Simulation III*	Monte Ung, Editor	April 1988
Vol. 19 No. 3	*Distributed Simulation, 1988*	Brian Unger and David Jefferson, Editors	July 1988
Vol. 19 No. 4	*Simulators V*	A. Ben Clymer and G. Vince Amico, Editors	October 1988
Vol. 20 No. 1	*AI Papers, 1988*	Ranjeet J. Uttamsingh, Editor	January 1989
Vol. 20 No. 2	*Simulation in Emergency Management and Technology*	Jim Sullivan and Ross T. Newkirk, Editors	April 1988
Vol. 20 No. 3	*Simulation and AI, 1989*	Wade Webster, Editor	April 1988
Vol. 20 No. 4	*Advances in AI and Simulation*	Ranjeet Uttamsingh and A. Martin Wildberger, Editors	March 1989
Vol. 21 No. 1	*Multiprocessors and Array Processors V*	Walter J. Karplus, Editor	March 1989
Vol. 21 No. 2	*Distributed Simulation, 1989*	Brian Unger and Richard Fujimoto, Editors	March 1989
Vol. 21 No. 3	*Simulators VI*	Ariel Sharon and Mohammad R. Fakory, Editors	March 1989
Vol. 21 No. 4	*Simulation in Business and Management*	Sal Belardo and Jay Weinroth, Editors	January 1990
Vol. 22 No. 1	*Distributed Simulation*	David Nicol, Editor	January 1990
Vol. 22 No. 2	*Simulators VII*	Ariel Sharon and M. R. Fakory, Editors	April 1990
Vol. 22 No. 3	*AI and Simulation: Theory and Applications*	Wade Webster and Ranjeet Uttamsingh, Editors	April 1990
Vol. 22 No. 4	*Simulation in Energy Systems*	W. Frisch, B. Cordier, and A. Höld, Editors	October 1990
Vol. 23 No. 1	*Advances in Parallel and Distributed Simulation*	Vijay Madisetti, David Nicol, and Richard Fujimoto, Editors	January 1991
Vol. 23 No. 2	*Simulation in Business and Management II*	Jay Weinroth and Joe Hilber, Editors	January 1991
Vol. 23 No. 3	*Object-Oriented Simulation, 1991*	Raimund K. Ege, Editor	January 1991
Vol. 23 No. 4	*Artificial Intelligence and Simulation*	Ranjeet J. Uttamsingh and A. Martin Wildberger, Editors	April 1991
Vol. 24 No. 1	*Simulators VIII*	Ariel Sharon, Editor	April 1991
Vol. 24 No. 2	*International Conference on Simulation in Engineering Education*	Hamid Vakilzadian, Editor	January 1992
Vol. 24 No. 3	*6th Workshop on Parallel and Distributed Simulation (PADS92)*	Marc Abrams and Paul F. Reynolds, Jr., Editors	January 1992

TABLE OF CONTENTS

TABLE OF CONTENTS

PREFACE

This is the sixth proceedings of a series of workshops on techniques to execute discrete-event simulation programs on parallel and distributed systems. Past proceedings are entitled *Distributed Simulation 1985*, *Distributed Simulation 1988*, *Distributed Simulation 1989*, *Distributed Simulation* (1990) and *Advances in Parallel and Distributed Simulation (PADS)* (1991). The fact that the workshop continues after seven years implies a healthy and continuing interest in the topic.

To continue the role of PADS as the workshop which publishes the best new work in parallel and distributed simulation, all papers and abstracts were subjected to a review process, with the authors' names and affiliations deleted, as announced in the Call for Papers. Each paper was sent to four and in some cases five referees. Referees were carefully selected for each paper using two criteria: to include individuals highly familiar with the subject material and to balance conflicting schools of thought (e.g., advocates of conservative versus optimistic simulation). Forty one papers and three poster abstracts were submitted. Of these, eighteen full papers (41%) from four countries were selected for presentation and twelve were invited as poster abstracts. I'd like to thank members of the program committee, who each reviewed four or more papers, and the remaining reviewers. Several individuals provided outstanding quality reviews, with multiple pages of detailed comments for authors.

A trend over the history of this workshop series has been to increase the proportion of papers reporting results from completed systems applied to realistic simulation problems, rather than accepting papers containing paper designs, preliminary results, and toy benchmarks. After reading about 150 review reports that reinforce this trend in their rankings and comments, I can say that the parallel simulation community has set for itself a high standard. This is good news, because the papers appearing in PADS ultimately are more likely to be applied to problems encountered by simulation practitioners.

Marc Abrams
Program Chair

6th Workshop on Parallel and Distributed Simulation (PADS)

20-22 January 1992
Newport Beach, California, USA

PROGRAM COMMITTEE:

Paul F. Reynolds, Jr. (University of Virginia), General Chair
Marc Abrams (Virginia Polytechnic Institute and State University), Program Chair
Jon R. Agre (Rockwell Institute)
Rassul Ayani (Royal Institute of Technology)
Rajive L. Bagrodia (UCLA)
Doug DeGroot (Texas Instruments)
Robert E. Felderman (USC/Information Sciences Institute)
Richard M. Fujimoto (Georgia Institute of Technology)
David Jefferson (UCLA)
Ron Kriz (Virginia Polytechnic Institute and State University)
Yi-Bing Lin (Bellcore)
Boris Lubachevsky (Bell Labs)
Vijay Madisetti (Georgia Institute of Technology)
David M. Nicol (College of William and Mary)
Peter L. Reiher (Jet Propulsion Laboratory)
Lisa M. Sokol (MRJ, Inc.)
David B. Wagner (University of Colorado at Boulder)

REFEREES:

H. H. Ammar
S. Arthur
O. Balci
M. Bailey
W. Bain
H. Bauer
S. Bellenot
A. Bloss
J. Briner
P. Brown
R. Chamberlain
C. Chandrasekar
D. Cheriton
M. J. Chung
A. Concepcion
B. Cota
N. Davis
S. Dhar
P. Dickens
N. Doraswamy
M. R. Eskicioglu
T. Fahriger
K. Ganugapati
A. Gupta
P. Heidelberger
D. Kafura
P. Konas
R. Kriz
K. Landry
G. Lomow
W. Loucks
A. Maloney
J. McAffer
H. Mehl
S. Midkiff
I. Mitrani
C. Nevison
M. Overstreet
E. Page
C. Pancerello
A. Prakash
B. Preiss
C. Ribbens
C. Ryan
D. Richardson
V. Sanjeevan
R. Schantz
J. S. Steinman
L. Soule
R. Subramanian
P. Tinker
D. Towsley
C. Tropper
H. Wu
G. Zhang

PROTOCOLS AND HARDWARE

MASSIVELY PARALLEL ALGORITHMS FOR TRACE-DRIVEN CACHE SIMULATION

David M. Nicol[1]* Albert G. Greenberg[2] Boris D. Lubachevsky[2]
Subhas Roy[1]
[1] College of William and Mary [2]AT & T Bell Laboratories
Williamsburg, VA 23185 Murray Hill, NJ 07974

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. The cache processes the trace serially; each 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 simulating this serial process on a massively parallel computer is considered, with the aim of determining which references are misses and related statistics. We show that a subtrace of N references directed to a C line set, updated using the Least-Recently-Used policy, can be simulated in $O(C \log N)$ time using N processors on an exclusive read, exclusive write (EREW) parallel model. We present timings of this algorithm's implementation on the MasPar MP-1, a machine with 2^{16} processors. We also consider a broad class of *reference-based* replacement policies, which includes, for example, the Least-Frequently-Used replacement policies. We show that a subtrace of N references targeted to a C line set, updated via one of these policies, can be simulated in $O(C \log N)$ time with high probability on an EREW model. The algorithms are simple, require just $O(1)$ space per reference, and are well-suited for SIMD implementation.

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 [4], 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 [4], 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 [15].

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 $C = 4$. Set size is typically small because the cost of the hardware needed to support fast associative search within a set grows rapidly with the set size,

*The work of this author was supported in part by NASA grants NAG-1-060 and NAS-1-18605, in part by NSF Grant ASC 8819373, and was initiated during a visit to AT&T Bell Laboratories.

and the experience has been that increasing the set size quickly leads to diminishing returns. 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 [3] 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 would provide a more powerful, more flexible solution.

We consider the problem of determining the misses in a given reference trace, x_1 , ..., x_N, directed to a set of size C. In Section 2, we show that if the replacement policy is Least-Recently-Used, the simulation can be carried out in $O(C \log N)$ time using $N/\log N$ PEs on an EREW (exclusive read, exclusive write) model. We report timings of this algorithm's performance on a MasPar [1] SIMD computer having 2^{16} PEs.

We also consider a class of *reference-based* replacement policies (defined below), which 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. Although we do not discuss it here, a minor modification to the algorithm we present allows us to simulate Random replacement. This is important, as random replacement is easy to implement and 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[4].

In Section 3, we sketch how to simulate a trace of N references targeted to a C line set, updated via a reference-based policy, in $O(C \log N)$ time with high probability using N PEs on an EREW model.

Our algorithms are simple, require just $O(1)$ space per reference, and break the computation down into calls to a few primitive parallel subroutines for both data access and computation, such as parallel prefix computations [7] (also known as scans [2]) and sorting. As a result the algorithms are well-suited for SIMD architectures, such as the Connection Machine [5] or MasPar [1]. The $O(C \log N)$ with high probability bound holds because we have assumed that fast probabilistic parallel methods are used for trapezoidal decomposition [13] and sorting using just $O(1)$ extra space per reference. (cf. [9, 12, 13, 14] and references therein). In practice, simpler, asymptotically slower methods may do better. Adopting the notation of [13], these algorithms run 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 $m > 1$.

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 may be effective

[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.

in practice, where a large block of consecutive references would be loaded in the local memory of each PE. Our algorithms generalize immediately to that setup, by simply replacing the underlying parallel primitives with efficient supersaturated counterparts (cf. [6, 11]). 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 all cache sizes 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 [3] presented a parallel simulation method for LRU. Their algorithm is intended for a network of P MIMD processors, and has complexity $O((N \log C/P) + C \log P)$ time and $O(C)$ space per processor. Lin, Baer, and Lazowska have considered parallelizing cache simulations, in the context of multiprocessor cache protocols[8]. 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[10]. 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/O 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 [16] recently proposed a simple method for random generation of realistic traces.

2 LRU

Let us consider the simulation of a single set cache, controlled using the LRU replacement policy, on a sequence of line references $x_1, x_2, \ldots, x_N$. The problem is to compute, for each reference x_t, whether x_t is a hit or a miss.

A little notation will help here and in succeeding sections. Treating the set 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 $x_t \in B_t(C)$. 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 the member of the cache $y_t(C) \in B_{t-1}(C)$ that was referenced least recently. LRU belongs to the important class of *stack replacement policies* [10], defined by the property that increasing cache size cannot increase cache misses:

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

Hence, we can order the lines in the cache by the set size necessary for their appearance. Define the i^{th} element of $B_t(C)$ for $t = 1, 2, \ldots,$ by

$$s_t(i) = \begin{cases} B_t(i) & \text{if } i = 1 \\ B_t(i) - B_t(i-1) & \text{if } |B_t(i)| = i \geq 1 \\ \emptyset & \text{otherwise} \end{cases}, \qquad (2)$$

where $\emptyset$ is the empty line marker. It is convenient to let "reference" x_0 be the empty line, so that $s_0(i) = \emptyset$ for all $i = 1, \ldots, C$, and $s_0(1) = y_1(1) = \emptyset$. Under the LRU replacement policy, we may think of the cache as a stack. The most recent reference is always stored at the top ($x_t = s_t(1)$), and the replaced line is always the last, $y_t(C) = s_{t-1}(C)$.

Figure 1 depicts an example of a trace of 18 references to 4 lines labeled a–d. Beneath the trace, the rows are separated into 4 blocks, showing the contents of the cache for set sizes $C = 1$, 2, 3, and 4; the t^{th} column shows the cache contents immediately following reference x_t. Consider the block for $C = 3$. The first three references, a, c, b, all miss and at time $t = 3$ are found in levels 3, 2, and 1 of the cache, respectively, having pushed the empty line markers out of the cache. At $t = 4$, reference c is a hit, moving c to level 1, pushing b down to level

Reference Index		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$C=1$	trace	a	c	b	c	$\underline{c}$	a	c	a	d	b	$\underline{b}$	d	c	d	a	$\underline{a}$	b	a
$C=2$	Level 1	a	c	b	c	$\underline{c}$	a	c	a	d	b	$\underline{b}$	d	c	d	a	$\underline{a}$	b	a
	Level 2	$\emptyset$	a	c	b	b	c	a	c	a	d	d	b	d	c	d	d	a	b
	Level 1	a	c	b	c	c	a	c	a	d	b	b	d	c	d	a	a	b	a
$C=3$	Level 2	$\emptyset$	a	c	$\underline{b}$	$\underline{b}$	c	$\underline{a}$	$\underline{c}$	a	d	$\underline{d}$	$\underline{b}$	d	$\underline{c}$	d	$\underline{d}$	a	$\underline{b}$
	Level 3	$\emptyset$	$\emptyset$	a	a	a	b	b	b	c	a	a	a	b	b	c	c	d	d
	Level 1	a	c	b	c	c	a	c	a	d	b	b	d	c	d	a	a	b	a
$C=4$	Level 2	$\emptyset$	a	c	b	b	c	a	c	a	d	d	b	d	c	d	d	a	b
	Level 3	$\emptyset$	$\emptyset$	a	$\underline{a}$	$\underline{a}$	$\underline{b}$	$\underline{b}$	$\underline{b}$	c	a	$\underline{a}$	$\underline{a}$	b	$\underline{b}$	c	$\underline{c}$	d	$\underline{d}$
	Level 4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	b	c	c	c	a	a	b	b	c	c

Figure 1: Example of the LRU replacement rule; $\emptyset$ is the empty line marker. The underscores indicate prior hits, as discussed in the text.

2, and leaving a in place at level 3. The underscores in the Figure show a pattern of *prior hits*, defined as follows. A reference x_t is said to be a prior hit at level i if it is a hit when the cache size is $i-1$. In Figure 1, in the block for cache size C, we have marked the prior hits x_t at level C by underscoring the symbol at level $C-1$ in column t. In studying this figure one should remember that an underscore on symbol $s_t(i)$ states 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 symbols 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 $B_7(2)$. A *prior miss* is a reference that is not a prior hit.

Consider $s_t(i)$, the line stored at level i of the cache just after reference x_t is issued. The observation behind our fast simulation is

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

To see this, note that if x_t is a prior miss then the cache update is to shift lines 1, 2, ..., $i-1$ down one position and then put x_t into level one. If x_t is a prior hit then x_t is already stored in a level $< i$, and its move to position one has no effect on level i. In Figure 1, for $C=3$, 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 $t=8$. In addition, note that (i) if x_t is a hit for cache size k, then it is a hit for cache sizes $k+1$, $k+2$, ...,C, and (ii) if x_t is both a hit for cache size k and a miss for cache size $k-1$ then the only possibility is $s_{t-1}(k) = x_t$; i.e., the hit finds x_t at cache level k.

Equation (3) allows us to build the contents of the cache efficiently level by level, along with associated hit/miss information. Specifically, for level i, we compute the two N-vectors,

$$S_i = (s_1(i), s_2(i), \ldots, s_N(i))$$

and

$$H_i = (h_1(i), h_2(i), \ldots, h_N(i))$$

, describing the cache contents at level i and marking the references that result in hits: $h_t(i)$ is $j \le i$ if j is the smallest cache size in which x_t is a hit, and is 0 otherwise. Observe that $h_t(i) \neq 0$ gives x_t's *stack distance* [10]. Thus given vector H_C and any $j \le C$, the number of hits in a j-line cache is the number of elements $h_t(C) \le j$. For any j this number can be computed in $O(\log N)$ time.

To initialize the computation, note that (i) S_1 is just the trace, which is given, and (ii) $h_t(1) = 1$ only if $s_t(1) = s_{t-1}(1)$, so H_1 can be computed in $O(1)$ time. Suppose S_{i-1} and H_{i-1} are given, $i > 1$, and held in vectors $\mathtt{S} = (\emptyset, \mathtt{s_1}, \ldots, \mathtt{s_N})$ and $\mathtt{H} = (\mathtt{h_1}, \ldots, \mathtt{h_N})$, $i > 1$. Similarly, the reference string is held in vector $\mathtt{X} = (\mathtt{x_1}, \ldots, \mathtt{x_N})$. First, we

want to update S to hold S_i. Right shift S into a vector $\mathtt{U} = (\mathtt{u_1}, \ldots, \mathtt{u_N})$ so that $\mathtt{u_t} = \mathtt{s_{t-1}}$. If $\mathtt{h_t} = 0$ (the t^{th} reference is a prior miss) then we want to update $\mathtt{s_t}$ to $\mathtt{u_t}$. Otherwise we want to update $\mathtt{s_t}$ to $\mathtt{u_k}$ where k is the greatest index less than t such that $\mathtt{h_k} = 0$ (the k^{th} reference is a prior hit). Computing these N updates is a *segmented copy scan* [2] where indices t with $\mathtt{h_t} = 0$ mark the segment boundaries. The time needed is $O(\log N)$ using $N/\log N$ PEs. To update H, we note that (i) if $h_t(i-1) \neq 0$ (i.e. $\mathtt{h_t} \neq 0$) then $h_t(i) = h_t(i-1) = \mathtt{h_t} \neq 0$ so that $\mathtt{h_t}$ is left untouched, and (ii) if $h_t(i-1) = 0$ (i.e. $\mathtt{h_t} = 0$) then $\mathtt{h_t}$ should be set to i only if $s_{t-1}(i) = x_t$ (i.e. $\mathtt{u_t} = \mathtt{x_t}$). Thus, updating H is trivially parallelizable, and costs $O(\log N)$ time using $N/\log N$ PEs. Since the updates are repeated $C-1$ times, the total time needed comes to $O(C \log N)$. The space used throughout is $O(1)$ per reference.

In a supersaturated implementation, the cost of a segmented copy-scan is $O(N/P + \log P)$ [6], bringing the overall cost to $O(C(N/P + \log P))$ on P PEs, with a space cost of $O(N/P)$ per PE.

We implemented this algorithm on a MasPar MP-1 computer [1], with 2^{16} processing elements. Each PE is a 4-bit processor with a clock cycle of 80 nanoseconds. A typical integer operation such as those common in our algorithms requires a few tens of clocks. Our implementation supports supersaturation of the PE's, as described earlier. The PE memory size permits us to assign as many as 2048 references to each PE, thereby supporting the simultaneous simulation of a trace with 2^{27} references (over 33 million).

The performance data we present includes only the time spent in the solution phase of the algorithm. Though the traces were generated randomly, the execution times are indicative of what one would experience using actual traces. An industrial-strength 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^{16} to 2^{27}, 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. These timings demonstrate the remarkable promise of massive parallelism for trace-driven cache simulation. Fewer than four and a half seconds of execution time were required to analyze the behavior of a 32-line set on a trace with 33,554,432 references. As a point of comparison, we measured the rate at which an optimized serial LRU simulation processes references. The serial code is written in C, and runs on a Sun Sparc 1++ workstation. After subtracting the time the serial code spends generating a random trace, we find that the MasPar implementation runs approximately 811 times faster. Of course, further experience needs to be gained with the important non-LRU policies, and with I/O problems. But the future of SIMD simulation of caches looks bright.

3 Reference Based Replacement Policies

We now broaden the scope of our methods to provide fast parallel simulations for a large family of replacement policies. Roughly, the class encompasses all

	Trace Length											
C	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}
4	1.4	1.7	2.2	3.2	5.1	9.0	16.9	32.7	64.2	127	253	505
8	2.8	3.4	4.4	6.5	10.6	18.9	35	68	135	268	533	1064
16	5.7	7.0	9.1	13.4	21.8	38.8	72.9	140	277	549	1093	2182
32	11.5	14.2	18.5	27	44	79	147	285	561	1112	2214	4417

Table 1: Execution time of the LRU algorithm on a MasPar MP-1 with 2^{16} PEs, in milliseconds, as a function of trace length and set size.

stack replacement policies where priorities controlling line replacement are static and can be computed efficiently in parallel.

Recall that:

- $B_t(C)$ denotes the cache contents just after x_t is issued and stored, and $y_t(C)$ denotes the line (in $B_{t-1}(C)$) that x_t replaces if x_t is a miss. We refer to $y_t(C)$ as a *replacee*. (By convention, if $B_t(C)$ is not full ($|B_t(C)| < C$) then we define $y_t(C) = \emptyset$, an empty line marker.)
- A stack replacement policy is one where increasing the cache size cannot increase the cache misses; i.e., equation (1) holds.
- Under any stack replacement policy, at any time t, the lines stored in the cache can be ordered by the set size necessary for their appearance. The ordering $s_t(i)$ is given in equation (2); the indices i are called levels.

Mattson et al. [10] show that a stack policy is obtained if a numerical *priority* $P(s_t(i))$ is assigned to each line $s_t(i)$ at time t, and the line $y_{t+1}(C)$ chosen for replacement on loading x_{t+1} is the one with least priority among the members of $B_t(C)$. In the class of *reference-based priorities* policies we 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. These policies are thus characterized as follows.

1. 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 can be established with a sort on the reference tags, followed by a segmented sum-scan.
2. A line's replacement priority does not change except when the line appears in the reference stream.

As before, let $\emptyset$ denote the empty line marker. We assume $s_0(i) = \emptyset$ for all $1 \leq i \leq C$ and, by convention, the priority of an empty line marker is $-\infty$.

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 $u \leq t$. To break ties we can, for example, give higher priority to the reference that has been in the cache the shortest length of time. ($P(x_t) = \text{Count}(x_t, t) - \exp\{-t\}$ would serve that purpose.)
- OPT: $P(x_t)$ is the negation of the smallest index $u > t$ such that $x_u = x_t$.

Figure 2 shows a possible execution of the LFU policy on the same 18 reference trace used to illustrate the LRU policy in Figure 1. As before, for a given cache size C, column t shows the cache contents just after x_t, and indicates that x_t is a prior hit at level C by underscoring the line at level $C-1$ in column t. The lines are arranged in level order. The subscript for each reference gives its LFU priority among all references in the cache. In the case of a tie we give the least-recently-used reference lower priority. Consider, for example, the result of reference $x_9 = d$ if the cache size $C = 2$. This is a miss so a line must be replaced: either a at level 1 or c at level 2. The priority of c (4) is greater than that of a (3), so a is replaced. This illustrates that, unlike LRU, for general reference-based policies the priority order need not coincide with the level ordering.

To simulate a reference-based policy we must compute priorities and then use them to compute hits and misses. Priority computation is simple and cheap for the policies listed above. Indeed, for LRU or OPT, we can dispense with priority computations, as these are implicit in the solution of Section 2. For LFU, a sort brings all references to the same line together in $\tilde{O}(\log N)$ time, and a segmented sum-scan produces the priority counts. Suppose then that the priorities are given. Two facts about reference-based policies which follow from the analysis of [10] lead to our parallel simulation algorithms:

- If a new line, say z_t, enters level i at time t (meaning $s_t(i) = z_t$, $s_{t-1}(i) \neq z_t$) then x_t must be a prior miss at level i. Furthermore, z_t must be identically $y_t(i-1)$. For instance, in the example of Figure 2, for $C = 3$, the prior misses