

**Computer Science Department Technical Report  
University of California  
Los Angeles, CA 90024-1596**

**MULTIPROCESSOR CACHE MEMORIES: SIMULATION  
AND DESIGN**

**Y. Wu**

**November 1993  
CSD-930037**



UNIVERSITY OF CALIFORNIA

Los Angeles

**Multiprocessor Cache Memories:  
Simulation and Design**

A Dissertation Submitted in Partial Satisfaction of  
the Requirements for the degree Doctor of Philosophy  
in Computer Science

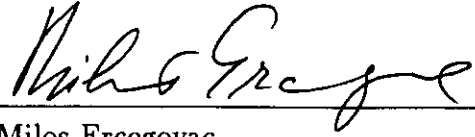
by

Yuguang Wu

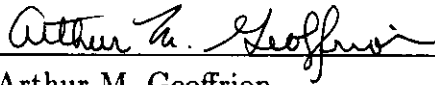
1993

©Copyright by Yuguang Wu, 1993  
All Rights Reserved

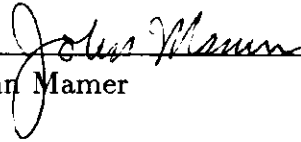
The dissertation of Yuguang Wu is approved.



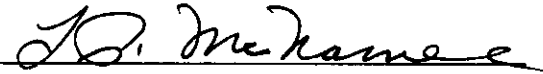
Milos Ercegovac



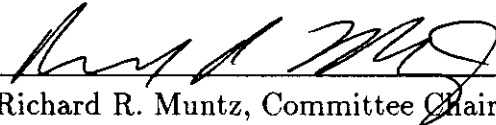
Arthur M. Geoffrion



John Mamer



Lawrence McNamee



Richard R. Muntz, Committee Chair

University of California, Los Angeles

1993

To my parents

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Cache Memories . . . . .	1
1.2	Stack Evaluation . . . . .	5
1.3	Write-Back Caches . . . . .	10
1.4	Multiple Block-Size LRU Caches . . . . .	11
1.5	Set-Associative Caches . . . . .	13
1.5.1	LRU set-associative evaluation . . . . .	16
1.6	Organization of the Thesis . . . . .	17
<b>2</b>	<b>Multiple Block-Size Write-Back Caches</b>	<b>18</b>
2.1	Introduction . . . . .	18
2.2	Informal Description . . . . .	19
2.3	Formal Description . . . . .	22
2.3.1	Correctness Discussion . . . . .	26
2.3.2	Limitation . . . . .	27
2.4	Simulation . . . . .	28
2.4.1	Trace Data . . . . .	28
2.4.2	Implementation . . . . .	29
2.4.3	Results . . . . .	32
2.5	Conclusion . . . . .	36
<b>3</b>	<b>Set-Associative Multiprocessor Caches</b>	<b>39</b>
3.1	Introduction . . . . .	39

3.2	Multiprocessor LRU Set-Associative Evaluation . . . . .	42
3.2.1	Marker Splitting . . . . .	43
3.2.2	Stack Updating . . . . .	46
3.2.3	Stack Distance Counting . . . . .	49
3.2.4	Time Complexity . . . . .	53
3.3	Simulation . . . . .	54
3.3.1	Trace Data . . . . .	54
3.3.2	Implementation . . . . .	55
3.3.3	Simulation Results . . . . .	62
3.4	Summary . . . . .	70
<b>4</b>	<b>All Set-Associative Evaluation</b>	<b>77</b>
4.1	Introduction . . . . .	77
4.2	All Set-Associativity Evaluation and LRU . . . . .	78
4.3	Summary . . . . .	90
<b>5</b>	<b>Multilevel Hierarchies</b>	<b>91</b>
5.1	Introduction . . . . .	91
5.2	Multilevel with Arbitrary Stack Algorithm . . . . .	94
5.2.1	Implementation . . . . .	95
5.2.2	Properties . . . . .	100
5.2.3	Evaluation . . . . .	104
5.3	Summary . . . . .	106
<b>6</b>	<b>Multiprocessor Cache Analysis</b>	<b>107</b>
6.1	Introduction . . . . .	107
6.2	Independent Reference Model . . . . .	110
6.3	Markov Chain Model . . . . .	112
6.4	Bounds . . . . .	115
6.5	Steady State Analysis of LRU Stack . . . . .	118
6.6	Summary . . . . .	122



<b>7</b>	<b>Tree Cache Directories</b>	<b>123</b>
7.1	Introduction . . . . .	124
7.2	Directory Schemes . . . . .	126
7.3	Balanced Binary-Tree Directory . . . . .	131
7.4	Discussion on Other Tree-Like Schemes . . . . .	143
7.5	Summary . . . . .	146
<b>8</b>	<b>Conclusions</b>	<b>147</b>
8.1	Summary . . . . .	147
8.2	Future Work . . . . .	149
8.2.1	Simulation methodology . . . . .	149
8.2.2	Validity of multiprocessor trace simulation . . . . .	150
8.2.3	Parallel stack simulation . . . . .	150
8.2.4	Application paradigms . . . . .	151
	<b>Bibliography</b>	<b>151</b>

# List of Figures

1.1	Memory Address . . . . .	6
1.2	General Stack Updating . . . . .	8
1.3	LRU Stack Updating . . . . .	9
1.4	Block Number . . . . .	14
1.5	Set-Associative Mapping: cache = $SD$ , memory = $SM$ , mapping is mod $S$ . . . . .	15
2.1	Coding Tree . . . . .	31
2.2	Miss ratio v.s. block size on UMIL1, UMIL2, and SPIC. . . . .	33
2.3	Miss ratio v.s. block size on MU10, MUL3, and MUL6. . . . .	34
2.4	Miss ratio v.s. block size on DEC0 and DEC1. . . . .	36
2.5	Miss ratio v.s. block size on ALLC and PASC. . . . .	37
2.6	Miss ratio v.s. block size on FORL, IVEX, DIA0, and LISP. . . . .	38
3.1	Constructing fully associative LRU stack with marker(s) . . . . .	44
3.2	Constructing set-associative LRU stack with marker splitting . . . . .	45
3.3	Counting of $\mu(r), \nu(r)$ . . . . .	52
3.4	Two-level hash table . . . . .	56
3.5	Stack distance calculation . . . . .	61
3.6	Miss ratio v.s. $\alpha$ for a cache of 128 64-byte blocks. . . . .	69
3.7	Optimal $\alpha$ v.s. Cache Size for 2-byte Blocks. . . . .	70
3.8	Optimal $\alpha$ v.s. Cache Size for 4-byte Blocks. . . . .	71
3.9	Optimal $\alpha$ v.s. Cache Size for 8 and 16-byte Blocks. . . . .	72

3.10	Optimal $\alpha$ v.s. Cache Size for 32 and 64-byte Blocks. . . . .	73
3.11	Optimal $\alpha$ v.s. Cache Size for 128 and 256-byte Blocks. . . . .	74
3.12	Optimal $\alpha$ v.s. Cache Size for 512 and 1024-byte Blocks. . . . .	75
3.13	Optimal $\alpha$ v.s. Cache Size for 2048 and 4096-byte Blocks. . . . .	76
5.1	Multilevel Hierarchy Control . . . . .	94
5.2	$M_d$ Free List Updating . . . . .	96
5.3	LRU Hierarchy Control . . . . .	98
5.4	Level Contents v.s. Stack Contents . . . . .	105
6.1	Shared-Memory Multiprocessor with Bus-Interconnection . . . . .	108
7.1	Full-map Directories . . . . .	128
7.2	Limited Directories . . . . .	129
7.3	Chained Directories . . . . .	130
7.4	Balanced Binary-Tree . . . . .	133
7.5	Node Addition: New Level . . . . .	135
7.6	Node Addition: Common Parent . . . . .	135
7.7	Node Addition: Distinct Parents . . . . .	136
7.8	List with Redundant Pointers . . . . .	144
7.9	Perfect Shuffle List . . . . .	145

# List of Tables

2.1	Trace Files . . . . .	29
2.2	Simulation Times . . . . .	35
3.1	Simulation times on FFT (including i/o) . . . . .	64
3.2	Simulation times on FFT (excluding i/o) . . . . .	65
3.3	Times on Simple (inc. i/o) . . . . .	66
3.4	Times on Weather (inc. i/o) . . . . .	67

## ABSTRACT OF THE DISSERTATION

Multiprocessor Cache Memories:  
Simulation and Design

by

Yuguang Wu

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1993

Professor Richard R. Muntz, Chair

This research concentrates on the efficient simulation and design of multiprocessor caches. We extend an efficient cache simulation technique, called stack evaluation, which has mainly been used in simulation of uniprocessor least-recently-used (LRU) caches, to that of multiprocessor LRU caches with any invalidation-based cache coherence protocol. In a one-pass processing of CPU reference traces, this method produces hit ratios for arbitrary cache size and set-associative mapping. Compared to existing techniques, it achieves a magnitude speed-up in simulation time. This simulation technique can also be applied to the evaluation of I/O devices such as shared disks in database systems. We show that the LRU is the only stack algorithm that allows one-pass stack processing for arbitrary set-associative mapping scheme, giving a formal proof to the commonly held but unproven belief. We propose a novel scheme of distributed cache directories for scalable large shared-memory multiprocessor computers. It has overall better space and time complexities than existing directory schemes.

Two results regarding uniprocessor caches are also obtained: we extend the stack simulation technique to LRU write-back caches for multiple block sizes, getting hit ratios and write ratios for arbitrary block sizes in one-pass trace processing; we demonstrate that the efficient stack evaluation techniques for two-level memory hierarchies like cache memories can be applied to certain multilevel memory hierarchies that use a same but arbitrary stack algorithm for replacement on all levels, in which one-pass processing yields hit ratios and write ratios for arbitrary configuration of a hierarchy.

No profit grows where is no pleasure ta'en; In brief, sir, study what you most affect.

— Shakespeare, THE TAMING OF THE SHREW, I.i.39

The whole secret of the study of nature lies in learning how to use one's eyes.

— George Sand, *Nouvelles lettres d'un voyageur* [1869]

# Chapter 1

## Introduction

**Abstract.** This thesis studies multiprocessor cache memories in three aspects: analysis, design, and simulation. In analysis, we use probabilistic models to analyze the performance of shared-memory multiprocessor caches. In design, we propose a new and optimal scheme of distributed cache directories for large and scalable shared-memory multiprocessor computers. In simulation, we extend the efficient stack evaluation techniques in the simulation of uniprocessor and multiprocessor cache memories. This chapter reviews existing work on stack evaluation techniques of cache memories, which are methods for producing performance metrics for a range of parameters with a one-pass evaluation of a CPU reference trace. We will build on these previous results and extend stack evaluation techniques in the following ways: first, it extends the one-pass write-back cache evaluation technique for LRU caches with multiple block sizes; second, it extends the one-pass technique for uniprocessor caches with arbitrary set-associative mapping schemes to multiprocessor LRU caches using an invalidation-based cache coherence protocol. The chapter concludes with an outline of the remainder of the thesis.

### 1.1 Cache Memories

Memory hierarchies are composed of multiple levels of memory. A higher level memory module has shorter response time to a memory access than a lower level module, but generally has less capacity due to its high cost. In general the central

processing unit (CPU) directly accesses data from the top-most memory module. The goal of a memory hierarchy is to provide average access time approximately that of the highest level and a capacity and its cost per byte approximately that of the lowest level. Cache memories can be considered two-level memory hierarchies where the cache is on the first level and the main memory is on the second. Although cache memory will be the focus of this thesis, it is a special case of a more extensive hierarchy, to which some of our work still applies. The usefulness of cache memories is demonstrated by the fact that they have appeared across a wide spectrum of systems, from mainframes to microcomputers[Smith 82].

When the CPU requests a data that is not in the cache, it needs to be brought in from the main memory. To reduce overhead, data transfer between the main memory and the cache is usually carried out in fixed chunks of bytes called *blocks* (or lines). When an absent block has to be moved into a full cache, a currently resident block must be replaced; if the replaced block was written by the CPU (dirty) and the main memory does not have the new data, then this block must be written back to the main memory. The decision as to which block to replace is called the replacement algorithm or policy. Commonly used replacement algorithms include least-recently-used (LRU), least-frequently-used (LFU), and first-in-first-out (FIFO). We study demand-fetching caches, where the cache brings in an absent block from the main memory only when a data in that block is requested by the CPU and the line is not already in the cache.

Efficient and accurate evaluation of cache memory designs plays an important role in practice[Shoemaker 90, Roberts 90, Edenfield 90] and has been an active research topic in the computer systems area[Smith 82]. It is especially crucial in the design of on-chip caches where resources such as physical space are expensive and must be prudently allocated among different processing units[Shoemaker 90]. There are mainly two methods of evaluation: simulation and analytic modeling. Trace-driven simulation is the most widely used evaluation method for performance studies of cache memories. It entails collecting a memory access trace during CPU operation; the collected trace is used to simulate different memory con-



figurations and compare their performance[Mattson 70, Thompson 89, Hill 89, Wang 91, Chaiken 90]. Analytic modeling provides a mathematical (mostly stochastic) model of memory systems, by making some assumptions on the characteristics of the CPU's memory access pattern, and deriving results for the memory system performance[Agarwal 89, Tzelnic 82]. Provided that the input trace is an accurate representation of the traced system, simulation tends to give more precise prediction about the system than analytic modeling. As a practical tool, it can closely model the operation of a real system and provides good approximate measures of performance on various designs. Due to the sheer size of trace data, simulation requires large amounts of storage space and computing time. Analytic modeling, on the other hand, is relatively inexpensive compared with simulation, since the major work is in the analysis. However, to make analysis tractable, one usually has to make simplifying assumptions about the real system under study, hence the results may not be accurate. Modeling is useful in quickly and qualitatively providing an overall description of the entire system's performance.

A few useful metrics are used as criteria in performance evaluation of cache memory designs. One such metric is the hit ratio: the percentage of the number of times that a memory access from the CPU finds the sought-for data in cache. A higher hit ratio means shorter response time to CPU read request, as more data are provided directly from the faster cache without access to the slower memory. Suppose the probability of a hit is  $p_1$ , and response times of the cache and the main memory are respectively  $T_1$  and  $T_2$ , then the expected response time of the cache memory is  $p_1T_1 + (1 - p_1)T_2 = T_2 - p_1(T_2 - T_1)$ . Since usually  $T_1 \ll T_2$ , it is desirable to make  $p_1$  as close to 1 as possible.

Another metric is the write ratio, which is the percentage of the number of writes to memory against the total number of writes issued by the CPU [Thompson 89]. The lower the ratio of memory writes to total writes, the less traffic demand on the interconnection network between the cache and the main memory. If every write issued by the CPU goes to both cache and memory, it

is called a write-through cache. For write-through, the write ratio is 100%. A more interesting cache design is write-back: when a write is issued by the CPU, it is directed to the data block in the cache (requiring a fetch from memory first, if the data block is not already in the cache); the copy of the block in the main memory is unchanged. The dirty data remains in the cache until it is replaced to make room in the cache for newly referenced data. Write-back generally provides quicker response to a CPU write request and requires less interconnect bandwidth than write-through, since it updates the main memory only when a dirty block is replaced from the cache, while the latter updates the main memory on every CPU write. If the dirty block stays in cache long enough to accommodate many writes, then the final update of consecutive bytes of the block in the main memory is overall more efficient[Smith 82, Shoemaker 90]. The downside of write-back is the more complex circuitry of hardware implementation, and it can perform worse than write-through if dirty blocks are flushed after they take only a few writes, when for example a task is switched[Shoemaker 90].

There are several key parameters that affect cache memory performance: cache size, block size, block placement in the cache (also called set-associative scheme, explained later), and as mentioned earlier, write handling (write-through v.s. write-back) and block replacement policy. The bigger the cache size, the higher its hit ratio, and in general thereto the better cache memory performance. However, three constraints have to be considered in practice: cost, physical limitations, and system interdependency[Smith 82]. Even though memories are getting increasingly cheaper, fast cache memories are still expensive. To build an on-chip cache, transistors have to be partitioned among all units including the CPU, hence only a limited number of them can be used for the cache; for example, on the i486 microprocessor the largest cache size that could fit the overall chip design was 8K bytes[Shoemaker 90]. Also, bigger cache requires more complex circuitry and incurs longer delay, demanding a slower system clock rate.

Block size is another parameter. Simulation studies (see [Smith 82]) showed that with the same cache size, moderately big blocks provide better hit ratios

than small blocks. The drawback with big blocks is the larger miss penalties in terms of having to bring in (and take out) more bytes on miss. When the initial latency is high but transfer time is low, then bigger block sizes are appropriate. This depends on the interconnect between the cache and the main memory[Shoemaker 90].

Set-associativity restricts the number of places a block can reside in the cache, reducing cache response time while normally yielding lower hit ratios and has more hardware complexity[Shoemaker 90].

Cache simulation was initially done on a per configuration basis. One simulation was run for each cache configuration to get precise performance result on that cache size, line size, etc. Typically runs were done on a few selected cache sizes, and the whole curve was approximated by extrapolation. The discovery of the stack evaluation (stack processing) technique for demand-fetched two-level memory hierarchies[Mattson 70] has made simulation a much more efficient evaluation approach and extended its applicability. Stack evaluation techniques aim at obtaining cache hit ratios for a range of parameters in one pass over the reference trace.

## 1.2 Stack Evaluation

Let  $X = x_1, x_2, \dots, x_L$  be a memory access trace, where each  $x_t$  for  $1 \leq t \leq L$  is a  $n$ -bit binary address for a byte. The entire address space of  $2^n$  bytes is partitioned into  $2^k$  blocks,  $k$  being a fixed number, as shown in Figure 1.1. Each block has  $2^{n-k}$  addressable bytes. As data movement between the cache and the main memory is carried out in blocks, it is equivalent to study the corresponding block trace  $X^k = x_1^k, x_2^k, \dots, x_L^k$ , with  $x_t^k = x_t/2^{n-k}$ . The superscript is omitted when the context is clear.

A cache block replacement algorithm is called a *stack algorithm* if, when being used, the cache contents in a demand-fetched two-level hierarchy always satisfies an *inclusion* property; namely, the contents of a smaller cache is always a subset

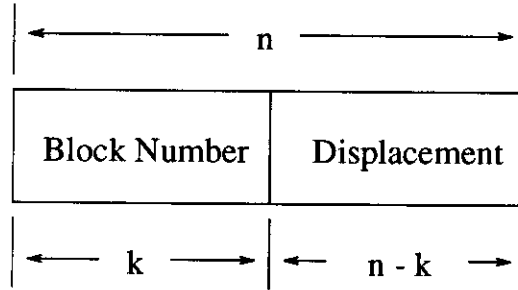


Figure 1.1: Memory Address

of that of a larger cache, for any CPU access sequence. Common replacement algorithms including the LRU, LFU, and the longest forward distance (MIN, the next access is the farthest into the future) are stack algorithms. FIFO is not a stack algorithm; its cache content does not always obey the inclusion property. Stack algorithms permit efficient one-pass evaluation of traces, as we will see below.

There is a more convenient criterion to decide whether a replacement algorithm is a stack algorithm. A *priority-based* replacement algorithm is a replacement algorithm which has, at any time  $t$ , a linear ordering  $P_t$  of previously accessed data blocks  $x_1, x_2, \dots, x_{t-1}$ , called the priority list. A priority-based algorithm makes replacement decisions according to the priority list; if a replacement is required at time  $t$ , among all blocks in the cache, the block with the lowest priority of  $P_t$  is picked. The priority list  $P_t$  must be independent of the cache capacity, although it may change with time  $t$ . It can be shown[Mattson 70] that (a) every priority-based replacement algorithm is a stack algorithm, and (b) every stack algorithm is equivalent to a priority-based algorithm. So a necessary and sufficient condition for a replacement algorithm to be a stack algorithm is that, at any time  $t$ , there is a priority list  $P_t$  of previously accessed data blocks which is not a function of cache size. For example, LRU's priority list is a list of previously accessed data blocks ordered by the decreasing time of their most recent access; LFU's priority list contains previously accessed data blocks ordered by the decreasing frequency of their past usage.

Let  $A$  be a stack algorithm under consideration. Due to the inclusion property, after each access time  $t$  (i.e., the CPU access to  $x_t$  has been fulfilled), the content of any cache can be represented succinctly by a list  $S_t = [s_t(1), \dots, s_t(r_t)]$ , where each  $s_t(i)$  is a distinct block and  $r_t$  is the number of distinct blocks referenced by time  $t$  ( $r_t \leq t$ ). The content of a  $C$ -block cache, after access time  $t$ , is the first  $C$  entries of  $S_t$ :  $[s_t(1), \dots, s_t(C)]$ .  $S_t$  is called the *stack* of  $A$ .

Let  $x_t$  be the block accessed at time  $t$ . The *stack distance*  $\Delta_t$  is the position of block  $x_t$  in the stack  $S_{t-1}$ , i.e.,  $x_t = s_{t-1}(\Delta_t)$ .  $\Delta_t$  is set to  $\infty$  if  $x_t$  is not in  $S_{t-1}$ . An access to  $x_t$  is a hit for a cache of size  $C$  if and only if  $\Delta_t \leq C$ . The percentage of stack distances that are less than or equal to  $C$  for the whole trace  $X$  is the hit ratio for a cache of size  $C$  with reference string  $X$ . Let  $n(d)$  be a counter for the number of times that the referenced block is at stack distance  $d$  in the entire trace. Then the total number of times an accessed block is found in cache with a capacity of  $C$  block frames is

$$N(C) = \sum_{d=1}^C n(d)$$

and the hit ratio is

$$P_{hr}(C) = N(C)/L$$

After each access  $x_t$ , the stack  $S_{t-1}$  is updated to  $S_t$  to reflect the change in the contents for all cache sizes. Stack updating, depicted in Figure 1.2, is done using the priority list  $P_t$  as follows: Denote  $s_{t-1}(1)$  by  $y_t(1)$ . First compare  $y_t(1) = s_{t-1}(1)$  with  $s_{t-1}(2)$ ; the one with higher  $P_t$  priority becomes  $s_t(2)$ , the other is denoted by  $y_t(2)$ . Secondly, compare  $y_t(2)$  with  $s_{t-1}(3)$ ; the one with higher  $P_t$  priority becomes  $s_t(3)$ , the other is denoted by  $y_t(3)$ . Generally, compare  $y_t(i)$  with  $s_{t-1}(i+1)$ ; the one with higher  $P_t$  priority becomes  $s_t(i+1)$ , the other is denoted by  $y_t(i+1)$ . This process continues until  $s_{t-1}(d) = x_t$  is found, or the end of the stack  $S_{t-1}$  is reached. If  $x_t$  is found at the  $d$ -th entry  $s_{t-1}(d)$  of  $S_{t-1}$ ,  $y_t(d-1)$  becomes  $s_t(d)$ , and  $s_{t-1}(j)$  becomes  $s_t(j)$  for all  $j > d$ . In both cases,  $x_t$  becomes  $s_t(1)$ .

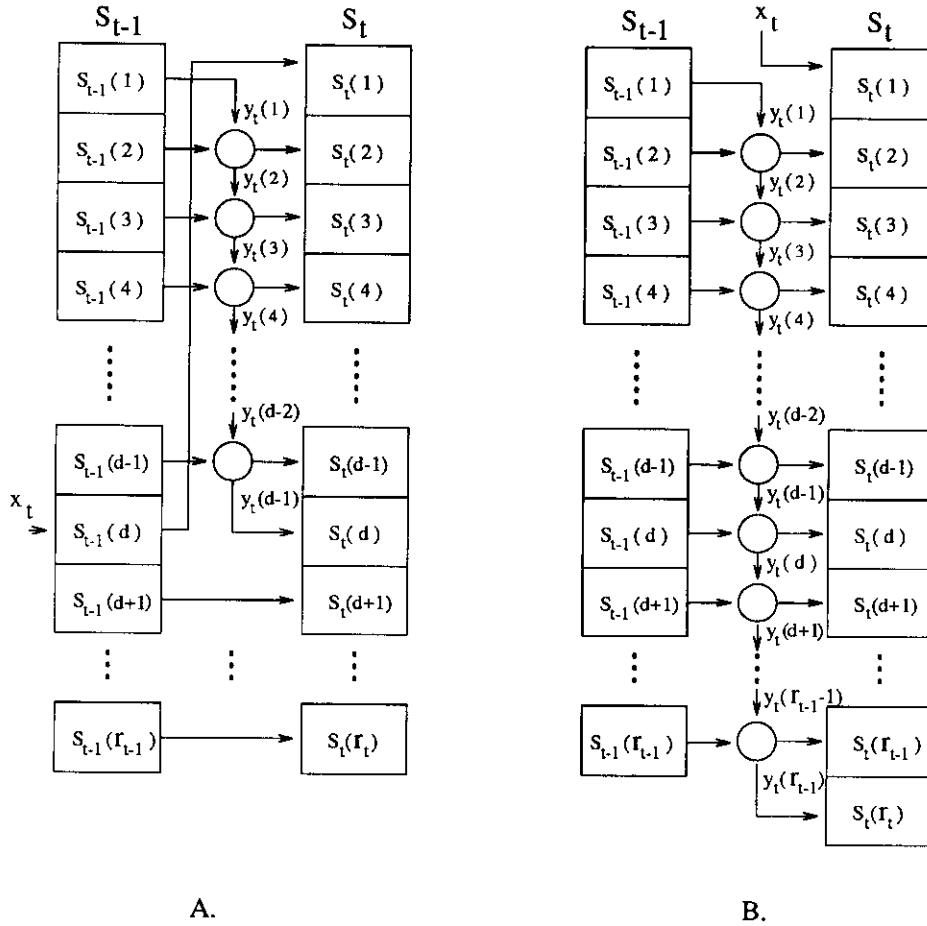


Figure 1.2: General Stack Updating

A.  $x_t$  is in  $S_{t-1}$ ; B.  $x_t$  is not in  $S_{t-1}$ .

Stack updating for LRU replacement is quite simple: if  $x_t$  is found in the stack, then  $x_t$  is pulled to the top of stack while entries previously above it are shifted down by one position; if  $x_t$  is absent, then  $x_t$  is appended to the stack head, as depicted in Figure 1.3. This is because the priority of each block is equal to its stack position.

Many extensions to the original stack evaluation techniques have been made by various researchers. Traiger and Slutz extended stack evaluation to multiple block sizes and multilevel hierarchies for LRU algorithm [Traiger 71]. Muntz and Opderbeck extended it to two-level directly addressable memory hierarchies for

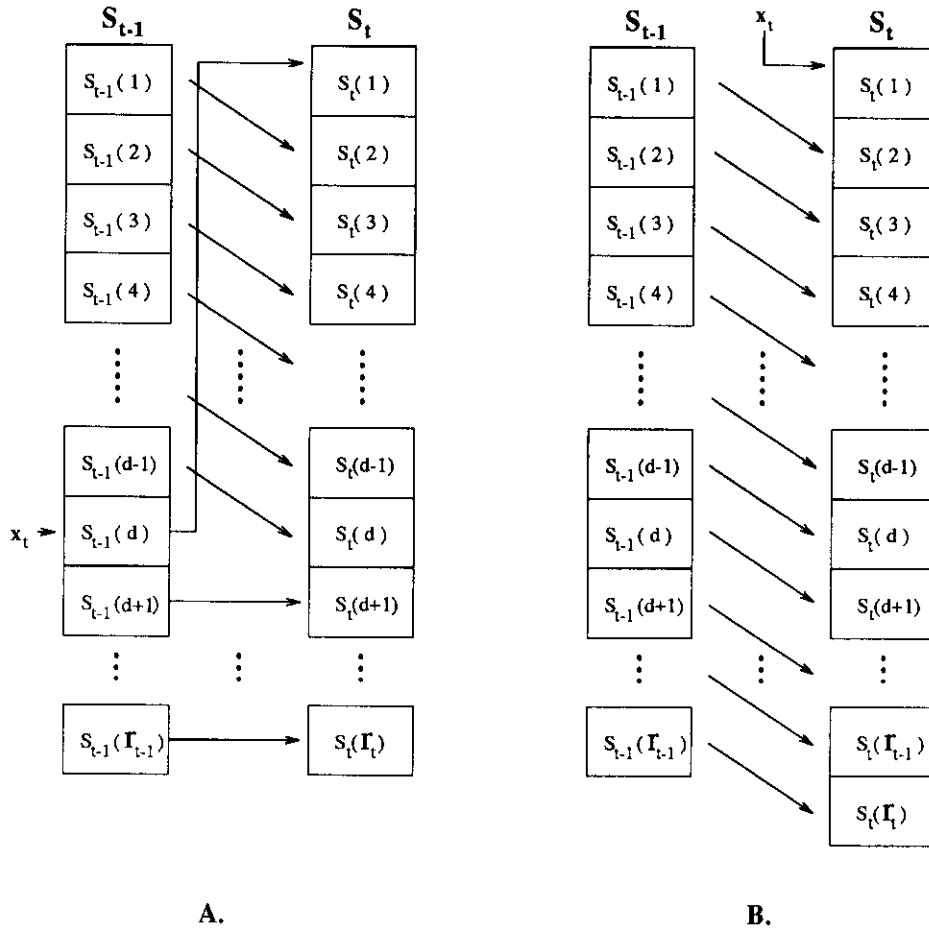


Figure 1.3: LRU Stack Updating

A.  $x_t$  is in  $S_{t-1}$ ; B.  $x_t$  is not in  $S_{t-1}$ .

an arbitrary stack algorithm where a memory access can by-pass the first level cache to go directly to the second storage[Muntz 74]. Gecsei made extensions to more general multilevel hierarchies where the top-most level has an arbitrary stack replacement algorithm and every lower level has its replacement algorithm dependent upon the immediately higher level replacement algorithm[Gecsei 74]. Thompson and Smith extended stack evaluation of a general stack algorithm to write-back caches and sector caches[Thompson 87, Thompson 89]. Wang and Baer extended write-back techniques to arbitrary set-associative caches for the LRU algorithm[Wang 89, Wang 91].

In the following sections we review some of these extensions. Some of the results reported in this thesis are related to or based on this previous work. Later chapters will assume familiarity with these sections.

### 1.3 Write-Back Caches

Initial work on stack evaluation techniques ignored write operations. It did not distinguish between reads & writes or equivalently, assumed all accesses were reads. In the case of cache memories, when the CPU writes to some memory address, the data can be written to cache only, or to both cache and memory. The former, termed write-back, provides better response time to the CPU, as writes to memory are done in asynchronous fashion and write requests usually only take cache delays. The dirty data is not written to the main memory until its block is replaced from the cache, at which time the whole dirty block is written out to the main memory. Evaluation of write-back caches requires study of the number of actual write operations made (in blocks) from the cache to the main memory. Thompson and Smith[Thompson 89] found a clever way to evaluate write-back cache memories within the general framework of stack evaluation. They noticed that if a dirty data block is written again while it is still in the cache, then the write-back of the previous write is avoided. In other words, if the cache size is greater than or equal to the longest stack distance, called *dirty level*, that a dirty data block has achieved since it was last written, and now the CPU issues another write to that block, then there will not be any write-back for the previous write. The number of write-backs is inversely proportional to the cache size: the bigger the cache, the fewer the number of write-backs. If a block is dirty in a smaller cache, it has not been replaced since last time it was written; by the definition of stack algorithm, it must also have resided in a larger cache since it was last written, i.e., it must also be dirty in a larger cache.

In the Thompson & Smith algorithm, each block is augmented with a dirty level value  $dl$ . A block never written has its dirty level set to infinity. When a



block is accessed, its dirty level is updated by taking the maximum of its current stack distance and its previous dirty level. If the current access is write, then a counter  $ws(dl)$  is incremented, and the block's  $dl$  is reset to zero. For a cache with  $C$  block frames, the number of writes (to the main memory) saved by is

$$S(C) = \sum_{d=1}^C ws(d)$$

If there are  $W$  ( $\leq L$ ) writes out of the  $L$  accesses, the write ratio is given by

$$P_{wr}(C) = 1 - S(C)/W$$

fraction of writes that actually cause memory writes.

## 1.4 Multiple Block-Size LRU Caches

The block size of  $2^{n-k}$ -byte has so far been considered a constant for any single simulation run. From the system designers' point of view, there are up to  $n + 1$  ways to partition the  $2^n$ -byte address space into blocks of sizes ranging from  $2^0$  to  $2^n$ . Assume that  $k$  is between  $v$  and  $w$ ,  $0 \leq v \leq w \leq n$ . When the block size  $2^{n-k}$ -byte is changed with  $k$ , one normally has to repeat stack evaluation on the block trace  $X^k = x_1^k, x_2^k, \dots, x_L^k$ . For LRU replacement, however, it is possible to do stack evaluation on the minimum-sized block stack  $S_t^w$  using the corresponding block trace  $X^w = x_1^w, x_2^w, \dots, x_L^w$  and obtain stack distances  $\Delta_t^k$  for all the  $k$ 's in  $v \leq k \leq w$  at once [Traiger 71].

Let  $S_t^a$  and  $S_t^b$  be the LRU stacks for blocks of  $2^{n-a}$  byte and  $2^{n-b}$  byte, respectively, after accessing byte  $x_t$ .  $S_t^a$  is the list of  $2^{n-a}$ -byte blocks ordered by their most recent access, while  $S_t^b$  that of  $2^{n-b}$ -byte blocks. If  $2^{n-a}$  evenly divides  $2^{n-b}$ , each  $2^{n-b}$ -byte block contains integral number of  $2^{n-a}$ -byte blocks; an access to (some byte in) some block of  $2^{n-a}$  bytes is also an access to a *unique* block of  $2^{n-b}$  bytes. Hence  $S_t^b$  can be obtained from  $S_t^a$  by orderly taking, from stack head to stack tail, its entries ( $2^{n-a}$ -byte blocks) which represent distinct  $2^{n-b}$ -byte blocks; i.e.,  $S_t^b$  is embedded in  $S_t^a$ . By the block partition scheme in Figure

1.1, the division of block sizes is  $2^{n-k}/2^{n-w} = 2^{w-k}$ , a  $2^{n-w}$ -byte block evenly divides a  $2^{n-k}$ -byte block. Any LRU stack  $S_t^k$  of  $2^{n-k}$ -byte blocks is embedded in stack  $S_t^w$ .  $S_t^k$  could be constructed from  $S_t^w$  by choosing, going from head to tail of the stack, all entries in  $S_t^w$  that have distinct  $k$ -bit prefixes, i.e., throwing out all lower entries in the stack that match the  $k$ -bit prefix of a higher stack entry.

To find the stack distance  $\Delta_t^k$  for  $2^{n-k}$ -byte blocks, we need to count, in stack  $S_{t-1}^w$ , the number of entries with distinct  $k$ -bit prefixes, from the stack head down to the first entry whose  $k$ -bit prefix equals  $x_t^k = x_t^w/2^{w-k}$ . To decide whether an entry has a distinct  $k$ -bit prefix, define a *left match function*  $LM(x, y)$  between two block addresses  $x$  and  $y$  as the number of consecutive high order bits that match. For example,  $LM(0111, 0100) = 2$ . Blocks  $x$  and  $y$  have the same  $k$ -bit prefix if and only if  $LM(x, y) \geq k$ . For each entry  $s_{t-1}^w(j)$  in  $S_{t-1}^w$  attach a variable  $MLM_j$ , which is the maximum value of the left match functions between  $s_{t-1}^w(j)$  and all entries  $s_{t-1}^w(i)$  above it ( $1 \leq i < j$ )

$$MLM_j = \begin{cases} 0 & j = 1 \\ \max_{1 \leq i < j} [LM(s_{t-1}^w(j), s_{t-1}^w(i))] & j > 1 \end{cases}$$

For each  $k$  between  $v$  and  $w$ , entry  $s_{t-1}^w(j)$  has a distinct  $k$ -bit prefix (i.e., no entry above it has the same  $k$ -bit prefix) if and only if  $MLM_j < k$ . As stack  $S_{t-1}^w$  is being searched for  $x_t^w$ , the stack distance variable  $\Delta_t^k$  is incremented at each entry  $s_{t-1}^w(j)$  if and only if  $MLM_j < k$ .

When an entry  $s_{t-1}^w(d)$  with  $s_{t-1}^w(d)/2^{w-k} = x_t^w/2^{w-k}$  is encountered in the stack,  $\Delta_t^k$  will not be incremented further, which means  $x_t^k$  is found. Another variable  $LIM$  is used to remember the minimum value of  $k$  such that  $x_t^k$  has not been found in the stack.  $LIM$  is initialized to  $v$ , and is set to  $\max[LM(x_t^w, s_{t-1}^w(j)) + 1, LIM]$  at every stack entry  $s_{t-1}^w(j)$  that is being searched. Stack searching stops when  $x_t^w$  is found or  $S_{t-1}^w$  is exhausted.

There is a simple way to count the stack distance  $\Delta_t^k$ . Let  $\beta(r)$  be a set of counters initialized to zero before processing each access  $x_t$ , and let  $n^k(\Delta^k)$ , for each  $k$ , be a counter for the number of times that stack distance  $\Delta^k$  is found

among the entire trace for  $2^{n-k}$ -byte blocks. When searching the  $j$ th stack entry, counter  $\beta(\max[LIM, MLM_j + 1])$  is incremented by one. At the end of stack searching for  $x_t$ , the stack distance is expressed by

$$\Delta_t^k = \sum_{r=v}^k \beta(r) \quad (1.1)$$

for  $v \leq k < LIM$ , and  $\Delta_t^k = \infty$  for  $LIM \leq k \leq w$ . After each access  $x_t$ , the stack distance counter  $n^k(\Delta_t^k)$  is incremented by one for each block size  $2^{n-k}$ .

When stack  $S_{t-1}^w$  is updated to  $S_t^w$ , the  $MLM$  values of its entries are updated as well. For entries in  $S_{t-1}^w$  that are below  $x_t^w$ , their  $MLM$ 's remain the same; for each  $j$ -th entry above  $x_t^w$ , after it is searched, its  $MLM$  is changed to

$$MLM_j = \max[MLM_j, LM(x_t^w, s_{t-1}^w(j))]$$

$s_{t-1}^w(j)$  now becomes the  $(j+1)$ -th entry  $s_t^w(j+1)$  in  $S_t^w$ .

Finally, the total number of times an access is found in a cache of  $C$  bytes ( $C/2^{n-k}$  blocks of  $2^{n-k}$  bytes) is

$$N^k(C) = \sum_{d=1}^{C/2^{n-k}} n^k(d)$$

and its hit ratio is

$$P_{hr}^k(C) = N^k(C)/L.$$

## 1.5 Set-Associative Caches

In the cache organizations considered thus far we have assumed that a data block can occupy any block frame in the cache memory; the mapping of data blocks in the cache is unconstrained. This is usually called *fully associative* mapping. This unconstrained mapping has the disadvantage of having to search the entire cache each time a block is to be located, causing slow response to CPU's access request. A constrained mapping, called *set-associative* mapping, is used in practice to reduce the search time, whereby each block is restricted to a subset of the cache block frames.

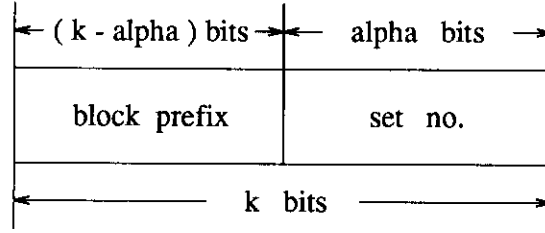


Figure 1.4: Block Number

The most common type of set associative mapping is the two's power congruence mapping, or congruence mapping[Mattson 70]. A congruence mapping is to partition the  $2^k$  blocks into  $2^\alpha$  disjoint sets of equal cardinalities, numbered from 0 to  $2^\alpha - 1$ . Each set has  $2^{k-\alpha}$  blocks. Blocks with equal  $\alpha$  lower-order bits in block address belong to the same set. The set that a block belongs to is determined by the  $\alpha$  lower-order bits of block's address, i.e., by operation mod  $2^\alpha$ , as shown in Figure 1.4. We call  $\alpha$  the *set length*, with  $0 \leq \alpha \leq k$ .

A congruence-mapping set-associative cache usually allocates an equal number of block frames, say  $D$ , for each set. For set length  $\alpha$ , the cache's total capacity is  $C = 2^\alpha \cdot D$  blocks. Such a cache is called  $D$ -way set associative cache; when  $D = 1$  ( $\alpha = \log C$ ), it is called *direct-mapped* cache, where each block has only one possible frame in which it can reside in the cache; see Figure 1.5. When a block  $x$  is accessed, only the  $D$  block frames of its set need to be searched. If it is not there, and if all  $D$  block frames are occupied, a block replacement decision is made to remove one of these  $D$  blocks from the cache.

Since the sets are disjoint, the cache can be treated as a collection of  $2^\alpha$  independent caches, one for each set. The two-level cache-main memory hierarchy can also be viewed as a collection of  $2^\alpha$  independent cache-memory hierarchies, each with a cache size of  $D$  frames. They can be handled separately using stack evaluation techniques.

For a general stack algorithm, stack evaluation must be applied individually to each value of the set length  $\alpha$ . A total of  $k + 1$  passes of trace evaluation are needed for all values of  $\alpha$  between 0 and  $k$ . Alternatively, one can maintain

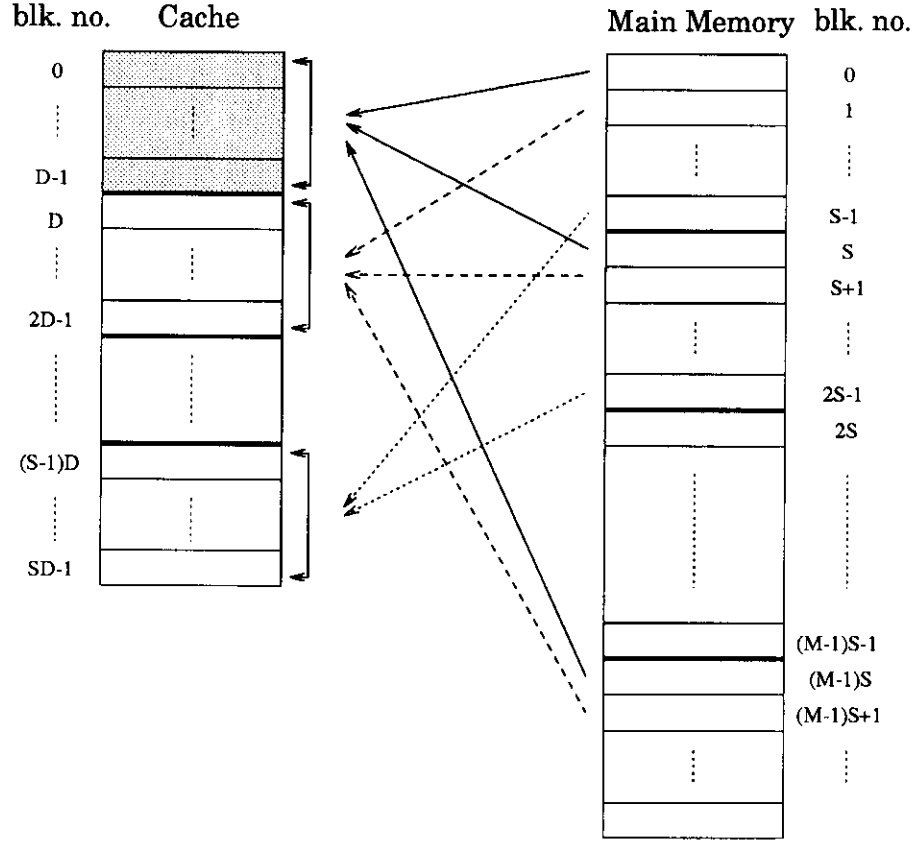


Figure 1.5: Set-Associative Mapping: cache =  $SD$ , memory =  $SM$ , mapping is mod  $S$ .

$k + 1$  groups of stacks, where each group is composed of stacks for a particular set length, and update all of them for each access  $x_t$ . Denote by  $S_{t-1}(i, \alpha)$  the “sub”-stack of the  $i$ th set for set length  $\alpha$ , and  $\Delta_t^\alpha$  the stack distance of  $x_t$  in the sub-stack  $S_{t-1}(x_t \bmod 2^\alpha, \alpha)$ . For each set length  $\alpha$ , we get  $x_t$ 's stack distance  $\Delta_t^\alpha$  from the sub-stack  $S_{t-1}(x_t \bmod 2^\alpha, \alpha)$ , which in turn is updated to  $S_t(x_t \bmod 2^\alpha, \alpha)$  according to the stack algorithms' priority list  $P_t$ . Other sub-stacks are unchanged:  $S_t(j, \alpha) = S_{t-1}(j, \alpha)$ ,  $j \not\equiv x_t \pmod{2^\alpha}$ .

Denote the stack distance counter for distance  $d$  of set length  $\alpha$  by  $n(\alpha, d)$ . For a  $D$ -way set associative cache with a total capacity of  $C = 2^\alpha \cdot D$  blocks, after processing  $L$  accesses, the number of times an access is found in the cache

is

$$N(\alpha, 2^\alpha D) = \sum_{d=1}^D n(\alpha, d)$$

The hit ratio is

$$P_{hr}(\alpha, 2^\alpha D) = N(\alpha, C)/L.$$

### 1.5.1 LRU set-associative evaluation

The LRU algorithm has the nice property that each sub-stack  $S_t(i, \alpha)$  is embedded in the global stack  $S_t$  [Mattson 70].  $S_t(i, \alpha)$  is the list of all previously accessed blocks that are in the  $i$ th set for set length  $\alpha i$ , ordered by most recent reference;  $S_t$  is the list of all previously accessed blocks ordered by most recent reference; the former can be recovered from the latter by picking up all the entries that are in the  $i$ th set for set length  $\alpha$ . Stack searching of  $S_t$  amounts to simultaneous searching of all sub-stacks  $S_t(i, \alpha)$ ,  $0 \leq i \leq 2^\alpha - 1$ ,  $0 \leq \alpha \leq k$ .

Suppose the current referenced block is  $x_t$ , and the  $j$ th entry  $s_{t-1}(j)$  of the global stack  $S_{t-1}$  is being searched. Define the *right match function*  $RM(a, b)$  as the number of consecutive low-order matching bits in  $a, b$ . For example,  $RM(1010, 0110) = 2$ ,  $RM(1000, 1001) = 0$ . Obviously block  $s_{t-1}(j)$  is in the same set with  $x_t$  for some set length  $\alpha$  if and only if  $RM(x_t, s_{t-1}(j)) \geq \alpha$ . Let  $\{\mu(r)\}$  be a set of counters initialized to zero for  $0 \leq r \leq k$  before the processing of each access  $x_t$ . To determine  $\{\Delta_t^\alpha\}$  for all  $\alpha$ , we just search the global stack  $S_{t-1}$  and increment counter  $\mu(RM(x_t, s_{t-1}(j)))$  at each stack entry. Searching stops when  $x_t$  is found, and the new global stack  $S_t$  is obtained by pulling  $x_t$  to the stack head. The stack distance  $\Delta_t^\alpha$  is given as

$$\Delta_t^\alpha = \sum_{r=\alpha}^k \mu(r) \tag{1.2}$$

for  $\alpha = 0, \dots, k$ . If  $x_t$  is not found in  $S_{t-1}$ ,  $\Delta_t^\alpha$  is set to  $\infty$  for every  $\alpha$ . Then the stack distance counter  $n(\alpha, \Delta_t^\alpha)$  is incremented by one, for each  $\alpha = 0, \dots, k$ .

After the entire trace is processed, these counters  $n(\alpha, d)$  ( $0 \leq \alpha \leq k, 0 \leq d \leq \infty$ ) contain the stack distance distributions for each set length  $\alpha$ .

## 1.6 Organization of the Thesis

This research will make new extensions to stack evaluation in the following ways: first, it extends the one-pass write-back techniques to LRU caches with multiple block sizes, getting hit ratios and write ratios for multiple block sizes in one pass; second, it extends the one-pass techniques for uniprocessor caches to multiprocessor LRU caches with any invalidation-based coherence protocol, getting hit ratios for multiple set-associative mappings in one pass; third, we show that the LRU is the only stack algorithm that allows one-pass stack processing for all set-associative mappings, giving a formal proof of a commonly held but unproven belief; fourth, we demonstrate that the efficient evaluation techniques for two-level memory hierarchies (cache memories) can also be applied to certain multilevel memory hierarchies (staging hierarchies) as well, which use a same but arbitrary stack algorithm for block replacement on all levels. In analysis, we present an analytic model for the performance of multiprocessor caches that use an invalidation-based coherence protocol. While the bulk of this thesis is focused on cache memory evaluation through either simulation or analysis, we provide, in the area of cache system design a novel scheme of distributed cache directories for scalable large shared-memory multiprocessor computers; it has provably optimal space and time complexities among all scalable directory schemes.

## Chapter 2

# Multiple Block-Size Write-Back Caches

**Abstract.** There exist one-pass evaluation techniques for read-only LRU cache memories with multiple block-sizes and for write-back LRU caches with a fixed block size. By exploring an useful property of LRU, we extend these techniques to evaluate write-back LRU caches for multiple block-sizes. Using a vector of dirty levels, we give a simple method that produces hit ratios and write ratios of write-back LRU caches for a range of block-sizes in a one-pass trace evaluation.

### 2.1 Introduction

It is known that for read-only memory hierarchies using LRU replacement algorithm, the hit ratios can be evaluated for multiple block-sizes in a single pass over a memory reference trace. It is also known that for write-back caches with a fixed block size, the write ratios can be evaluated in a single pass. We extend these techniques, showing that for write-back cache memories using LRU replacement, the hit ratios and write ratios can both be evaluated for multiple block-sizes in a single pass over the reference trace.

The enormous amount of disk space needed for storing CPU trace data is a



costly problem in trace-driven simulation of cache memories. One way to solve this problem is through *on-the-fly* simulation, which evaluates trace data as they are being collected. Since trace data do not need to be saved anymore, on-the-fly simulation solves the storage problem. As a one-pass algorithm, the method of write-back evaluation for multiple block-sizes is a good candidate for on-the-fly trace-driven simulation.

This chapter presents this new technique. It combines the stack evaluation technique for multiple block-size read-only LRU caches with the technique for fixed block-size write-back caches. Its correctness is proved by a method of loop-invariance, and is empirically verified by simulation results on real trace data.

## 2.2 Informal Description

We use the same framework as in [Traiger 71] for read-only multiple block-size LRU stack evaluation to handle write-backs. For a specific block size of  $2^{n-k}$  bytes, it is easy to calculate its hit ratio and write ratio by using one dirty level variable, since the required stack  $S_t^k$  is implicitly embedded in the minimum block size LRU stack  $S_t^w$ , and the method of [Thompson 89] can be readily applied. To calculate the write ratio for each  $k = v, \dots, w$ , it is natural to use an array of dirty level variables, one for each block size (equivalently, each  $k$ -bit block prefix of the  $n$ -bit address). We call the dirty level variable for  $2^{n-k}$ -byte blocks a *k-bit dirty level*. Together they compose a *dirty level vector* of  $w - v + 1$  elements

Dirty-level vectors have been used for stack simulation of set-associative cache memories in [Wang 91], where evaluation of multiple set-associativities with a fixed block size was studied. The problem there is a bit simpler, where the block size is fixed, and the dirty level variable for each set-associativity is associated with some stack entry. As we will see below, this is not the case for multiple block-sizes, where a dirty level variable has to be moved between stack entries.

Since there may be multiple entries in  $S_t^w$  with identical  $k$ -bit prefixes, there is the question of which will store the  $k$ -th bit dirty level. We will store the dirty

level in the top-most one among these entries. In other words, let  $x$  be a stack  $S_t^w$  entry that is written, then we keep its  $k$ -th bit dirty level on the top-most entry  $s_t(f)$  of stack  $S_t^w$  that has the same  $k$  bit prefix as  $x$ . We denote this top-most stack entry by a special symbol  $[x_t]^k$ :

$$[x_t]^k \stackrel{\text{def}}{=} s_t(f)$$

where

$$f = \min\{ i \mid s_t(i)/2^{w-k} = x/2^{w-k} \}$$

This ensures that, whenever  $x_t$  is found, the dirty level of  $x_t$  for *every* block size has been found. This eliminates the need to search the entire stack for all relevant dirty levels, and conforms to the traditional stack searching method that stops whenever the current  $2^{n-w}$ -byte block number  $x_t$  is found.

Our algorithm works as follows: for each block access  $x_t$ , search the stack  $S_{t-1}^w$  until  $x_t$  is found or the stack end is reached. During the searching,  $k$ -th bit stack distances  $\Delta_t^k$  are accumulated in some auxiliary counters  $\beta(r)$ , as explained in section 1.4. The  $k$ -th bit dirty level of  $x_t$  is collected in the  $k$ -th element  $pdl^k$  of a temporary vector  $pdl = [pdl^v, \dots, pdl^w]$  (*previous dirty levels*). When the searching is done, the current block  $x_t$  is put on the top of the new stack. Stack distances are computed as usual by Equation 1.1 in section 1.4; namely

$$\Delta_t^k = \sum_{r=v}^k \beta(r) \quad (2.1)$$

For each  $k$ ,  $pdl^k$  is updated by

$$pdl^k = \max[pdl^k, \Delta_t^k] \quad (2.2)$$

If the access is a read, the dirty level vector of the top stack entry  $x_t$  is set to  $pdl$ . If the access is a write, write-saved counters  $ws^k(pdl^k)$  are incremented for each  $k$ ; the dirty level vector of  $x_t$  is set to an all-1's vector  $\underbrace{[1, \dots, 1]}_{w-v+1}$ .

This is best explained by an example. Consider a reference trace  $X = 1001, 0000, 1100, 0100, 0010, 0000$ . Here  $v = 1, w = n = 4$ . Accesses  $x_2 = 0000$

and  $x_5 = 0010$  are writes, the rest are reads.  $a_j$  represents the  $j$ -th stack entry.  $MLM_j$  is the max left match value, and  $DL_j = (DL_j^v, DL_j^{v+1}, \dots, DL_j^w)$  is the dirty level vector, respectively, for entry  $a_j$ . The dirty level value of 0 represents infinity. If the value of a dirty level  $DL_j^k$  was changed during the previous access, this is indicated with an underline.

After access  $x_2 = 0000$ , stack  $S_2$  is simply as follows. As it's a write access,  $DL_1$  is all 1's.

$a_j$	$MLM_j$	$DL_j$
0000	0	1111
1001	0	0000

After access  $x_3 = 1100$ , stack  $S_3$  is as follows.

$a_j$	$MLM_j$	$DL_j$
1100	0	0000
0000	0	1111
1001	1	0000

After access  $x_4 = 0100$ , stack  $S_4$  is as follows.  $x_4$  is in the same  $2^{4-k}$ -byte block with  $s_3(2) = 0000$  for  $k = 1$ , so  $pdl^1 = 1$ , the value of the 1st dirty element of of  $s_3(2)$ , which is in turn reset to zero. At the end of current stack searching,  $\Delta^1 = 2$ , so  $pdl^1$  is changed to 2 by Equation (2.2). As this is a read access,  $pdl$  is stored in  $s_4(1)$ ,

$a_j$	$MLM_j$	$DL_j$
0100	0	<u>2</u> 000
1100	0	0000
0000	1	<u>0</u> 111
1001	1	0000

After access  $x_5 = 0010$ , stack  $S_5$  is as follows. Notice that since  $x_5$  is in the same  $2^3$ -byte block 0 with  $s_4(1) = 0100$  and the same  $2^2$ -byte block 00 with  $s_4(3) = 0000$ , so  $pdl^1 = 2, pdl^2 = 1$ . At the end of the current stack searching,  $\Delta^1 = 1, \Delta^2 = 3$ , thereby  $pdl^1 = 2$  and  $pdl^2 = 3$  by Equation (2.2). Since it is

a write access,  $ws^1(pdl^1) = ws^1(2)$  and  $ws^2(pdl^2) = ws^2(3)$  are incremented by one, and  $DL_1$  is set to all 1's.

$a_j$	$MLM_j$	$DL_j$
0010	0	1111
0100	1	<u>0000</u>
1100	0	0000
0000	2	<u>00</u> 11
1001	2	0000

After one more access  $x_6 = 0000$ , stack  $S_6$  is as follows.  $x_6$  is in the same  $2^3$ -byte block 0 and  $2^2$ -byte block 00 with  $s_5(1) = 0010$ , so  $pdl^1 = 1, pdl^2 = 1$ ; it is in the same 1-byte block 0000 with  $s_5(4) = 0000$ , so  $pdl^3 = 1, pdl^4 = 1$ . As  $\Delta^1 = \Delta^2 = 1, \Delta^3 = \Delta^4 = 4$ , By Equation (2.2),  $pdl^1 = pdl^2 = 1, pdl^3 = pdl^4 = 4$ . As it is a read access,  $pdl$  is stored in  $s_6(1) = 0000$ .

$a_j$	$MLM_j$	$DL_j$
0000	0	<u>1144</u>
0010	2	<u>00</u> 11
0100	1	0000
1100	0	0000
1001	2	0000

## 2.3 Formal Description

Here we present the combined procedure for stack distance counting, dirty level computing, and stack updating. It is described in a PASCAL-like pseudo code, where  $x \leftrightarrow y$  denotes switching the values of variables  $x$  and  $y$ . The accessed block is  $x_t$  in minimum block size, and  $w_t = \phi$  if  $x_t$  is a read access. The  $j$ th entry of  $S_{t-1}^w$  is  $a_j$ .  $STACKLEN$  is the length of the current stack.  $LIM$  denotes the minimum unfound prefix  $k$ , or equivalently, the maximum unfound block size of  $2^{n-k}$  bytes.  $MLM_j$  is the maximum left match value for  $a_j$ . In addition, there

is a dirty level vector of  $l = w - v + 1$  elements attached to each stack entry; as for the  $j$ th entry  $a_j$ , it is  $DL_j = (DL_j^v, DL_j^{v+1}, \dots, DL_j^w)$ . Clean blocks in  $S_i^w$  have an all-infinity dirty level vector. When a block is written, its dirty level vector is set to all 1's. Counters  $\beta(r)$  are used to calculate the stack distance  $\Delta^k$  for every  $k$ . Counters  $n^k(r)$  are used to store the distance frequencies of  $\Delta^k$ .  $ws^k(r)$  is the write-saved counter for  $2^{n-k}$ -byte block number.  $L$  is the trace length, and  $W$  is the number of writes of the  $L$  accesses.  $y, z, b^v, \dots, b^w$  are the temporary variables used for entry down-pushing during LRU stack updating.  $pdl^k$  is the  $k$ -th element of temporary dirty level vector  $pdl$  that holds the dirty level of  $[x_t]^k$ , which is the first stack entry that is in the same  $2^{n-k}$ -byte block as  $x_t$ .

Functions  $max(a, b)$  and  $left\_match(a, b)$  return the proper values for  $a$  and  $b$ . At the beginning of stack evaluation, counters  $n^k(d)$  and  $ws^k(d)$  are initialized to zero. Before processing each new access, counters  $\beta(r)$  are set to zero, and  $pdl^k$  and  $b^k$  are set to infinity for each  $k$ .

For each accessed block  $x_t$ , the procedure searches the stack top-down, during which the stack distance for the  $2^{n-k}$ -byte block entry  $[x_t]^k$  is accumulated in  $\beta(r)$ 's, the dirty level for the  $2^{n-k}$ -byte block entry  $[x_t]^k$  is collected in  $pdl^k$ , and meanwhile the stack is updated. Stack searching is completed when either  $x_t$  is found or the stack is exhausted; in either case  $x_t$  is put into the stack's top entry before the next round of searching (for  $x_{t+1}$ ). Each  $pdl^k$  is set to the new stack distance  $\Delta^k$  if it is smaller. If the current access to  $x_t$  is a read,  $pdl$  is stored with  $x_t$  in the top stack entry; if it is a write, then counter  $ws^k(pdl^k)$  is incremented for each  $k$ , and the dirty level vector of the top stack entry for  $x_t$  is set to all 1's.

**Algorithm.** Multiple block-size write-back LRU stack evaluation

```

1  for  $k = v$  to  $w$  do
2      for  $j = 1$  to  $\infty$  do
3           $n^k(j) = ws^k(j) = 0$ ;
4   $STACKLEN = 0$ ;
5  for  $t = 1$  to  $L$  do
6      get  $x_t, w_t$ ;
7       $j = 1$ ;  $y = x_t$ ;  $z = 0$ ;  $LIM = v$ ;  $flag = \text{true}$ ;
8      for  $k = v$  to  $w$  do
9           $\beta(k) = 0$ ;

```

```

10      $pdl^k = \infty$ ;
11      $b^k = \infty$ ;
12     while  $flag$  and  $j \leq STACKLEN$  do
13          $r = \max(LIM, MLM_j + 1)$ ;
14          $\beta(r) = \beta(r) + 1$ ;
15          $LM = left\_match(x_t, a_j)$ ;
16          $MLM_j = \max(MLM_j, LM)$ ;
17         if  $LIM \leq LM$ 
18             then
19                 for  $k = LIM$  to  $LM$  do
20                      $pdl^k = DL_j^k$ ;
21                      $DL_j^k = \infty$ ;
22                      $LIM = LM + 1$ ;
23             if  $x_t = a_j$ 
24                 then
25                      $a_j = y$ ;
26                      $MLM_j = z$ ;
27                     for  $k = v$  to  $w$  do  $DL_j^k = b^k$ ;
28                      $flag = \text{false}$ ;
29             else
30                  $y \leftrightarrow a_j$ ;
31                  $z \leftrightarrow MLM_j$ ;
32                 for  $k = v$  to  $w$  do  $b^k \leftrightarrow DL_j^k$ ;
33              $j = j + 1$ ;
34     if  $flag$ 
35         then
36              $a_j = y$ ;
37              $MLM_j = z$ ;
38             for  $k = v$  to  $w$  do  $DL_j^k = b^k$ ;
39              $STACKLEN = STACKLEN + 1$ ;
40     for  $k = v$  to  $LIM - 1$  do  $\Delta^k = \sum_{r=v}^k \beta(r)$ ;
41     for  $k = LIM$  to  $w$  do  $\Delta^k = \infty$ ;
42     for  $k = v$  to  $w$  do
43          $n^k(\Delta^k) = n^k(\Delta^k) + 1$ ;
44          $pdl^k = \max(pdl^k, \Delta^k)$ ;
45     if  $w_t \neq \phi$ 
46         then
47              $W = W + 1$ ;
48             for  $k = v$  to  $w$  do
49                  $DL_1^k = 1$ ;
50                  $ws^k(pdl^k) = ws^k(pdl^k) + 1$ ;
51     else
52         for  $k = v$  to  $w$  do  $DL_1^k = pdl^k$ ;

```

Lines 1 to 4 and 7 to 11 initialize variables. Line 5 iterates the algorithm for the entire reference trace. Line 6 gets the block address  $x_t = x_t^w$  of the current access.

Lines 12 through 33 are involved with stack searching and stack updating; this block of instructions is exited when either the current accessed block  $x_t$  is found (*flag* is set to false) or the stack is exhausted ( $j$  is greater than *STACKLEN*).

Line 13 sets the value of  $r$  to the least of all block prefixes that need to count the current stack entry,  $a_j$ , toward their stack distances. Line 14 remembers this  $r$  value in the proper counter  $\beta(r)$ .

Line 15 calculates the left-match value  $LM$  between  $x_t$  and  $a_j$ . Line 16 updates the maximum left match value of  $a_j$ .

Lines 17 through 22 deal with the case of  $LIM \leq LM$ , where for block prefixes  $k = LIM, LIM + 1, \dots, LM$ , the sought-for  $2^{n-k}$ -byte block entry  $[x_t]^k$  has just been found. Here their dirty levels  $DL_j^k$  are transferred to temporary variables  $pdl^k$  in line 20 and are reset to  $\infty$  in line 21.  $LIM$  is set to the new value  $LM + 1$  in line 22, since all blocks with block size greater than  $2^{n-LM}$  have been found.

If  $a_j$  is exactly equal to  $x_t$  (lines 23 to 28), then the  $k$ -th bit block entry  $[x_t]^k$  has been found for all  $k = v, v + 1, \dots, w$ . The previous stack entry is placed here by line 25 to 27, and stack searching is terminated with *flag* set to false.

If  $a_j$  is not exactly equal to  $x_t$  (lines 29 to 32), i.e.  $LM < w$ , then  $2^{n-k}$ -byte block entry  $[x_t]^k$  has not been found for  $LIM \leq k \leq w$ , and stack searching will continue. The current stack entry should be shifted down to the next entry, while the previous stack entry being shifted into current entry. Lines 29 to 32 do just that by interchanging  $y, z, b$  with  $a_j, MLM_j, DL_j$ . When the current entry is the top one ( $j = 1$ ),  $y = x_t$  is put in.

Line 33 increments the current stack entry index  $j$ , advancing stack searching on to the next stack entry  $a_{j+1}$ .

When stack searching is terminated but  $x_t$  is not found, a new entry is created, and the last stack entry examined is inserted by lines 34 to 38. Stack size *STACKLEN* is incremented in line 39.

Lines 40 through 52 are for stack distance counting and dirty level computing. Lines 40 and 41 calculate stack distances  $\Delta^k$  for each block prefix  $k$ . For  $v \leq k < LIM$ ,  $\Delta^k$  should be the number of times that  $v \leq r \leq k$  happened in line

13, which is  $\sum_{r=v}^k \beta(r)$ . For  $LIM \leq k \leq w$ ,  $2^{n-k}$ -byte block entry  $[x_t]^k$  is not in stack, hence its stack distance is  $\infty$  in line 41. All stack distances are recorded in their corresponding counters in line 43. If  $pdl^k$  is less than current stack distance  $\Delta^k$ , it is updated in line 44, by Equation (2.2).

If the current access is a write operation (Lines 45 to 50), then for each block prefix  $k$ , the top stack entry's dirty level vector is set to all 1's in line 49, and the number of writes avoided is counted in line 50. The count of total write operations is incremented in line 47. If the current access is a read operation, then the temporary dirty level vector  $pdl$  is stored into the dirty level vector  $DL_1$  of the new top entry in line 52.

### 2.3.1 Correctness Discussion

Our method employs two techniques: (1). using a dirty level for each block prefix; (2). if  $x$  has ever been written, always keep its  $k$ -th bit dirty level on the top-most entry  $[x]^k$  of stack  $S_t^w$  that has the same  $k$ -bit prefix as  $x$ . The need to maintain individual dirty level for each block prefix is due to the fact that there is no definite relation between stack distances (and hence dirty levels) of different block sizes. Consider this reference trace:  $0(w), 2^{w-k}(r), 2 \times 2^{w-k}(r), 3 \times 2^{w-k}(r), \dots, (2^k - 1) \times 2^{w-k}(r), 0(r)$ . Here  $p(r)$  denotes a read access to address  $p$ , and  $p(w)$  a write access to address  $p$ . At the end of the trace, the  $(k-1)$ -bit prefix dirty level and  $k$ -bit prefix dirty level of block  $p = 0$  are  $\Delta^k = 2^k$ ,  $\Delta^{k-1} = 2^{k-1}$ . In this case,  $\Delta^k = 2 \times \Delta^{k-1}$ . Consider another reference trace:  $0(w), 2 \times 2^{w-k}(r), 4 \times 2^{w-k}(r), 6 \times 2^{w-k}(r), \dots, (2^k - 2) \times 2^{w-k}(r), 0(r)$ , the dirty levels of block  $p = 0$  in the end are  $\Delta^k = \Delta^{k-1} = 2^{k-1}$ . Generally,  $\Delta^{k-1} \leq \Delta^k \leq 2 \times \Delta^{k-1}$ . One might try to exploit the monotonic property of stack distances among different block sizes, namely  $\Delta^{k_1} \leq \Delta^{k_2}$  for  $k_1 < k_2$ ; but because of the quite arbitrary disparity between  $\Delta^k$  and  $\Delta^{k-1}$ , it is necessary to keep track of the difference  $\rho^k = \Delta^k - \Delta^{k-1}$  for each  $k$ , which amounts to a vector.

The reasons to keep the dirty level on the top-most relevant stack entry are



two-fold: first, whenever  $x_t$  is found, we can be sure that the dirty levels for every  $2^{n-k}$ -byte block in which  $x_t$  is contained, namely  $x_t/2^{w-k}$  for  $k = v, \dots, w$ , have been obtained, therefore stack searching can be terminated; second and more important, the independence between stack distances as demonstrated above makes it essential to keep it on the top-most relevant stack entry, so that individual dirty levels are correctly maintained with each stack updating.

In Algorithm 1, lines 15, 16, 17 through 22, and 45 through 52, are responsible for storing the  $k$ -th bit dirty level information with  $x_t$ , which is to be put at the top entry of  $S_t^w$  at the end of current stack searching and updating. Doing this in every round of stack searching and updating effectively puts the  $k$ -th bit dirty level information for  $x_t$  on the top-most stack entry  $[x_t]^k$  that belongs to the same  $2^{n-k}$ -byte block as  $x_t$ . From this loop-invariant property, the correctness of the algorithm readily follows.

### 2.3.2 Limitation

This method applies only to fully associative caches. Cache management usually employs set associative caching to improve cache response time. One-pass evaluation of read-only LRU caches for multiple block-sizes and set associativities is possible[Traiger 71], and it can be extended in a similar way to write-back caches. However, now one needs an array, instead of just a vector, of dirty level variables, for each stack entry in the minimum block stack. Such memory consumption by the simulation can be quite overwhelming. We are currently looking into properties of the algorithm similar to those in subsection 2.4.2 that enables one to dynamically allocate dirty level arrays and reduce simulation requirement on memory.

## 2.4 Simulation

To examine its application, we ran simulations using our method on some real trace data. In this section, we will describe first the trace files used, then a simple implementation of the algorithm, and finally the simulation results.

### 2.4.1 Trace Data

The trace data were produced by Agarwal on a VAX-11 architecture for various workloads, using a microcode-based tracing tool called ATUM [Agarwal 86]. They contain both single-task and multiprogramming traces. Each different trace is stored in a separate file.

Both user mode addressing and kernel mode addressing were recorded. All addresses are virtual addresses. The addresses correspond to four-byte words. Each record consists of an operation code and a word address. The operation code indicates whether it is an instruction access or a data access, and whether a data access is read or write.

Table 2.1 summarizes the characteristics of the trace files. Trace files are named NAME.num, where NAME indicates the type of workload traced, and num is the sample number of the trace taken on that workload. For example, the trace file lisp.000 represents the first sample of the (single task) lisp workload, while the trace file mul3.1 the second sample of a workload with three active processes (mul3).

In Table 2.1, the first column is the name of each trace, the second column is the total number of references in each trace (trace size). The third column is the total number of data references, the fourth column is the number of read references, and the fifth column is that of write references. The sixth column is the number of unique data addresses referenced, and the last column is the average number of references to the same data location.

Table 2.1: Trace Files

Trace	T refs	D refs	reads	writes	U refs	repeats
allc	15334	4405	2400	2005	1147	3.84045
dec0.001	334775	164492	99897	64595	6030	27.2789
dec1.001	329613	161818	99342	62476	9297	17.4054
dia0	336093	139203	90819	48384	12425	11.2035
forl.000	314110	158397	100019	58378	16189	9.78424
forl.001	362518	192616	116460	76156	15980	12.0536
ivex.000	307172	127445	97238	30207	31517	4.04369
ivex.003	396775	225408	137022	88386	8178	27.5627
lisp.000	262760	115527	99067	16460	5678	20.3464
lisp.001	261451	115224	98600	16624	6833	16.8629
mu10.0	337353	161356	122398	38958	20670	7.80629
mu10.1	360325	185594	134442	51152	27260	6.80829
mu10.2	372610	196250	139659	56591	20120	9.75398
mul3.0	351089	171803	102505	69298	13388	12.8326
mul3.1	338946	160937	96533	64404	17624	9.1317
mul3.2	373408	198634	109771	88863	13166	15.0869
mul6.0	400698	234962	171440	63522	12812	18.3392
mul6.1	367205	191563	135282	56281	21925	8.73719
mul6.2	394185	225444	149833	75611	20214	11.1529
pasc.001	540567	360547	264969	95578	19256	18.7239
spic.000	358168	208336	136088	72248	7710	27.0215
spic.001	422818	250121	151510	98611	5584	44.7924
umil1	357132	185315	167328	17987	11516	16.092
umil2	359462	196211	182390	13821	2233	87.8688

## 2.4.2 Implementation

We use a simple doubly-linked list to implement the stack  $S^w$  of minimum blocks. Since the algorithm requires update of the  $MLM$  variable of each entry above the currently accessed block in the stack, a linear list is a natural choice. Thompson[Thompson 87] compared various data structures in the implementation of write-back stack evaluation algorithm and found that sophisticated data structures yield very little improvements in simulation time for real CPU traces.

To alleviate memory consumption by the algorithm, we will dynamically allocate dirty level variables to the stack entries, instead of statically allocating an array of dirty level variables to each stack entry. The following properties of

the algorithm were observed during experiments, and the first one is useful in reducing the algorithm's demand on memory.

**Property 2.1** *Each dirty vector has a contiguous subvector of finite elements. That is, for the  $j$ -th stack entry, and  $v \leq i' < i < i'' \leq w$ , if  $DL_j^{i'} < \infty, DL_j^{i''} < \infty$ , then  $DL_j^i < \infty$ .*

**Proof.** Let the minimum-size block address of the  $j$ -th stack entry be  $b$ . For any  $i$  such that  $v \leq i \leq \text{MLM}_j$ , there is a stack entry above that has the same  $i$ -bit prefix as  $b$  (i.e.,  $b/2^{w-i}$ ); the  $i$ -th bit dirty level must therefore be stored in that entry above. So  $DL_j^i = \infty$ ,

For any  $I$  with  $\text{MLM}_j < I \leq w$ , if  $DL_j^I < \infty$ , then a block with the same  $I$ -bit prefix as  $b$  has been written before. Thus, for any  $v \leq i \leq I$ , then  $i$ -th bit dirty level should be finite. As the current ( $j$ -th) stack entry is the *top-most* stack entry  $[b]^i$  for each  $i$  in  $\text{MLM}_j < i \leq I$ , the  $i$ -th bit dirty level is, by the correctness of the algorithm, stored in the current stack entry. Therefore,  $DL_j^i < \infty$ .  $\square$

**Property 2.2** *Each dirty level vector has a non-decreasing finite subvector.*

**Proof.** From statement 40 in the algorithm it is clear that the stack distance vector  $\Delta^k$  is non-decreasing. From statement 44, the new dirty level vector is the maximum of the old dirty level vector and the new stack distance vector, element-wise. If the old dirty level vector is non-decreasing, then the new dirty level vector will still be non-decreasing. Hence the conclusion by induction.  $\square$

It is clear from the proof of Property 2.1, that for any  $j$ -th stack entry, there exists a value  $u_j < w$  such that  $DL_j^i < \infty$  for  $\text{MLM}_j < i \leq u_j$ , and  $DL_j^i = \infty$  for  $v \leq i < \text{MLM}_j$  and  $u_j < i \leq w$ . Hence each stack entry does not need a fixed vector of  $w - v + 1$  elements; instead, it only needs an integer vector of variable length for only the relevant dirty level variables. The vector can be dynamically reallocated according to the change in its length.

To speed up stack searching, we want to avoid unnecessary searching steps. Stack search can stop whenever further searching down the stack will no longer affect any stack entry or stack distance. From the algorithm, it is clear that the

necessary and sufficient condition for the current stack entry to be the stop point of the minimal searching is, that the current stack entry has the maximum left-match function value with  $x_t$ . At the beginning of each stack searching for new access  $x_t$ , we calculate the maximum value of the left-match function between  $x_t$  and all entries in stack  $S_{t-1}$ ; the stop point is the first entry whose left-match function with  $x_t$  equals this value.

The calculation can be easily done with a binary coding tree in a manner similar to Huffman-coding. A binary tree, initially a single root node, is used to record the addresses of all minimum blocks referenced so far. Each node, except the root, corresponds to a binary address. The single digit addresses 0 and 1 correspond to the left and right child of the root, respectively. Generally, if an address  $A$  corresponds to a node  $N$ , then addresses  $A0$  and  $A1$  correspond to the left and right child of node  $N$ , respectively. When a new address appears, its corresponding node is created, together with all the missing internal nodes leading from it up to the root. For example, Figure 2.1 shows how the tree evolves, as the sequence of addresses 0, 10, 1, and 101 are referenced.

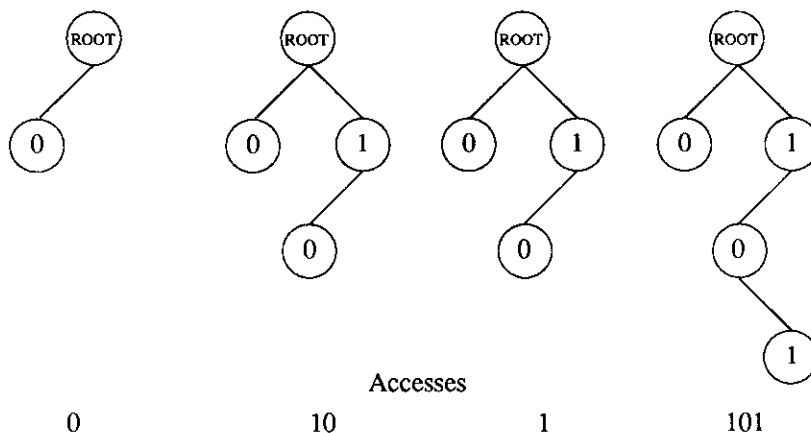


Figure 2.1: Coding Tree

To calculate the maximum left-match value between an address  $a$  and all previous addresses, we traverse from the root down in the binary coding tree as deep as possible, while consuming the binary digits of  $a$  from left to right. At

the  $i$ -th step, we make a left turn if the  $i$ -th left-most digit of  $a$  is 0, and make a right turn if it's 1. We stop when we try to make a left (or right) turn but the current node has no left (or right) child. The depth of the current node is the value we want (root node has depth 0).

At the beginning of each stack search, we search the coding tree for  $x_i$ . At the end of the stack search, we create the node for  $x_i$  if it is not there (i.e., the value found is less than  $w$ ). Note that since all addresses for minimum blocks have the same length (number of digits) of  $w - v + 1$ , all the leaf nodes of the coding tree in this case have the same depth of  $w - v + 1$ .

### 2.4.3 Results

We treat kernel space address and user space address in the same way. Since we are interested in write-back caches, and many cache systems now use a separate data cache and instruction cache, we only consider data accesses in the traces. The simulation of the trace data is done both by the one-pass multiple block-size method, and by the fixed block-size method, assuming the size of a block varies from 8-byte to  $2^{32}$ -byte. Both types of simulation produce exactly the same distributions of stack distance and dirty level on all trace files, validating the correctness of the algorithm.

Write ratios are easily obtained from the simulation results. As suggested in [Agarwal 86], we concatenated the sample traces with the same name into one bigger trace to get longer traces. Figure 2.2 and Figure 2.3 illustrate, for the combined traces, the miss ratio against the block size  $B$  (number of bytes) on a fixed cache capacity of 2048 bytes. The  $x$ -axis uses base-2  $\log B$ . For these particular traces, bigger block sizes almost always result in less write-backs, indicating both the locality and the sequentiality of the reference of these traces.

The simulation running times (user time + system time) are listed in Table 2.2, using a lightly loaded Sun SPARCstation SLC. The running time of multiple block-size evaluation is overall just slightly better than the total time of sequen-

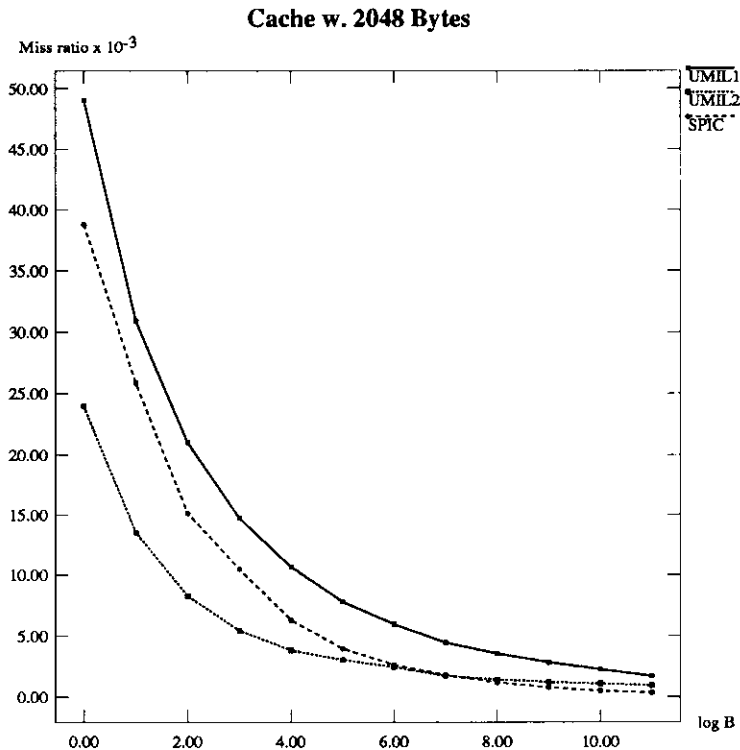


Figure 2.2: Miss ratio v.s. block size on UMIL1, UMIL2, and SPIC.

tial runs of all the separate fixed block-size evaluations. The reason for this unimpressive simulation speed is due to the necessity of our algorithm to search until it hits the stack entry which has the maximum left-match value with the current address. If we solve the memory problem for multiple set-associativity, then a one-pass evaluation for multiple block-sizes and set-associativities might give dramatic speed-ups.

The advantage of this one-pass algorithm lies in its ability of being run *on-the-fly*, i.e., concurrently with trace collection[Hill 89]. Nowadays, the demand on large disk space to store real trace data has become a costly problem nowadays. On-the-fly techniques are a viable way to deal with this problem, by totally avoiding the need to store trace data on disk. Because of the spatial locality of memory access by a CPU, the number of distinct addresses contained in a trace is extremely small when compared to the length of the trace itself. In other words,

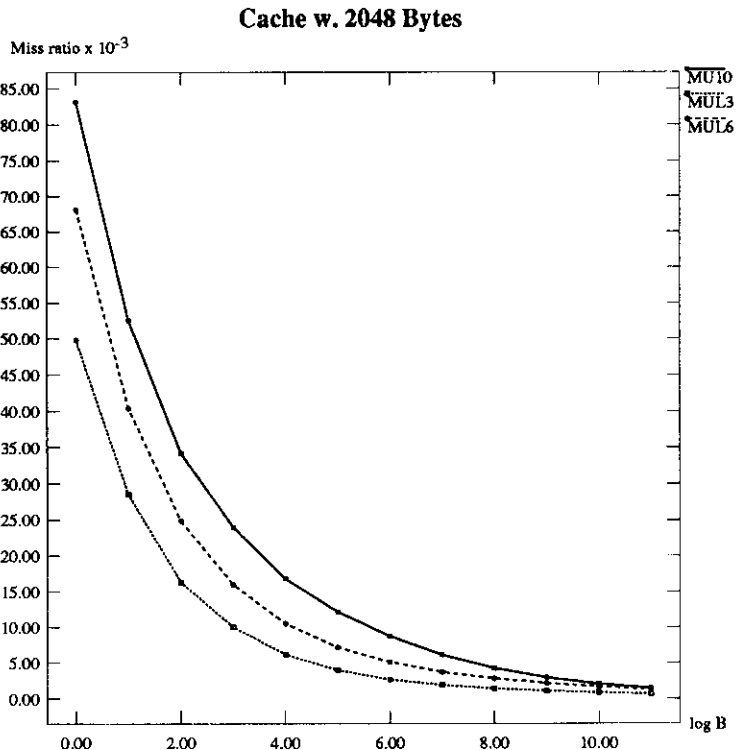


Figure 2.3: Miss ratio v.s. block size on MU10, MUL3, and MUL6.

the size of the simulation stack is very short in comparison to the length of the driving trace file. Through using a little more main memory (allocating extra dirty level variables for each stack entry), our one-pass algorithm can save large amount of disk space.

Running all the fixed-block simulations simultaneously in multiple independent processes and sending the trace data to all of them, one is also a way of doing on-the-fly simulation. However, this requires more memory than multiple block-size simulation, since now each simulation of a fixed block size has its own stack. For example, the pipe-lined running of fixed block-size simulations on trace *forl.001* takes 30MB, 3.5 times as much memory space as the multiple block-size simulation. The problem will become worse when larger address spaces are studied, since the trace will contain a larger number of unique addresses (more memory space used by each stack), and the range of block-size to



Table 2.2: Simulation Times

Trace	fixed ( $f$ )	multi ( $m$ )	$(m - f)/f$
allc	8.3	5.5	-33.7 %
dec0.001	201.7	196.2	-2.7 %
dec1.001	220.9	217	-1.8 %
dia0	195.5	169.4	-13.4 %
for1.000	361.5	224.9	-37.8 %
for1.001	332.6	315.4	-5.2 %
ivex.000	955.8	547.7	-42.7 %
ivex.003	226.6	190.2	-16.1 %
lisp.000	177.3	142.4	-19.7 %
lisp.001	192.8	132.9	-31.1 %
mul10.0	634.9	375	-40.9 %
mul10.1	1005	972.5	-3.2 %
mul10.2	515.7	481.2	-6.7 %
mul3.0	296.7	238.1	-19.8 %
mul3.1	380.1	346.9	-8.7 %
mul3.2	364.3	413.4	13.5 %
mul6.0	349.9	296.4	-15.3 %
mul6.1	495.3	405.9	-18.1 %
mul6.2	503.3	504.5	0.2 %
pasc.001	1055.2	908.4	-13.9 %
spic.000	258.6	296.4	14.6 %
spic.001	481.6	645.1	33.9 %
umil1	246.8	208.8	-15.4 %
umil2	204.7	164.5	-19.6 %

*Note:* Running-time in seconds.

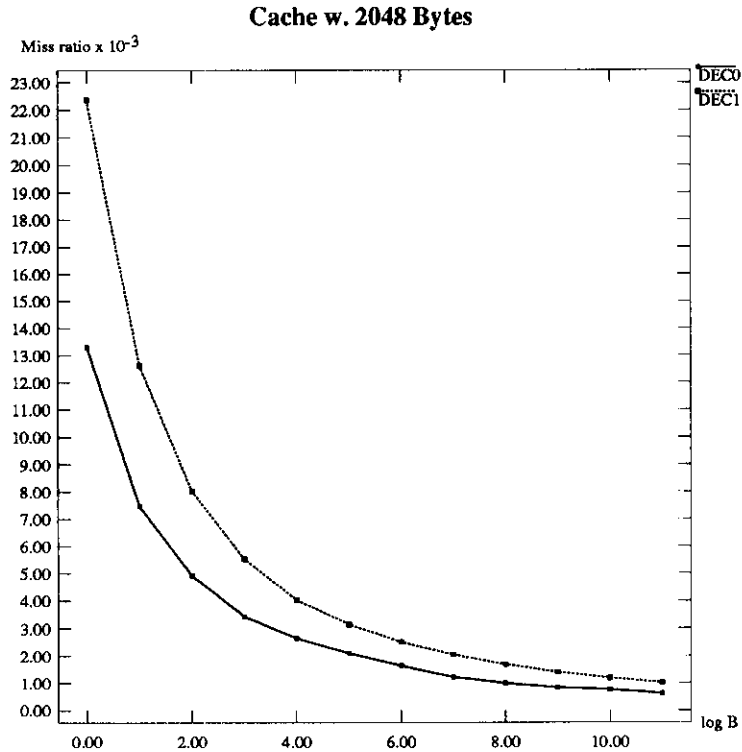


Figure 2.4: Miss ratio v.s. block size on DEC0 and DEC1.

be considered is bigger (more concurrent processes to run); due to more frequent context switching and disk swapping on simulating host, large memory allocation by the concurrent simulation processes could greatly slow down the entire simulation.

## 2.5 Conclusion

We propose a one-pass evaluation algorithm that evaluates write-back LRU caches for multiple block-sizes. As a candidate for on-the-fly simulation, it is a useful tool in solving the problem of doing trace-driven simulation without storing enormous trace data on disk. Combining previous stack evaluation techniques on multiple block-size read-only LRU caches and fixed block size write-back caches, one can extend efficient stack evaluation techniques to multiple block-size write-back

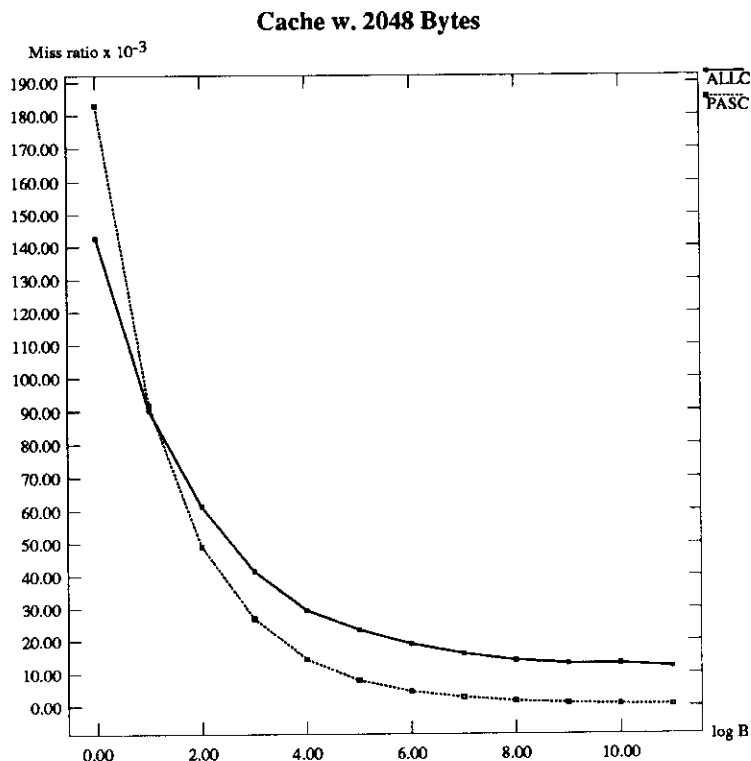


Figure 2.5: Miss ratio v.s. block size on ALLC and PASC.

LRU caches. There is a useful fact about LRU replacement: even though there can be many  $2^{n-k}$ -byte blocks in the  $2^{n-w}$ -byte block LRU stack  $S_t^w$  at time  $t$  for  $k < w$ , only the top-most entries with distinct  $k$ -bit prefixes in  $S_t^m$  are relevant for evaluating the stack distance of  $2^{n-k}$ -byte block. By keeping information on previous writes in the first  $S_t^w$  entries with distinct  $k$ -bit prefixes, one can determine the stack distance and dirty level of the currently accessed data block  $x_t^k$ , for every value of block-bit  $k < w$ , in one scan of the minimum-block stack  $S_t^w$ . Using a vector of dirty levels on each stack entry, and keeping the dirty level for any block-size attached to its corresponding top-most stack entry, the multiple block-size technique for read-only caches is extended to write-back caches.

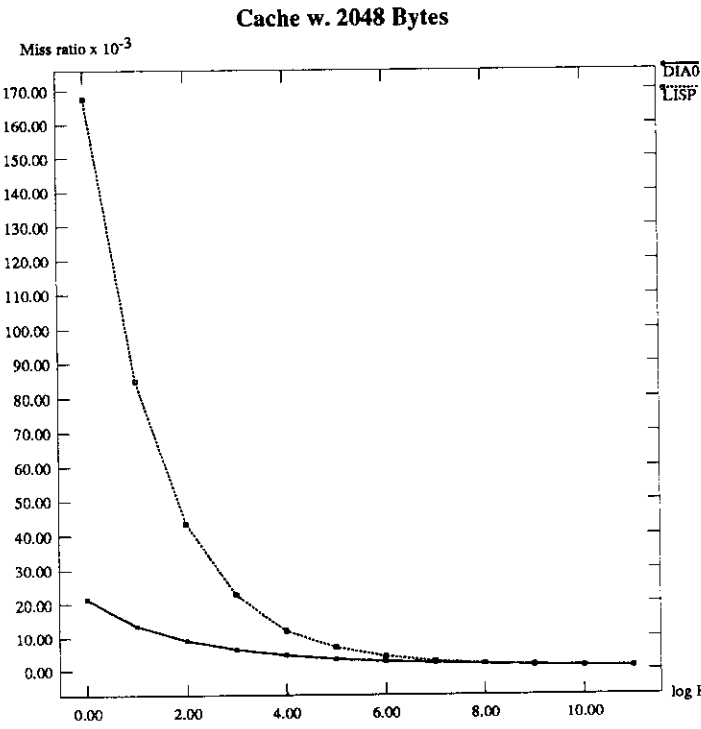
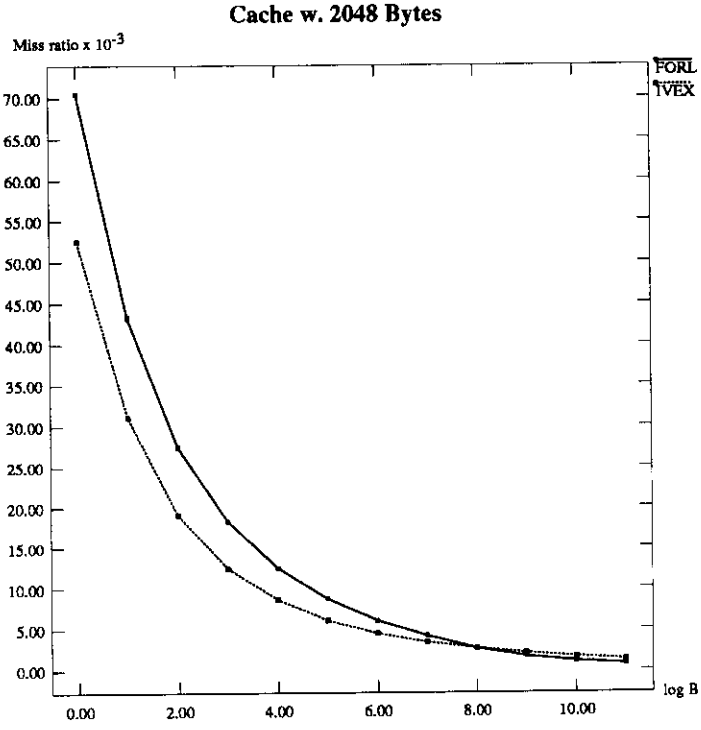


Figure 2.6: Miss ratio v.s. block size on FORL, IVEX, DIA0, and LISP.

# Chapter 3

## Set-Associative Multiprocessor Caches

**Abstract.** We propose a simple solution to the problem of efficient stack evaluation of LRU multiprocessor cache memories with arbitrary associative set-mapping. It is an extension of the existing stack evaluation techniques for all set-associative mapping schemes LRU uniprocessor caches. Special marker entries are used in the stack to represent data blocks (or lines) deleted by an invalidation-based cache coherence protocol. A method of marker-splitting is employed when a data block below a marker in the stack is accessed. Using this technique, one-pass evaluation of memory access trace yields hit ratios for all cache sizes and set-associative schemes of multiprocessor caches in a single pass over a memory reference trace. Simulation experiments on real multiprocessor trace data show an order-of-magnitude speed-up in simulation time using this one-pass technique.

### 3.1 Introduction

When the least-recently-used (LRU) algorithm is used by the cache as its block replacement algorithm, a one-pass stack processing can produce hit ratios for all set-associative mappings[Mattson 70]. We are interested in extending this set-associative stack evaluation technique for an LRU cache on an uniprocessor

computer to LRU caches on a multiprocessor computer.

There are issues regarding the validity of trace-driven simulation for multiprocessor caches: perturbation to the trace data by the tracing mechanism, differences in trace data across different runs of the same program, and particular system configuration (e.g., cache sizes) under tracing. All these factors can potentially change the global execution order of the program(s) being traced, since they all affect the way processors interact with one another, including their relative order of arriving at some synchronization point, their relative speed in finishing their assigned jobs, and consequently their job scheduling.

It has been found[Koldinger 91] that there is insignificant difference in simulation results due to tracing perturbation, for both process-based coarse-grained parallel programs and thread-based medium-grained parallel programs; miss ratios did not vary much between different runs of a program, especially for coarse-grained programs. For stack evaluation, even though the actual ordering of memory access requests may be altered by the changes in exactly which references are cache misses due to the different cache sizes, the changes would be slight and their effect on simulation results insignificant[Smith 93]. Generally, stack evaluation methods are useful in predicting the general performance trend of cache memories, making a helpful tool in the early design stages. They can be used to narrow the initially vast design space into a few, more manageable choices, which in turn can be studied using other less efficient but more accurate methods such as case-by-case simulations or software-driven emulation.

On multiprocessors where each CPU has its own local cache, the issue of cache coherence arises. Like a uniprocessor cache, each cache still has to implement some block replacement algorithm, in case a referenced block is not in the cache and no cache space is available for it. Unlike a uniprocessor cache, however, multiprocessor caches have to interact among themselves by some cache coherence protocol to keep all the caches consistent[Dubois 82, Dubois 88, Stenström 90]. There are three categories of cache coherence protocols: immediate-coping, invalidation, and validation. Immediate-coping is the case where the updating cache

broadcasts the changed data to all other caches [McCreight 84]. Invalidation, on the other hand, does not broadcast the data; instead the updating cache sends an invalidation message to all other caches holding a copy of the data block, and they discard their copies [Censier 78, Papamarcos 84, Archibald 86, Cheong 88, Li 89, Chaiken 90]. With the validation methods, the updating cache does not broadcast the new data nor send an invalidation to other caches; before accessing any local data block, a cache must make sure that it has acquired the latest version of that data block. If the cache owns the block, then it is guaranteed to have the latest copy; otherwise, it has to contact the owner of the block (a remote cache or the shared memory) in order to have the latest version. Generally, immediate-coping is good for infrequent writes and high degree of data sharing, and invalidation is suitable for frequent writes and low data sharing. A validation-based protocol incurs high overhead and is deemed impractical.

For an immediate-coping protocol, a write to a block in one cache does not change the presence or absence of that block in another cache. Other CPU's writing will update a block if it is in cache—no effect if it is not. From the view-point of cache evaluation, a CPU's local cache is unchanged by reads or writes in other caches and stack evaluation can be applied independently for each cache. For a validation-based protocol, the situation is similar: the content of a CPU's local cache is independent of those of other caches and independent stack evaluation can be done for each cache.

The interesting case is an invalidation-based protocol, where a write in one cache results in blocks being invalidated (deleted) in other caches. In other words, an invalidation by a write in one cache produces an empty block frame in all other caches that have a copy of the changed data block. From the standpoint of stack evaluation, this effectively leaves an empty block in the stacks of the effected caches.

Mattson, et al. [Mattson 70] used a special marker entry “#” in the stack to represent an empty block frame caused by the invalidation of an I/O operation. All marker entries contribute to stack evaluation, and the invalidated block

frames in the cache have the highest priority of being selected in replacement decisions. For fully-associative LRU caches, a marker will remain at the same position in the stack until another data block below it is accessed, at which time the marker is moved down to the stack position of the newly accessed data block, and the referenced data block is moved up to the top of the stack. For set-associative caches, whether a marker should move, when a data block below it is accessed, is dependent upon whether the empty frame (more precisely, the invalidated data block from which the empty frame was obtained) and the newly requested data block are in the same set. As two blocks can be in the same set for one associativity and in different sets for another associativity, the movement of a marker is not obvious for arbitrary set-associative mapping. This problem was proposed and its difficulty discussed by Wang and Baer[Wang 91].

In this chapter we propose a solution to this problem. In section 2 we present a new method for one-pass evaluation of multiprocessor caches with invalidation-based cache coherence protocols, yielding performance measures for all set-associative mappings. In section 3 we give simulation results of the method on some multiprocessing application trace data. Section 4 concludes with a summary.

## 3.2 Multiprocessor LRU Set-Associative Evaluation

As with uniprocessor caches, we want to make a one-pass scan of the fully associative LRU stack  $S_{t-1}$  and get stack distance  $\Delta_t^\alpha$  of every set-associative mapping scheme (or set length)  $\alpha$  for multiprocessor caches. As mentioned earlier, an invalidated block in the stack is represented by a marker entry. If a data block below a marker in the stack is accessed, the marker may need to be moved down to a new stack location, depending on whether the accessed data block and the marker are in the same set. The newly accessed data block is always moved up



to the top of the stack.

With fully-associative caches, the movement of a marker is simple. If there is one marker above the accessed data block in the stack, the marker is moved down to the stack location of the data block. If there are two or more markers above the accessed data block, then only the top-most one is moved down to the data block's stack location, while the other markers remain in their locations[Mattson 70, Thompson 87]. To update LRU stack  $S_{t-1}$ , we scan down the stack until  $x_t$  is found or the stack is exhausted, remembering the location of the first marker along the way. This is shown in Figure 3.1A. In case the accessed block is not in the stack, the top-most marker is removed from the stack, as is shown in Figure 3.1B. One can easily see it is the correct way to update a stack with markers, by considering each case where the cache size is at least as big as the stack position of the  $i$ -th marker in the stack but less than that of the  $(i + 1)$ -th marker in the stack, before stack updating.

To update an arbitrary set-associative LRU stack with markers, We will employ a method called marker splitting.

### 3.2.1 Marker Splitting

When a data block has just been invalidated, it becomes a marker that has presence in  $(k + 1)$  sets, for set length  $\alpha$  ranging from 0 to  $k$ . These are the sets that the invalidated data block would be in for each set length. We represent each marker  $m$  with a 2-tuple  $(b, v)$ , where  $b$  is the block address of the original data block that was invalidated, and  $v$  is a  $(k + 1)$  element vector whose elements are either 0 or 1:  $v[i] = 0, 1$  for  $0 \leq i \leq k$ . We shall call  $b$  the *address*, and  $v$  the *covering vector*, respectively, of the marker.  $m$  represents a marker presence in set  $(b \bmod 2^\alpha)$  of set length  $\alpha$  if its  $v[\alpha] = 1$ . When a data block  $b$  is invalidated, it changes into a marker  $m = (b, v)$  with  $v$  set to 1's:  $v[i] = 1$  for  $0 \leq i \leq k$ , representing  $(k + 1)$  singular, indivisible markers in each set length. Let  $|m|$  be the number of singular markers  $m$  is composed of:  $|m| = \sum_{i=0}^k v[i]$ .

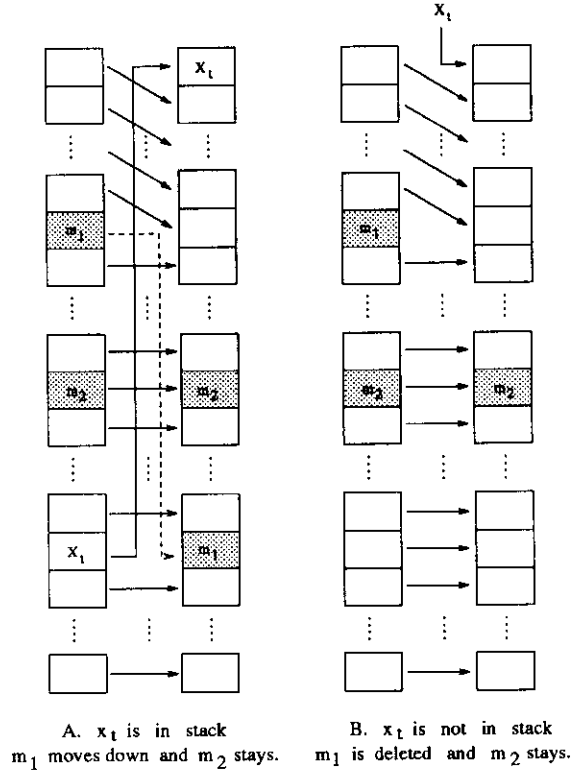


Figure 3.1: Constructing fully associative LRU stack with marker(s)

A marker is *composite* if it denotes multiple singular markers for different set lengths. Two markers are *disjoint* if the dot product of their covering vectors is zero, i.e.  $\sum_{i=0}^k v_1[i] \times v_2[i] = 0$ . Disjoint markers do not have presence in the same set for any set length.

When a data block in the stack is accessed, and there is one marker  $m$  above it in stack, with  $m = (b, v)$ , then  $m$  is split into two disjoint markers  $m' = (b, v')$  and  $m'' = (b, v'')$ , with  $|m| = |m'| + |m''|$ .  $m'$  represents the original marker's presence in the  $|m'|$  sets that do not contain the data block, while  $m''$  denotes the original marker's presence in the  $|m''|$  sets that contain the currently accessed data block.  $m'$  replaces  $m$ , and  $m''$  is moved down the stack to where the data block resides; this is illustrated by Figure 3.2A. If either  $m'$  or  $m''$  is empty ( $|m'| \cdot |m''| = 0$ ), there is no splitting:  $m$  either moves down or stays where it is, depending on which one of  $m'$  and  $m''$  is empty.

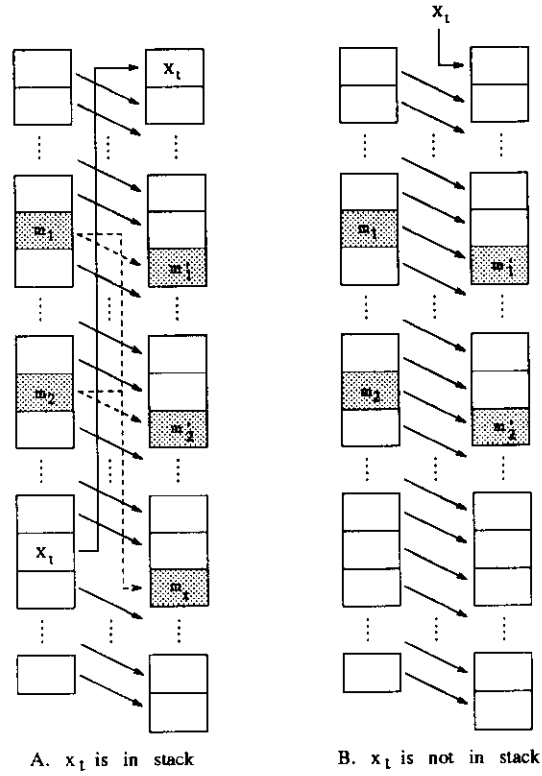


Figure 3.2: Constructing set-associative LRU stack with marker splitting

If, when a data block in the stack is accessed, there are multiple markers above it in the stack, the situation becomes a bit more complex. Unlike the fully associative case, where we only need to remember one top-most marker, here we have to remember the top-most *singular* marker for each set length. Since each marker above the accessed block in stack might contain some singular marker that is in a same set as the data block for some set length, we must examine each one of them and remember the top-most singular marker for each set length along the way. Remember that for each set length, we only want to move down the top-most singular marker, while keeping other singular markers in place.

Details of the stack updating procedure is given in section 3.2.2. Here we summarize the three possible ways of adjustment of a marker which is above the accessed data block in the stack:

- marker remains unchanged and in the same location; i.e., the marker and

the referenced block are not in a same set for any set length, or for any set length that they do share the same set, there was a singular marker above in the stack.

- marker remains unchanged but is moved down to the data block's stack location; i.e., the marker and the referenced block are in the same set for all set lengths which the marker has a valid singular marker, this singular marker is the top-most one in the stack for its set length.
- marker gets split into two disjoint markers; one of them remains in the same location, the other is moved down to the data block's stack location.

The outcome depends on the addresses of the data block and the marker, the covering vector of the marker, and the existence of other markers that are above the current marker in the stack.

### 3.2.2 Stack Updating

Suppose there is only one marker  $m = (b, v)$  above the accessed block  $x_t$  in the stack  $S_{t-1}$ . Let  $r = \text{RM}(x_t, b)$ . If  $v[i] = 0$  for all  $0 \leq i \leq r$ ,  $m$  is a non-existent marker to block  $x_t$  ( $m'$  is empty), and hence can be treated as a normal data block. If  $\exists j$  with  $0 \leq j \leq r$  and  $v[j] = 1$ , but  $v[i] = 0$  for all  $r < i \leq k$ , then  $m$  is in the same set as  $x_t$  for every relevant set length ( $m'$  is empty); in this case  $m$  is moved down to the stack location of  $x_t$ . If  $\exists i, j$  such that  $0 \leq i \leq r < j \leq k$  and  $v[i] = 1 = v[j]$ , then  $m$  is replaced by two disjoint markers  $m' = (b, v')$  and  $m'' = (b, v'')$ , whereby

$$v'[i] = \begin{cases} 0 & \text{if } 0 \leq i \leq r \\ v[i] & \text{if } r < i \leq k \end{cases}$$

$$v''[i] = \begin{cases} v[i] & \text{if } 0 \leq i \leq r \\ 0 & \text{if } r < i \leq k \end{cases}$$

$m'$  replaces  $m$  in the stack, and  $m''$  is moved down to that of  $x_t$ , as shown in Figure 3.2A.

If  $x_t$  is not in stack  $S_{t-1}$ ,  $m'$  still replaces  $m$ , but  $m''$  is thrown from the stack. This is shown in Figure 3.2B.

If there are multiple markers above an accessed block in the stack, then for each set length, the top-most singular marker that shares the same set with the accessed block needs to be moved to the location of the data block. All the top-most singular markers in stack  $S_{t-1}$  are collected during the search for  $x_t$  and, upon finding  $x_t$ , they are placed at  $x_t$ 's location. The general procedure is: search stack  $S_{t-1}$  for the accessed data block  $x_t$ , and whenever a marker is found, be it composite or singular, check if it contains any singular marker that shares the same set with  $x_t$  for a set length, for which a singular marker has not been seen yet. If there are no such singular markers, leave the current marker intact. Otherwise, collect all eligible singular markers, and remove them from the current marker; if the current marker subsequently becomes empty, delete this marker entry from the stack.

Finally, when the data block  $x_t$  is found, all the collected singular markers are grouped into one composite marker and it replaces  $x_t$  in the stack, while  $x_t$  is moved to the top of the new stack  $S_t$ . Since these singular markers share some set with  $x_t$ , and these sets are for different set lengths, each of them can be represented by a marker  $(x_t, v)$ ; the covering vector  $v$  has only one non-zero element (of value 1), whose index is the set length for which the represented singular marker shares the same set with block  $x_t$ . Therefore they can be put into one composite marker  $(x_t, v)$ , whose covering vector is simply the sum of those covering vectors of the collected singular markers.

If the data block  $x_t$  is not in stack  $S_{t-1}$ , all collected singular markers are discarded.

The LRU stack updating procedure for stack  $S_{t-1}$  while scanning stack for  $x = x_t$  is as follows:

### Stack updating procedure

Let  $w$  be a  $k + 1$  element integer vector initialized to all zero.

Scan down the stack  $S_{t-1}$  and do the following at each entry  $E = s_{t-1}(j)$  until  $x = x_t$  is found or  $S_{t-1}$  is exhausted:

If  $E = (b, v)$  is a marker, let  $r = \text{RM}(x, b)$ ,  $U = \{ i \mid (0 \leq i \leq r) \wedge (v[i] = 1) \}$ ,  $W = \{ i \mid (0 \leq i \leq r) \wedge (w[i] = 0) \}$ . Change  $E$ 's covering vector  $v$  into

$$v[i] = \begin{cases} 0 & \text{if } i \in U \cap W \\ v[i] & \text{otherwise} \end{cases}$$

Change vector  $w$  into

$$w[i] = \begin{cases} 1 & \text{if } i \in U \cap W \\ w[i] & \text{otherwise} \end{cases}$$

Leave marker  $E$  where it is if  $v$  is not an all-zero vector; remove  $E$  from the stack if  $v$  is all-zero.

If  $E$  is a data block but  $E \neq x$ , continue.

If  $E$  is a data block and  $E = x$ , then replace  $s_{t-1}(j)$  by a new marker  $m = (x, w)$  if  $w$  is not all zero, and put block  $x$  on top of the stack. Break out of the loop.

If  $S_{t-1}$  is exhausted but  $x$  is not in the stack, throw away the collected vector  $w$ , and just pull  $x$  to the top of the stack.

The updated stack  $S_{t-1}$  is  $S_t$ .

### Example

There are two markers above a referenced data block  $x = 0101$  in the stack. The first marker is  $(b_1, v_1) = (1101, 11001)$ , and the second marker is  $(b_2, v_2) = (1001, 10011)$ . All vectors are listed with their highest element first. For example,  $v_1[4] = 1, v_1[3] = 1, v_1[2] = 0, v_1[1] = 0, v_1[0] = 1$ .  $\text{RM}(x, b_1) = 3, \text{RM}(x, b_2) = 2$ . Initially  $w = 00000$ . After scanning  $b_1, b_1 = (1101, 10000), w = 01001$ . After

scanning  $b_2$ ,  $b_2 = (1001, 10001)$ ,  $w = 01011$ . When  $x$  is reached,  $x$  is moved to the top of the stack, and a new marker  $(0101, 01011)$  is put in  $x$ 's previous slot.

**Theorem 3.1** *The stack updating procedure correctly places markers for all set lengths.*

**Proof.** It is clear, from the stack updating procedure, that the vector  $w$  records which set lengths have already had a marker in the same set as  $x$ . That is, for each set length  $\alpha$  with  $0 \leq \alpha \leq k$ ,  $w[\alpha] = 1$  if and only if we have already seen at least one marker in the stack belonging to its set  $[x]_\alpha$ . The changing of the covering vector  $v$  of each marker  $E = (b, v)$  encountered by  $x$  in the stack is to take away all the *first* markers that are in the same set with block  $x$  for some set length. The change to the vector  $w$  is to add those newly found *first* markers. If  $x$  is found in the stack, then these first markers should all be moved to the location of  $x$ , which is what the replacement of  $x$  by  $m = (x, w)$  does in the procedure. If  $x$  is not found in the stack, then all these first markers should be removed from the stack.  $\square$

### 3.2.3 Stack Distance Counting

We have just shown how to update the global LRU stack. In this section we discuss how to calculate stack distances for all set lengths. These two operations, stack updating and stack distance counting, are in fact carried out simultaneously during stack scanning.

First let us define a couple of simple and useful functions. The *trailing one function*  $\text{TO}(v, k)$  on vector  $v$  is the set of indices of the *last* element in each string of consecutive 1's in  $v$  that are less than or equal to  $k$ . That is,  $\text{TO}(v, k) = \{ i \mid (v[i] = 1) \wedge (((i < k) \wedge (v[i + 1] = 0)) \vee (i = k)) \}$ . For example,  $\text{TO}(\{10010110\}, 7) = \{2, 4, 7\}$ , and  $\text{TO}(\{01010111\}, 7) = \{2, 4, 6\}$ .

Define the *trailing zero function*  $\text{TZ}(v, k)$  on vector  $v$  to be the set of indices of the *last* element in each string of consecutive 0's in  $v$  that are less than  $k$ . That is,  $\text{TZ}(v, k) = \{ i \mid (i < k) \wedge (v[i] = 0) \wedge (v[i + 1] = 1) \}$ . For example,

$\text{TZ}(\{10010110\}, 7) = \{0, 3, 5\}$ , and  $\text{TZ}(\{01010111\}, 7) = \{3, 5\}$ .

Let  $\{\mu(r)\}$  and  $\{\nu(r)\}$  be two groups of counters for  $0 \leq r \leq k$ . To determine  $\{\Delta_t^\alpha\}$  for all  $\alpha$ , scan down the stack  $S_{t-1}$  until  $x = x_t$  is found or the stack is exhausted. Suppose the current stack entry being examined is the  $j$ th entry in stack  $S_{t-1}$ :  $E = s_{t-1}(j)$ . If  $E$  is a data block, increment counter  $\mu(\text{RM}(x, E))$ . If  $E$  is a marker  $(b, v)$ , consider values of  $r = \text{RM}(x, b)$  and  $v[i]$ 's. If  $v[i] = 0$  for all  $0 \leq i \leq r$ , marker  $E$  is non-existent to block  $x$ , do nothing. Otherwise, increment  $\mu(i)$  for all  $i \in \text{TO}(v, r)$  and  $\nu(j)$  for all  $j \in \text{TZ}(v, r)$ . If  $x$  is found in the stack  $S_{t-1}$ , then each stack distance  $\Delta_t^\alpha$  is given by

$$\Delta_t^\alpha = \sum_{r=\alpha}^k (\mu(r) - \nu(r)) \quad (3.1)$$

where  $0 \leq \alpha \leq k$ . If  $S_{t-1}$  is exhausted and  $x$  is not found, all stack distances  $\Delta_t^\alpha$  are set to  $\infty$ . As in the uniprocessor case, the stack distance counter  $n_\alpha(\Delta_t^\alpha)$  is incremented for each set length  $\alpha$ .

### Numerical example

There are three data blocks and two markers above a referenced data block  $x = 0101$  in the stack. The three data blocks are  $x_1 = 0111$ ,  $x_2 = 0000$ ,  $x_3 = 0001$ . The markers are  $(b_1, v_1) = (1101, 11001)$ ,  $(b_2, v_2) = (1001, 10011)$ . The order of their appearance on the stack is, from top down,  $x_1, b_1, x_2, b_2, x_3, x$ . As before, all vectors are listed with their highest element first. Initially,  $\mu = 00000$ ,  $\nu = 00000$ .

scanning  $x_1$ :  $\text{RM}(x, x_1) = 1$ ,  $\mu = 00010$ ,  $\nu = 00000$ .

scanning  $b_1$ :  $\text{RM}(x, b_1) = 3$ ,  $\mu = 01011$ ,  $\nu = 00100$ .

scanning  $x_2$ :  $\text{RM}(x, x_2) = 0$ ,  $\mu = 01012$ ,  $\nu = 00100$ .

scanning  $b_2$ :  $\text{RM}(x, b_2) = 2$ ,  $\mu = 01022$ ,  $\nu = 00100$ .

scanning  $x_3$ :  $\text{RM}(x, x_3) = 2$ ,  $\mu = 01122$ ,  $\nu = 00100$ .

When  $x$  is reached, the stack distances are, according to Equation (3.1),

$$\Delta^0 = 5, \Delta^1 = 3, \Delta^2 = 1, \Delta^3 = 1, \Delta^4 = 0.$$

**Theorem 3.2** *Equation 3.1 correctly computes the stack distances  $\Delta_t^\alpha$ .*



**Proof.** When the stack entry is a data block, Equation (1.2) in section 1.5.1 applies, namely

$$\Delta_t^\alpha = \sum_{r=\alpha}^k \mu(r). \quad (3.2)$$

So we only need to consider the case where the current stack entry is a marker. We depict in Figure 3.3 the covering vector of a marker  $E = (b, v)$  encountered during stack scanning for block  $x$ , with the left-most rectangle representing the  $k$ th vector element and the right-most one representing the 0th vector element. A shaded rectangle denotes an element of value 1, and an unshaded one denotes an element of value 0. Here  $r = \text{RM}(x, b)$ . Notice that all rectangles indexed from  $r$  up to  $k$  are all unshaded (zero elements), since they are irrelevant to  $x$ .

It is clear that marker  $E$  represents an empty entry in the same set as  $x$  for set length  $\alpha$  (i.e.,  $[x]_\alpha$ ) if and only if the  $\alpha$ th rectangle in the figure is shaded. Thus exactly these set lengths, whose corresponding element in the figure is shaded, should increment their respective stack distance counters by one. For the specific depiction in Figure 3.3, these set lengths are  $a, i, p, p - 1$ , and 1.

Suppose at first  $\mu(r) = \nu(r) = 0$  for each  $r$ . Then after evaluating the first marker,

$$\sum_{r=\alpha}^k (\mu(r) - \nu(r))$$

is equal to 0 if the  $\alpha$ th rectangle is not shaded, and is equal to 1 if it is shaded. Generally, suppose a new referenced block  $x$  has been searched in the stack, and  $\mu^+(r), \nu^+(r)$  are the new counter values. By the same reasoning, it follows that

$$\sum_{r=\alpha}^k (\mu^+(r) - \nu^+(r)) - \sum_{r=\alpha}^k (\mu(r) - \nu(r)) = 0$$

if the current  $\alpha$ th rectangle is not shaded, and = 1 if it is shaded. This means that Equation (3.1) correctly counts stack distance when the stack entry is a marker.  $\square$

The stack distance contributed by markers for each set length  $\alpha$  can be counted directly during stack scanning. Let  $\lambda(\alpha)$  be the stack distance counter for markers of set length  $\alpha$ . When  $E$  is a data block, increment counter  $\mu(\text{RM}(x, E))$

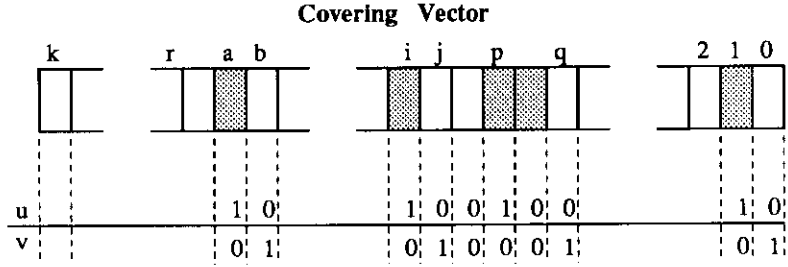


Figure 3.3: Counting of  $\mu(r), \nu(r)$

as before. When  $E$  is a marker  $(b, v)$ , increment  $\lambda(\alpha)$  for all  $\alpha \in \{i \mid (0 \leq i \leq \text{RM}(x, b)) \wedge (v[i] = 1)\}$ . For Figure 3.3, counters  $\lambda(a), \lambda(i), \lambda(p), \lambda(p-1), \lambda(1)$  are incremented. We have another way to calculate the stack distances:

**Theorem 3.3** *The stack distance  $\Delta_t^\alpha$  is also given by*

$$\Delta_t^\alpha = \sum_{r=\alpha}^k \mu(r) + \lambda(\alpha) \quad (3.3)$$

**Proof.** Incrementing of counter  $\lambda(\alpha)$  remembers the number of times the  $\alpha$ th rectangle is shaded during the stack scanning process. This is exactly the number of times an empty entry has appeared in set  $[x]_\alpha$  before  $x$  is found. Hence, counting both data blocks and empty entries, the sum of the right-hand side of Equation (3.2) and  $\lambda(\alpha)$  gives the total stack distance of  $x$  for set length  $\alpha$ .  $\square$

### Numerical example

We use the same example. Initially,  $\mu = 00000, \lambda = 00000$ .

scanning  $x_1$ :  $\text{RM}(x, x_1) = 1, \mu = 00010, \lambda = 00000$ .

scanning  $b_1$ :  $\text{RM}(x, b_1) = 3, \mu = 00010, \lambda = 01001$ .

scanning  $x_2$ :  $\text{RM}(x, x_2) = 0, \mu = 00011, \lambda = 01001$ .

scanning  $b_2$ :  $\text{RM}(x, b_2) = 2, \mu = 00011, \lambda = 01012$ .

scanning  $x_3$ :  $\text{RM}(x, x_3) = 2, \mu = 00111, \lambda = 01012$ .

When  $x$  is reached, the stack distances are given by Equation (3.3) as

$$\Delta^0 = 5, \Delta^1 = 3, \Delta^2 = 1, \Delta^3 = 1, \Delta^4 = 0.$$

This is the same result as before.

All the auxiliary functions  $TZ(v, k)$ ,  $TO(v, k)$  and  $RM(x, b)$  used in stack distance counting and stack updating are simple operations. So the time spent in processing a marker stack entry is not much more than that spent in processing a data block entry.

### 3.2.4 Time Complexity

Our one-pass evaluation saves time in two aspects: the trace data needs to be read only once, and the stack processing is only done once for each access. What we do with arbitrary set-associative evaluation is use one composite marker to represent all  $k + 1$  possible singular markers, and split it when we have to. The number  $M$ , of markers in the stack, is bounded by  $M \leq (k + 1)I$ , where  $I$  is the number of effective block invalidations, the invalidations that actually find their target blocks in the stack. Under the assumption that the rate of actual invalidations is not high in real applications, the number of markers produced in the stack will not be very large. Once a marker becomes singular, it will not split further. Therefore the stack does not grow indefinitely because of marker splitting. In addition, markers never ascend in the stack; they tend to descend in the stack as data blocks below them in the stack are accessed. Whenever a marker becomes the last entry of the stack, it can be dropped. The extra work needed in arbitrary set-associative evaluation is simple; the vector operations in singular marker collection and stack updating can be done with efficient bit operations on the simulating machine.

As the method requires sequential scanning of all stack entries above the accessed block in the stack, it defies efficient search data structures such as balanced trees for the representation of the stack. However, as we will see later in simulation experiments, with the exception of general hash tables which can be used by almost any stack evaluation method, sophisticated data structures such as search trees do not noticeably reduce the overall simulation time of a stack evaluation

method<sup>1</sup>. This is because the locality property of CPU access produces on average short stack distances, making linear search of the stack quite inexpensive. Moreover, if most references are near the top of the stack, stack searching does not “see” many of the markers on most references.

### 3.3 Simulation

We have implemented both the arbitrary set associative evaluation algorithm and the conventional single set associative evaluation algorithm, in order to compare their performances. In this section we report simulation experiments on some real multiprocessor trace data. We are mainly interested in the comparison of simulation times in getting stack distance distributions for all set lengths on a given block size. The characteristics of the trace data are given, followed by a detailed description of algorithm implementations. The simulation results are given in the end.

#### 3.3.1 Trace Data

Three traces of parallel applications are used: Weather, Simple, and FFT. They were obtained using the IBM postmortem scheduling method and represent a possible execution on a 64-CPU multiprocessor [Cherian 89, Chaiken 90]. The Weather application partitions the earth atmosphere into a three dimensional grid and uses finite-difference methods to solve a set of partial differential equations describing the system state. The Simple application models the behavior of fluids and also uses finite difference methods to solve equations on hydrodynamic behavior. FFT is a radix-2 fast Fourier transform application. Each reference record consists of a one-byte CPU number (ranging from 1 to 64), a one-byte operation code (for data/instruction read/write), and a four-byte memory ad-

---

<sup>1</sup>Thompson first observed this phenomena while comparing different data structures in the implementation of stack evaluation of uniprocessor write-back caches [Thompson 87].

dress. The length of each trace, i.e., the number of references, is respectively 7461123(FFT), 27172624(Simple), and 31777053(Weather).

### 3.3.2 Implementation

We will compare the run-time efficiency of the one-pass evaluation algorithm for all set-associative schemes with that of conventional multiple-pass evaluation algorithm for a single set-associative scheme. The data and instruction accesses are “unified”; i.e., we treat them as being cached together. To obtain fair and convincing results, we tried to make each implementation of an algorithm run as fast as possible.

Preliminary tests showed that, for the single set-associative algorithm, an implementation of the linked-list stack structure without using hashing ran significantly slower. So we applied the hashing technique in all the implementations and do not consider any non-hashing implementation for performance comparisons.

#### Single Set-Associative Algorithm

For a set-associative cache, space is partitioned according some congruence set-mapping scheme; different sets are independent of one another. This translates to one (sub)stack for each set. So the conventional single set-associative algorithm maintains as many stacks as there are different sets in the trace data. Of course, the total number of distinct blocks (hence stack entries for valid data blocks) is the same regardless of the associativity.

One simple technique for efficient stack simulation is to maintain a hash table of all data blocks currently residing in the stack[Thompson 87]. It helps eliminate fruitless searches for blocks not even in the stack. Hashing proves to be very effective on uniprocessor traces[Thompson 87].

We use a two-level hashing table to hold all *valid* data blocks currently in stacks. It should provide faster look-up than one-level hashing, particularly when

the number of distinct blocks is not small. At the beginning of each stack search, the program first determines whether the referenced block is currently in the corresponding stack (by looking in the hash table), and whether there are any marker entries in the stack (by checking a counter variable). If neither holds, then there is no need to search the stack.

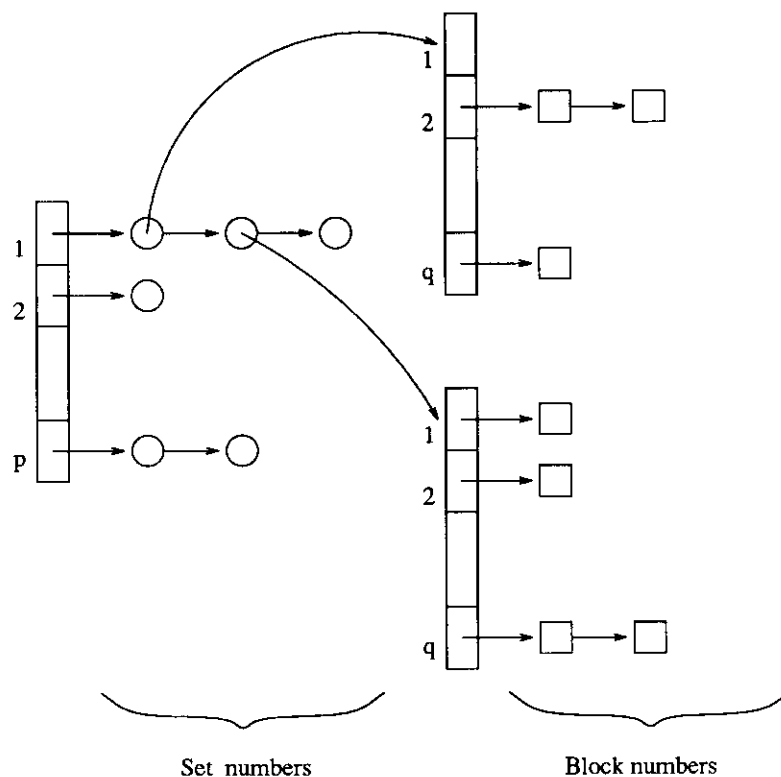


Figure 3.4: Two-level hash table

The two-level hashing table, shown in Figure 3.4, is organized for a set length  $\alpha$  under study. For a block number  $b$ , its associated set number  $s = b \bmod 2^\alpha$  is used to probe the first level to find the set of the block, and its block number  $b$  to probe the second level to find the block itself. The reason for using a hash table on set numbers is as follows: when the value of  $\alpha$  is not trivial, for example  $\alpha = 20$ , the number of different sets is not small. An array indexed by set numbers requires a lot of space, some of which may be unused; a dynamically allocated linked-list has a relatively long search time.

At the first level, there is an array of  $p$  pointers, each one of which points to a list containing distinct set numbers with equal value mod  $p$ . The set number lists dynamically grow when new set numbers are encountered in trace. It is more efficient than static allocation of hash table entries for all possible sets; some application traces might not utilize all possible sets, especially for big values of set length  $\alpha$ . Not shown in the figure, each element in the set lists has a pointer to the substack (implemented with either a list or a tree, see section 3.3.2) of the set whose set number is contained in this element.

At the second level, each element in the set lists contains  $q$  pointers, each of which points to a list of elements containing block numbers with equal value mod  $q$ . These block number lists dynamically grow (when a new data block is referenced) or shrink (when an existing data block is invalidated) during trace processing. When a valid data block in a stack gets zapped, its corresponding block element is deleted from the second level in the hash table.

As stated before, any marker at the tail of the stack is void and is promptly dropped by all implementations. This eliminates unnecessary memory consumption and improves algorithm performance.

To save space, instead of using an array of counters with fixed dimension and having a lot of zero elements, stack distance counters are also dynamically implemented with a linked-list, sorted with increasing stack distance value. Even though incrementing a counter is no longer done in constant time now, thanks to locality in memory reference, stack distances tend to be small, and the time spent in looking for the right counter is negligible. We found virtually no difference in execution time of simulation whether array counters or link list counters are used.

### **Data Structures for Stack Implementation**

The most natural data structure for a stack is a linked-list of entries, where the stack updating procedure is readily carried out by entry deletion from the middle of list and entry addition to the head of list. The program for a linked-list stack

is a simple one.

The potential drawback with a list is its linear search time; this might be significant for traces such as data base applications with long average stack distance. However, as we will see in our experiment results, a simple linked-list competes well with other complex data structures; thanks to reference locality, the referenced data block tends to be close to the stack head.

Sophisticated data structures with lower asymptotic time complexities such as binary search trees can be used to implement the stack. Bennett and Kruskal used the leaves of a fixed-structure sparse tree to represent stack entries, and Olken used an AVL balanced search tree to represent stack entries with both the external and internal tree nodes (see [Thompson 87]).

In a binary search tree stack implementation, all data blocks in the left subtree of any node are higher in the stack, while those in the right subtree are lower in the stack. The embedded stack order is the *inorder* traversal of the tree. The tree node containing the currently referenced data block can be quickly found with the hashing table, in which each element in the block lists has a pointer to the corresponding tree node. The stack distance of a data block can be found by walking up the tree to the root and counting the number of tree nodes to the left along the way. This can be done by storing in each tree node the number of nodes in its left subtree [Thompson 87]. Alternatively one can store in each node the number of nodes in the subtree with itself as the root.

In order to know if there is any marker on the left (i.e., ahead in the stack) while walking up the tree from the currently accessed node, each node also stores the number of marker nodes in its left subtree. Finding the left-most (i.e., top-most in the stack) marker node requires a walk down from the root in the tree.

Stack updating is achieved through normal node deletion and insertion in binary search trees; only here node insertion always occurs on the left end of the tree, which corresponds to the stack head.

Because of the extremely biased node insertion, the search tree can quickly degenerate into a linear list. In fact, it becomes the reversed stack, performing



much worse than a simple linked-list stack which benefits from reference locality. A search tree with rebalancing is desirable. Among the many kinds of search trees, three are considered: AVL tree[Wyk 88] and red-black tree (also called 2-4 tree)[Guibas 78, Wyk 88] are balanced trees, and splay tree[Sleator 85a] is a self-adjusting tree.

It was observed[Guibas 78] that AVL and red-black trees have similar performance for basic operations (node rotation) on some sequence of 20,000 random accesses. There was no comprehensive performance comparison for splay trees. We implemented these three data structures and informally tested them with some random input data. For short sequences of random accesses, red-black tree performs the best and is twice as fast as AVL tree; for a long sequence of 200,000 random accesses, the splay tree is the fastest, while AVL tree remains the slowest.

It has been proven[Sleator 85a] that, in terms of *amortized time*, which is defined as the time per operation averaged over a *worst-case sequence* of operations, a splay tree is within a constant factor as efficient as any uniformly balanced tree and any fixed search tree for a sufficiently long sequence of accesses; more interestingly, the time to access an item is approximately the logarithm of one plus the number of distinct items accessed since the last time the given item was accessed[Sleator 85a]. Based on the theoretical results and our preliminary experiments, we choose the splay tree to implement the stack.

## Splay tree

The splay tree a self-adjusting binary search tree. The central idea is splaying, a restructuring heuristic that moves a designated node to the root of a tree through a series of rotations which approximately halves the depths of all nodes along the path. All tree operations, including access, insertion, and deletion, are implemented using splaying[Sleator 85a]. Splaying can be done both bottom-up and top-down. Bottom-up is appropriate if there exists direct access to the node at which splaying is to occur, while top-down if efficient for a to-be-splayed node which has to be searched from the root. Details are described in [Sleator 85a].

We will use the bottom-up splaying for the data block obtained through hashing, and top-down splaying for the left-most marker node in the tree.

### **Separate marker-list**

If there is any marker node to the left of the currently accessed data block in the tree, a top-down search has to be initiated to locate the left-most marker node. Olken[Thompson 87] suggested the use of a separate list to store the markers, sorted by the last access time; the tree at any time only contains valid data blocks. The benefits of this approach are that locating the top-most marker in stack is a constant time operation, and that it is easy to check whether there is a marker above the currently referenced data block by simply comparing their last access times. The drawback is slow invalidation: when a data block is zapped by an invalidation, before we only need to locate the data block node in the tree by hashing and change its flag to make it a marker, taking almost constant time; now we have to delete the node from the tree, and put it into the proper position (by sorted last access time) in the marker-list, not a constant time operation anymore.

We implemented the stack with splay tree using both approaches. There is a better way to implement the separate marker-list, though. Instead of using last access time, we use, equivalently, the actual *stack position* of a marker in the stack as its sorting key in the marker-list. As the stack distance has a much smaller value than the (potentially unbounded) trace length, the space needed to hold a key is less.

The stack updating affects the mark-list in the following way: when a data block is accessed which is not in the stack, then the first marker (if any) in the marker-list is thrown away; when a valid data block is accessed, and its stack distance is bigger than that of the first marker, then the stack distance of this first marker is set to the data block's stack distance, and moved down the marker-list to its proper new position, keeping the list ordered by distance keys. Clearly, the stack distances of all other markers are unchanged.

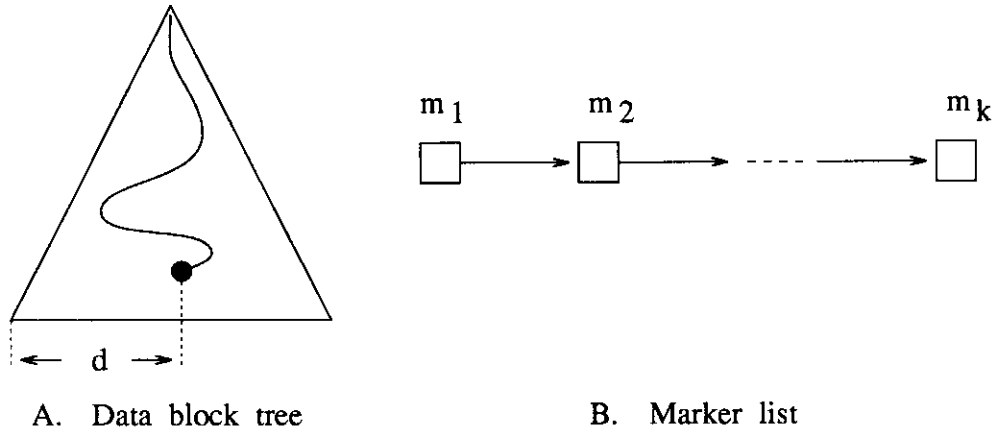


Figure 3.5: Stack distance calculation

The remaining question is how to calculate the stack distance of a valid data block, since walking in the tree can only tell how many nodes are to its left in the tree, i.e., how many valid data entries are above it in the stack. Suppose that a data block is the  $d$ -th node in the inorder traversal of the tree, and the marker-list has elements with stack distances  $m_1 < m_2 < \dots < m_i$ , we want to find the real stack distance  $sd$  for this data block. Let  $m_i < sd < m_{i+1}$ , then among the first  $sd$  stack entries,  $i$  of them are marker entries, and the remaining  $(sd - i)$  are data block entries, as shown in Figure 3.5. We know there are exactly  $d$  data blocks in the range, hence  $sd = d + i$ . From  $m_i < sd = d + i < m_{i+1}$ , we have

$$m_i - i < d \leq m_{i+1} - (i + 1),$$

the criterion for finding  $i$ .

### Arbitrary Set-Associative Algorithm

From the description of the arbitrary set-associative algorithm before, one clearly needs to collect the relevant dirty level variables for all the set lengths from higher entries in the global stack. Each entry above the currently accessed block entry in global stack may contain some of those required dirty levels, and has to be examined. A linear list is therefore a natural data structure choice for the global

stack. Implementation of this algorithm is simple and similar to the linked-list implementation of single set-associative algorithm, but using one global stack instead of many substacks. It involves slightly complicated bit-vector manipulation when the current stack entry under scanning is a marker. Every 0/1-bit vector (such as covering vectors) is just an integer variable, taking up little extra memory than the single set-associative algorithm. The vector operations in the algorithm are done with concise and efficient bit-wise operations of the C programming language.

As above, a two-level hashing scheme is used to at the beginning of each stack search to quickly check whether the referenced data block is in the stack. Since there is no specific set-associative mapping scheme here, we pick an arbitrary hashing function, instead of a congruence set-mapping scheme, for hashing at the first-level. To the arbitrary set-associative algorithm which only processes a trace only, we believe the choice of a hashing function is not crucial.

Stack distance counters are stored in linked-lists, with one list per set length, for up to a maximum of 33 lists. As before, markers are dropped as soon as there is no valid block entries below them in the stack.

### **3.3.3 Simulation Results**

The stack simulation algorithms have simple logic and control flow, and execution time is mostly spent in memory manipulation and data input, not complex CPU operations. Using the Unix code-profiling tool *gprof* indicates that for the arbitrary set-associative algorithm, its disk I/O accounts for about half of the entire running time. Since all trace files have the same data format, and reading trace data is an integral part of any trace-driven simulation, we count I/O time as part of the entire simulation time. The ever-increasing disparity of speed among CPU, main memory, and I/O can make I/O become a more important factor in trace-driven simulation.

## Output Data

All the simulation programs produce exactly the same output, i.e., stack distance distributions, on all input traces and various block-size specifications, verifying not only the correctness of our one-pass algorithm, but also that all the implementations for single set-associative evaluation using different data structures are done right.

## Memory

From the description of arbitrary set-associative algorithm, it is clear that its memory consumption is just a little more than that for the linked-list stack structure implementation of single set-associative algorithm. Specifically, each stack entry uses one more integer field (4 bytes) to hold the covering vector. Using the Unix command *top*, we find that the total program size (code + data + stack) of the arbitrary set-associative algorithm is approximately the same as that of the single set-associative algorithm with the linked-list stack implementation.

The splay-tree stack implementations of single set-associative algorithm have three more fields per each stack entry than the linked-list implementation (one more pointer field and two more integer fields). Their run-time memory size is about 20% more than their linked-list counterpart.

The number of marker entries in the arbitrary set-associative program can also be more than that in the single set-associative program. But experiments show the number of extra marker entries in the algorithm is quite insignificant. One reason is that we keep dropping the marker at the tail of the stack, to prevent their number from growing; another reason: the number of markers might be scarce anyway.

## Running Time

Simulations were run on a lightly loaded Sun SPARCStation 10. As there is little disturbance from other activities on the machine, the measurements were stable

Table 3.1: Simulation times on FFT (including i/o)

b	mul	sin.link	sin.splay	sin.mlist
13	86.9	764.4	764.0	765.8
12	86.1	800.3	803.3	805.1
11	87.9	853.0	849.2	866.9
10	91.8	899.3	904.3	901.7
9	96.7	929.9	936.6	940.5
8	100.2	1001.7	991.2	985.5
7	108.3	1017.1	1023.1	1030.3
6	107.4	1064.8	1071.1	1087.1
5	113.4	1111.8	1120.7	1147.0
4	123.4	1171.8	1176.8	1204.8
3	118.2	1249.2	1250.0	1301.8
2	111.7	1297.7	1314.7	1317.8
1	116.1	1373.1	1393.0	1395.1

Block-size =  $2^b$  bytes. Run-time in seconds.

and repeatable. We use the Unix *time* command to measure the execution times of each simulation, and the real times are very close to the sums of user times and system times, due to light load on the test machine. The arbitrary set-associative algorithm runs much faster than the single set-associative algorithm on all three traces.

Tables 3.1, 3.2, 3.3, and 3.4 illustrate the running times of the various implementations of the algorithms for the three traces. These results are from tests done for a typical CPU. For each trace, we randomly selected a number of CPUs and did stack simulation on them; their results were nearly identical. It is probably due to the fact that the traces were produced in a very symmetrical way (see [Cherian 89]).

The tests are done for a variety of block sizes. The first columns (b) indicate the base-2 logarithmic values of block sizes. The second columns (mul) are the running times of the arbitrary set-associative algorithm; rest columns are the running times of the single set-associative algorithm implemented with various

Table 3.2: Simulation times on FFT (excluding i/o)

b	mul	sin.link	sin.splay	sin.mlist
13	64.8	742.3	741.9	743.7
12	64.0	756.1	759.1	760.9
11	65.8	786.8	783.0	800.7
10	69.7	811.0	816.0	813.4
9	74.6	819.5	826.2	830.1
8	78.1	869.2	858.7	853.0
7	86.2	862.6	868.6	875.8
6	85.3	888.2	894.5	910.5
5	91.3	913.1	922.0	948.3
4	101.3	951.0	956.0	984.0
3	96.1	1006.4	1007.2	1059.0
2	89.6	1032.8	1049.8	1052.9
1	94.0	1086.1	1106.0	1108.1

Block-size =  $2^b$  bytes. Run-time in seconds.

data structures (sin.link for linked-list, sin.splay for splay tree, and sin.mlist for splay tree with separate marker list).

Tables 3.1, 3.3, and 3.4 include the disk i/o time of trace-reading in the total simulation time for the three traces, and Table 3.2 excludes that from the simulation time of FFT trace. Compare Table 3.1 and Table 3.2, we see that i/o played a small role in the simulations. Overall, our arbitrary set-associative algorithm ran approximately ten times faster than all implementations of the single set-associative algorithm.

For the single set-associative algorithm, the linked-list implementation with hashing performs best, confirming previous studies on uniprocessor stack simulation implementations [Thompson 87]. For the splay tree version, we did a faithful implementation of the original data structure, not dealing specially with the fact that all insertions occur at the left end. Specializing the implementation to exploit this characteristic might speed up execution, but the potential gain is probably small.

Table 3.3: Times on Simple (inc. i/o)

b	mul	sin.link
13	296.5	3231.6
12	302.1	3436.1
11	318.3	3644.3
10	350.3	3897.3
9	409.0	4131.6
8	428.0	4304.5

Run-time in seconds.

Ideally, one wants to implement the stack with all other balanced tree structures (AVL, red-black) and compare their performances. However, our simulation results indicate that any improvement using sophisticated data structures will be minimal and hardly worth the effort.

For completeness, we also ran the algorithms on randomly generated long synthetic trace data. The same magnitude speed-up in simulation time on the part of arbitrary set-associative algorithm over the single set-associative algorithm still holds. This indicates the stability of the algorithm’s performance on different traces.

As most trace files are quite large, they are often stored in compressed format and are uncompressed on-the-fly for a simulation. We piped the results of uncompressing some \*.Z files into the various simulation programs and did not see any noticeable change in simulation time. The reason is that the “uncompress” program runs faster than the simulations, the overall speed of the pipeline still depends on the simulation. Therefore using compressed trace files will get almost the same speed-up result for the above simulations.

We did one more comparison. Instead of using one process to run the single set-associative algorithm for each set length, we lumped all of them into one program, which controls multiple independent groups of stacks, one per each set



Table 3.4: Times on Weather (inc. i/o)

b	mul	sin.link
13	350.2	3798.2
12	373.6	4024.1
11	408.7	4298.1
10	483.1	4599.2
9	567.4	4912.1
8	582.4	5167.4
7	584.4	5416.7

Run-time in seconds.

length. On each trace reference, the program sequentially does stack processing on all the stacks. This considerably reduces the I/O time of the single set-associative program. But its memory consumption is, however, much larger than that of the arbitrary set-associative simulation. This excessive space requirement can become a big burden when testing large traces with many distinct addresses. While testing the Weather trace, this kind of simulation failed to finish in a reasonable amount of time.

As a one-pass evaluation method, the arbitrary set-associative algorithm can be run on-the-fly, i.e., simultaneously with a trace generating program[Hill 89], and the saving of a long trace data onto disk can be avoided. This kind of on-the-fly simulation is especially useful, when the amount of *distinct* addresses in the trace can be safely accommodated by the main memory, but the entire trace is extremely long, in which case the required disk space could be overwhelming. As the disk space needed for trace storage is becoming too large even for a short operating period of time of a moderately fast computer nowadays, investigation of on-the-fly techniques is becoming necessary[Baer 91]. Our arbitrary set-associative algorithm is also an applicable tool in this regard.

## On Concurrent Simulation

One might consider concurrently running all the simulation programs of the single set-associative algorithm, using the Unix piping mechanism to pass trace data sequentially from the first simulation program through other simulation programs, relieving them of the necessity of getting trace data through slow disk I/O.

The problem with this concurrent-execution approach is again the enormous amount of main memory required. The memory demand of each single set-associative program is approximately equal: each uses the same number of stack entries for valid data blocks; and the difference in the number of stack entries for invalid data blocks (markers) is small, since the number of markers is kept small in any stack by the dropping of markers from the tail of stack. For  $K$  set lengths, the main memory demand of the concurrent simulation is approximately  $K$  times that of the sequential execution. For small block sizes (hence large set length ranges), that becomes a serious burden on the testing machine's memory system. Excess demand on main memory can cause frequent memory paging and context switching in virtual memory, generating new disk I/O for paging and swapping. Consequently, the real running times of the simulations would be much larger than the sums of their respective user times and system times. Our test on the FFT trace found that the overall running time of this kind of concurrent simulation was comparable to that of the sequential simulation.

As our arbitrary set-associative algorithm uses almost the same amount of memory space as the single set-associative algorithm, while one does concurrent simulations with the single set-associative algorithm on one trace, we can instead run concurrent simulations on different traces with the arbitrary set-associative algorithm—using the same resources of CPU, memory, and time to simulate more traces.

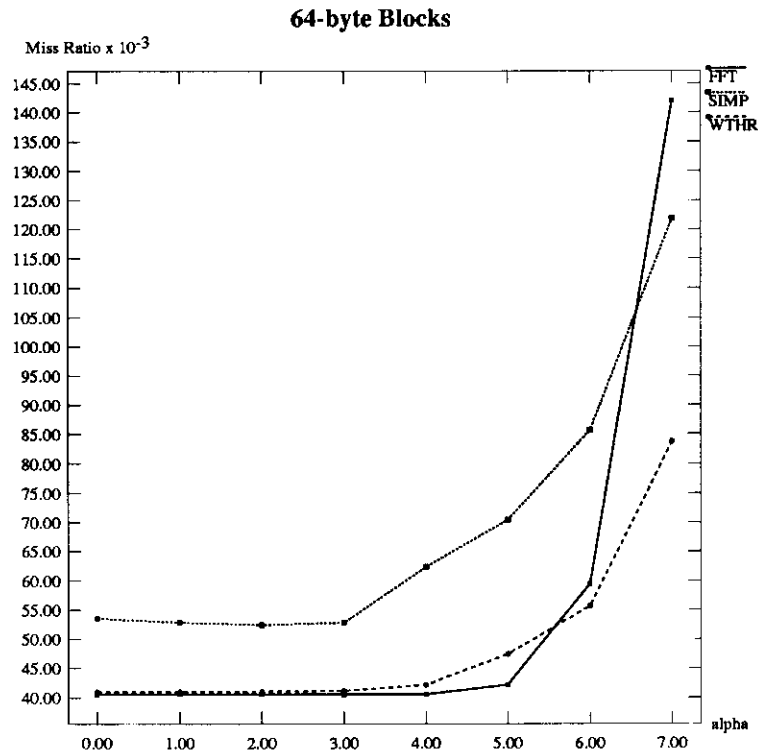


Figure 3.6: Miss ratio v.s.  $\alpha$  for a cache of 128 64-byte blocks.

### Example of Simulation Results

With the new simulation algorithm, we can get miss ratios for arbitrary cache size (in number of fixed-size blocks) and arbitrary set-associative mapping function in one-pass trace processing. Figure 3.6, illustrates, regarding a particular CPU for all traces, the relationship between miss ratio and set length  $\alpha$  on a cache of 128 blocks, each block with 64 data bytes. For a given cache size, generally (but not always) the fully associative mapping has a lower miss ratio than a set-associative mapping; but occasionally some set-associative mapping has the same or even lower miss ratio than the fully-associative, such as  $\alpha = 3$  on trace FFT and  $\alpha = 2$  on trace SIMP as shown by Figure 3.6.

Various performance quantities can be obtained using the stack distance distribution data obtained from the efficient one-pass simulation. For example, given the cache capacity, one might need to find the optimal set-associative mapping

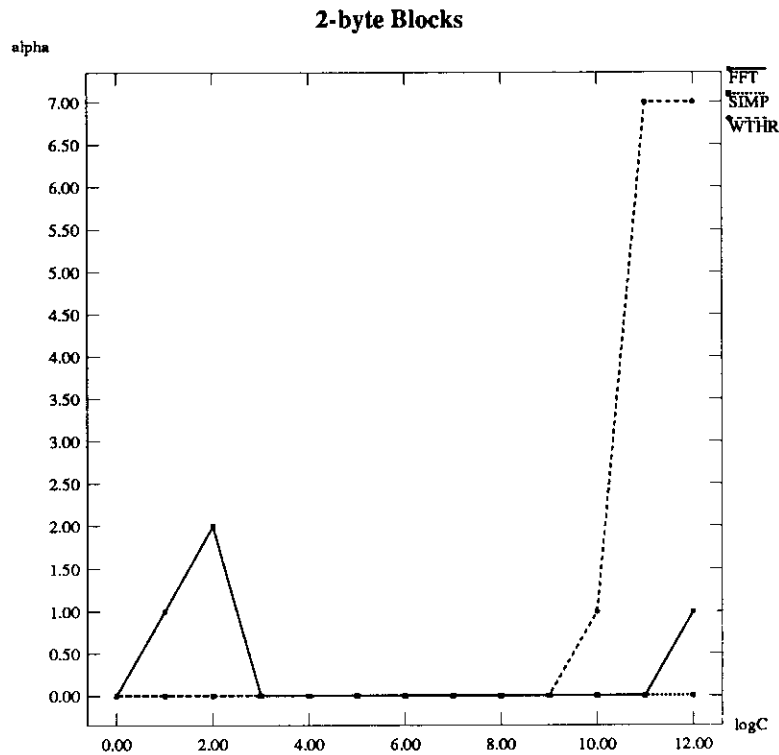


Figure 3.7: Optimal  $\alpha$  v.s. Cache Size for 2-byte Blocks.

scheme that has the lowest miss ratio. Figure ?? through Figure ?? illustrate, regarding the same CPU as above, the relationship between the set length  $\alpha$  that yields the minimum miss ratio, and the cache size  $C$  in number of blocks (the  $x$ -axis uses base-2  $\log C$ ). Each figure is for a specific block size, ranging from 2-byte block to 4096-byte block. When there is a tie in minimum miss ratio, we break the tie by choosing the  $\alpha$  with a larger value; for a fixed cache size, more sets (i.e., larger  $\alpha$ ) provide quicker cache searching.

### 3.4 Summary

We show that efficient stack analysis can be extended to arbitrary two's power congruence set-associative mapping for LRU caches on multiprocessors. For block addresses between 0 and  $2^k - 1$ , instead of running stack evaluation on the same

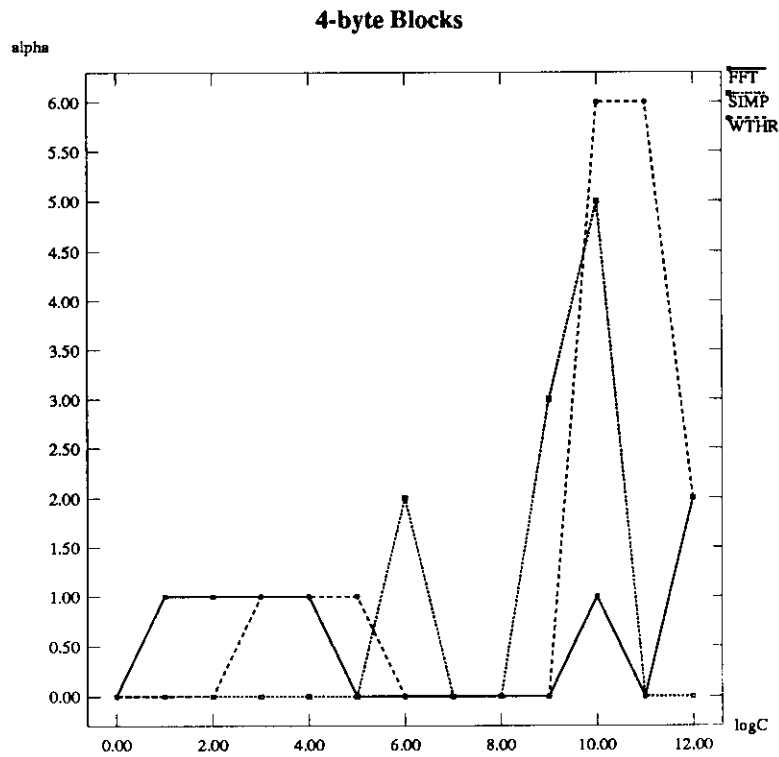


Figure 3.8: Optimal  $\alpha$  v.s. Cache Size for 4-byte Blocks.

trace  $k + 1$  times for all the possible set lengths, one run of stack evaluation on the trace can give us the same hit ratio function for all set lengths. Thanks to the locality property of CPU access in real applications, the necessity of using a simple linear list stack structure for the arbitrary set-associative evaluation does not compromise its simulation time. Simulation on real multiprocessor trace data show an order-of-magnitude speed up by our algorithm in simulation time.

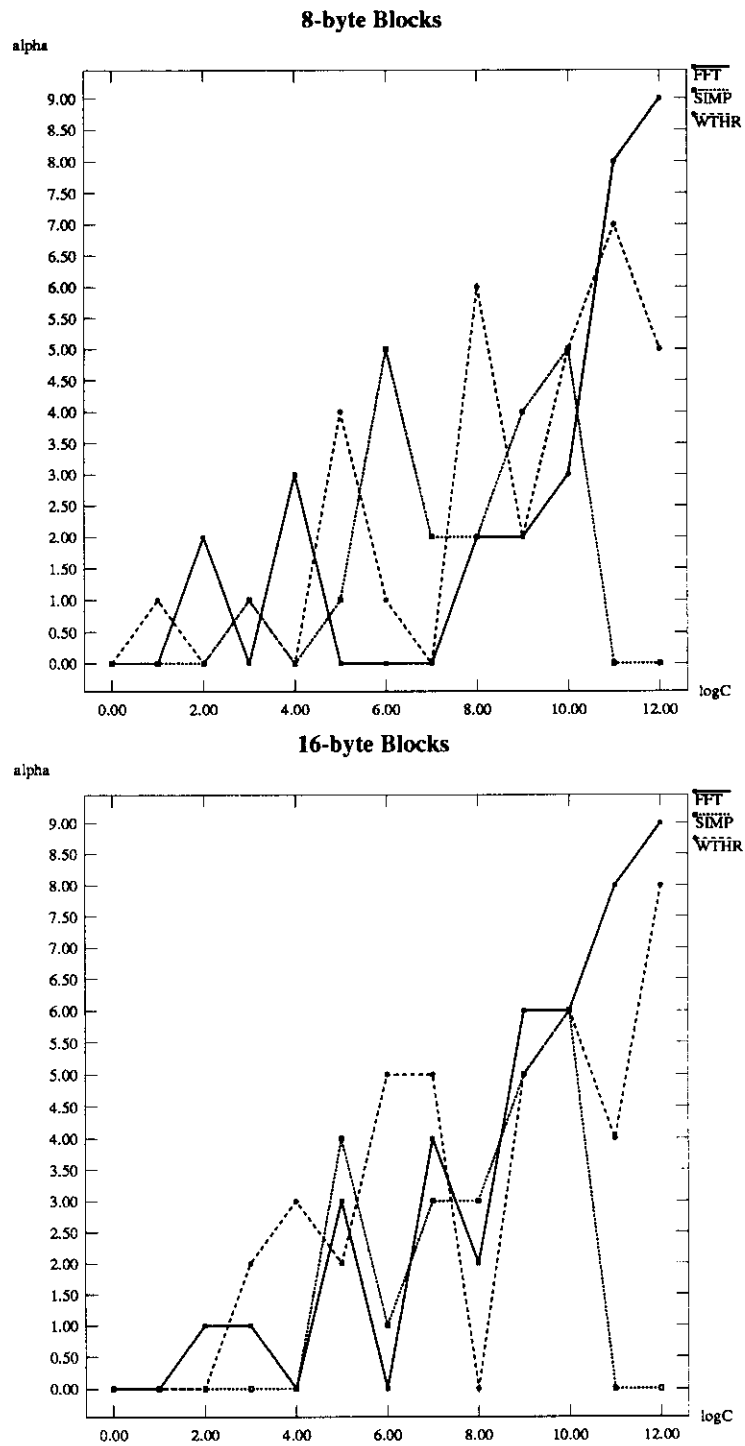


Figure 3.9: Optimal  $\alpha$  v.s. Cache Size for 8 and 16-byte Blocks.

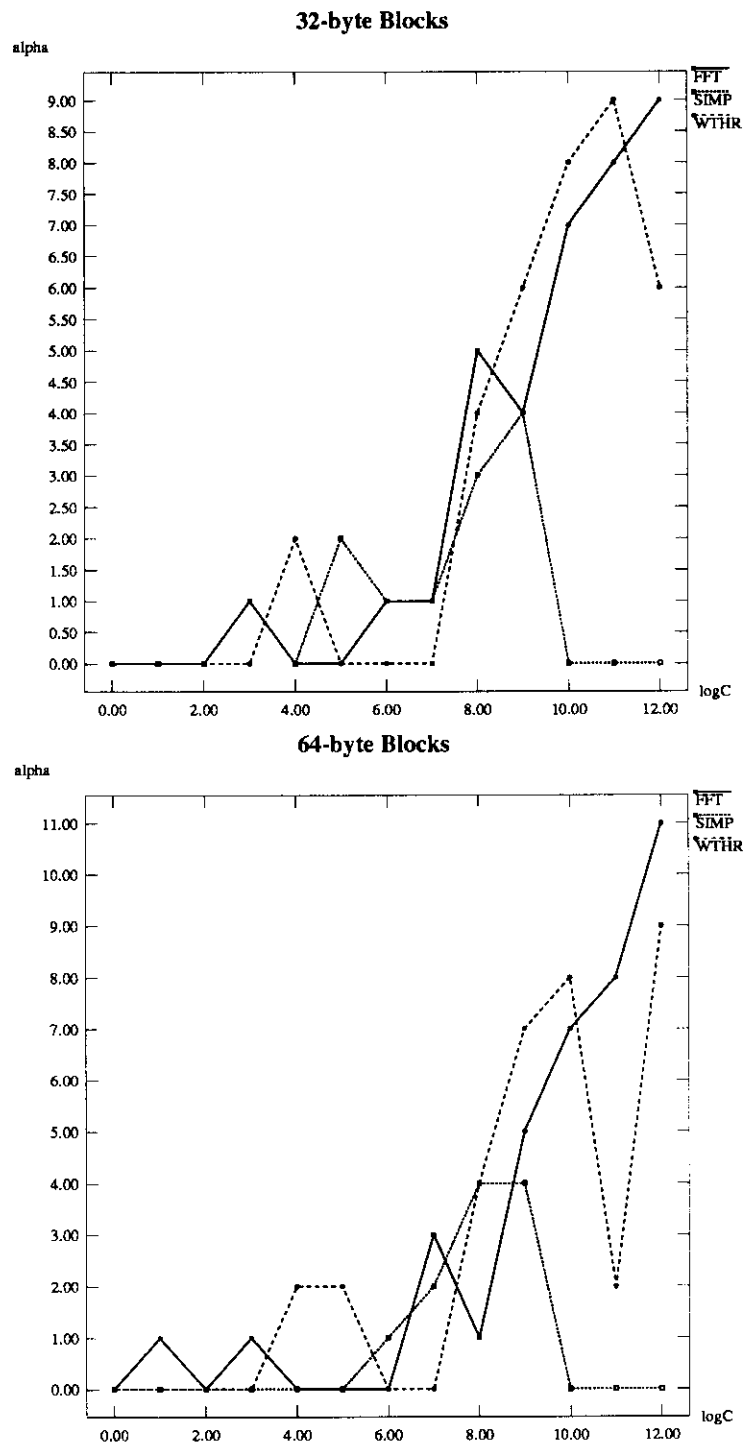


Figure 3.10: Optimal  $\alpha$  v.s. Cache Size for 32 and 64-byte Blocks.

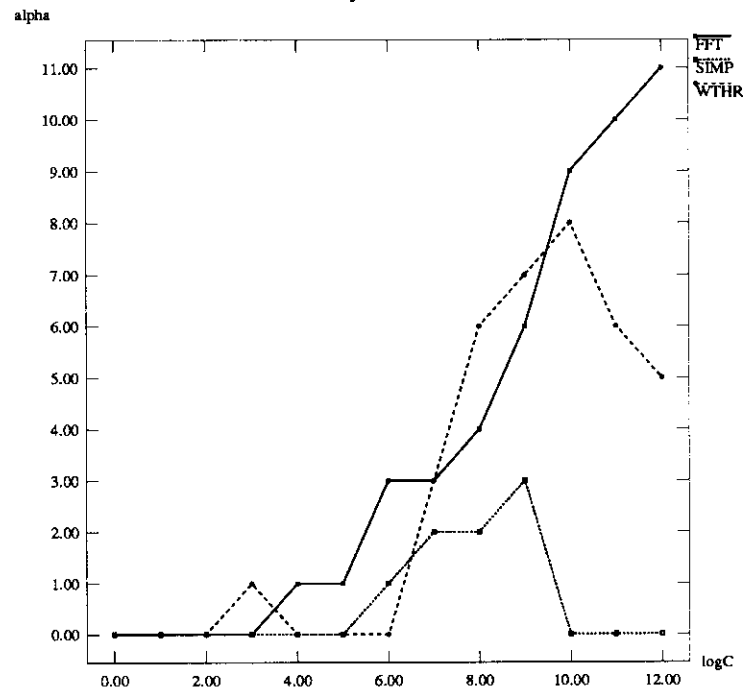
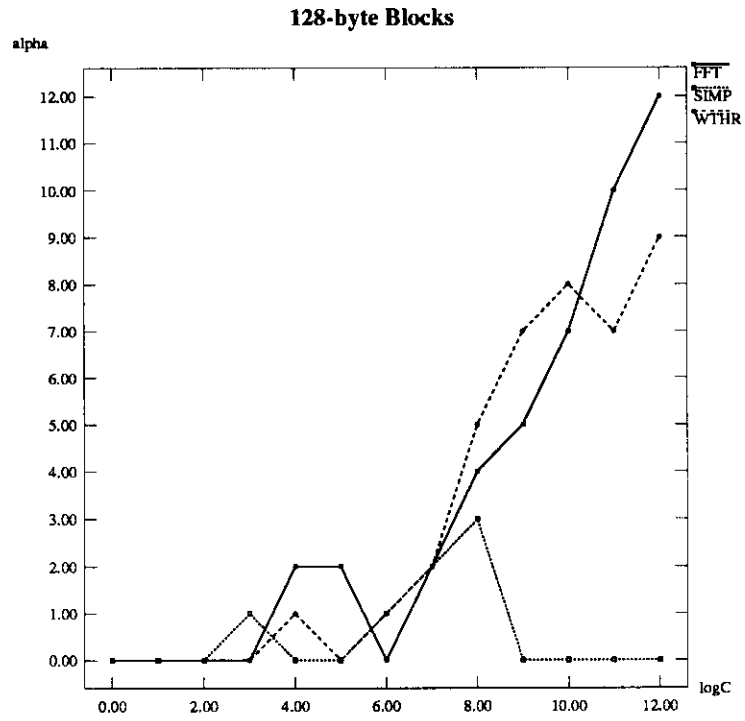


Figure 3.11: Optimal  $\alpha$  v.s. Cache Size for 128 and 256-byte Blocks.



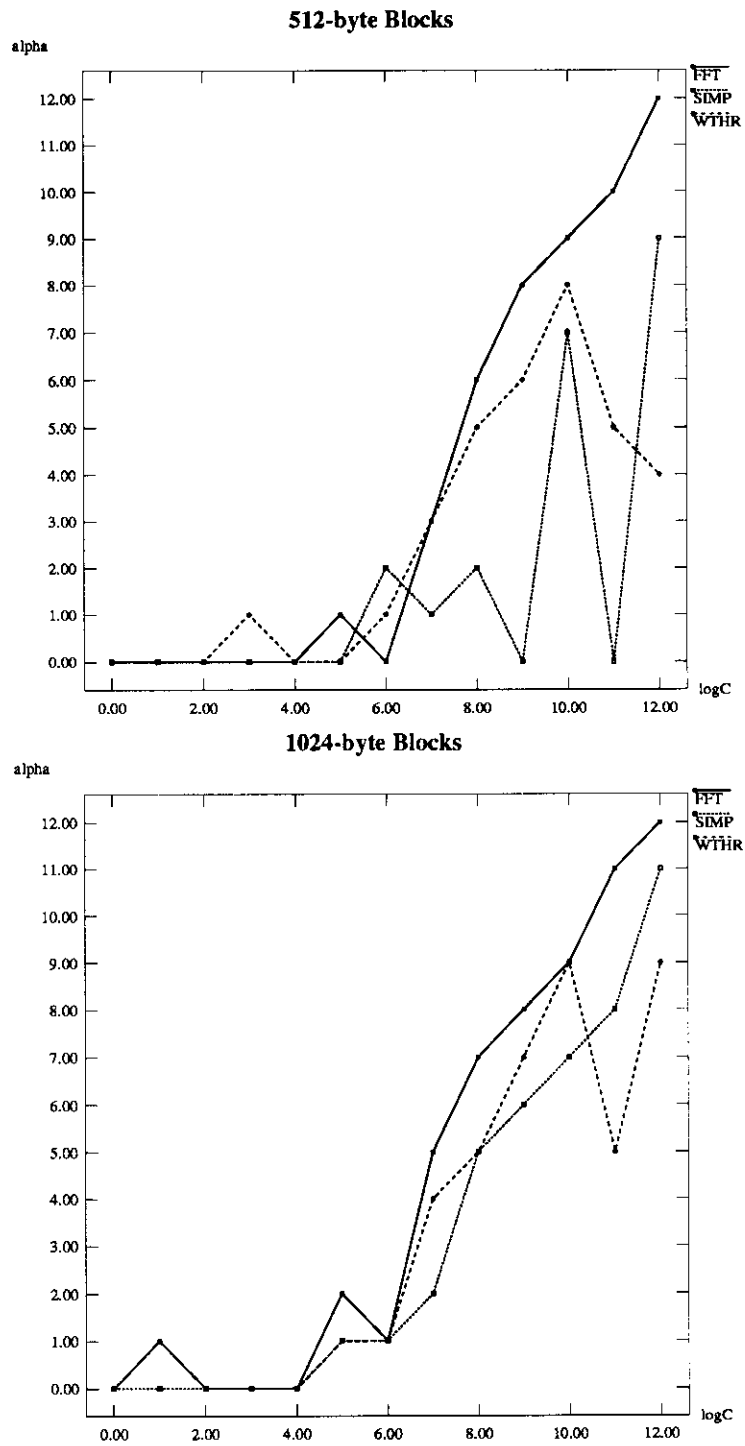


Figure 3.12: Optimal  $\alpha$  v.s. Cache Size for 512 and 1024-byte Blocks.

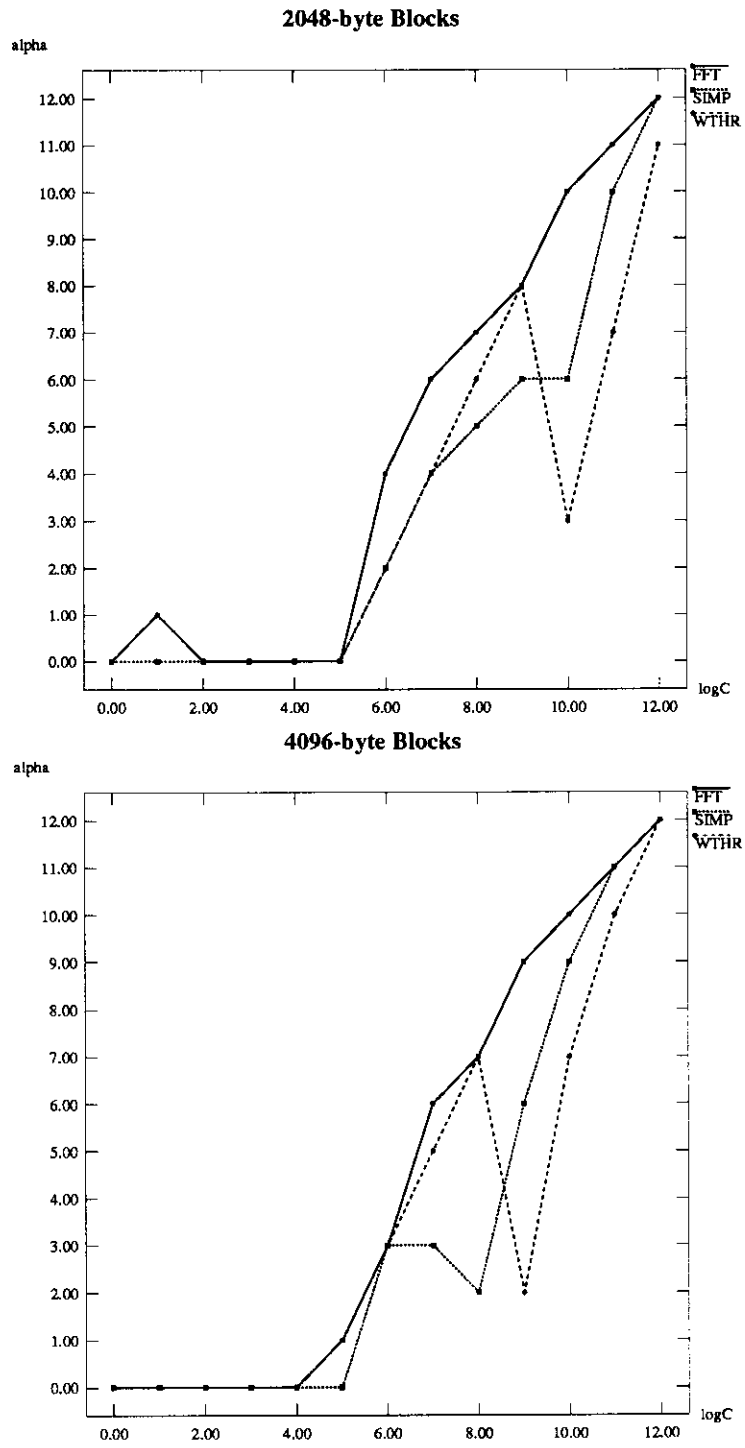


Figure 3.13: Optimal  $\alpha$  v.s. Cache Size for 2048 and 4096-byte Blocks.

# Chapter 4

## All Set-Associative Evaluation

**Abstract.** It is known that the LRU block replacement algorithm allows efficient stack evaluation for all two's power congruence-mapping set associativities in one-pass processing of memory access traces, by using just a single stack. It is natural to ask if any other stack algorithm permits such an efficient evaluation. We show that LRU is the only such stack algorithm among all stack algorithms which do not base replacement decisions on the numerical values of block address. This conclusion can be generalized from the two's power congruence-mapping set associative scheme to any set associative scheme with more than two different mapping schemes. If a set mapping scheme includes all possible groupings of data blocks, then LRU is the only such stack algorithm among all stack algorithms, regardless of whether or not they base replacement decisions on the numerical values of block address.

### 4.1 Introduction

Generally, stack algorithms require that stack evaluation be applied separately for each value of the set length  $\alpha$ . With  $0 \leq \alpha \leq k$ ,  $k + 1$  runs of trace analysis are needed. Mattson, et al. [Mattson 70] showed that the LRU block replacement algorithm can be evaluated for all congruence-mapping set associativities in one-pass, using a single stack.

For LRU replacement, the global stack  $S_{t-1}$  is the list of previously (from

time 0 to  $t - 1$ ) accessed blocks in decreasing order by the time of their most recent reference; the substack  $S_{t-1}(i, \alpha)$  is the list of previously accessed blocks that are in the  $i$ th set of set length  $\alpha$ , also ordered by the decreasing time of their most recent reference.  $S_{t-1}(i, \alpha)$  can be recovered from  $S_{t-1}$ , by taking in order all the stack entries of  $S_{t-1}$  that are in the  $i$ th set of set length  $\alpha$ . Therefore, the list of the stack entries above  $x_t$  in *any* substack  $S_{t-1}(i, \alpha)$  would constitute a sublist of that of the stack entries above  $x_t$  in the global stack  $S_{t-1}$ . When searching for  $x_t$  on the global stack  $S_{t-1}$  is done, either by finding  $x_t$  or by reaching the end of the the global stack, one is certain that the entries in substack  $S_{t-1}(i, \alpha)$ , which would be scanned in  $S_{t-1}(i, \alpha)$  if the search were done directly on  $S_{t-1}(i, \alpha)$ , are all scanned in the right order. In other words, a search for  $x_t$  in the global stack  $S_{t-1}$  includes simultaneous searching of all the substacks  $S_{t-1}(i, \alpha)$ ,  $0 \leq i \leq 2^\alpha - 1, 0 \leq \alpha \leq k$ .

To do one-pass evaluation of LRU algorithm for all congruence-mapping set associativities, only the global stack is needed. At time  $t$ , search stack  $S_{t-1}$  until  $x_t$  is found or  $S_{t-1}$  is exhausted; at each entry  $s_{t-1}(i)$  of  $S_{t-1}$ , increment the stack distance counter  $\Delta_t^\alpha$  for every set length  $\alpha$  such that  $s_{t-1}(i)$  is in the same set as  $x_t$  ( $s_{t-1}(i) \equiv x_t \pmod{2^\alpha}$ ). If  $x_t$  is not found,  $\Delta_t^\alpha$  is set to  $\infty$  for every set length  $\alpha$ .

Notice that congruence-mapping is an unnecessary assumption. The above one-pass evaluation can be carried out for LRU on *any* set mapping scheme.

In the next section we shall show that LRU is the only stack algorithm among *a wide range of stack algorithms* that permits such a one-pass trace evaluation for all set associativities.

## 4.2 All Set-Associativity Evaluation and LRU

To clarify what we mean by “a wide range of stack algorithms”, let us define the following subset of stack algorithms:

**Definition 4.1** *An address-independent algorithm is a stack algorithm whose priority list is independent of the numerical values of block addresses.*

LRU, LFU, MIN are examples of address-independent stack algorithms. Their priority lists depend entirely on the *time aspects* of block usage, e.g. when or how often a block has been used, or will be used. They are not related to the actual numerical values of the block addresses. On the other hand, both the least-transition-probability (LTP, choosing the block least likely to be referenced after the current block) and the least-next-reference (LNR, choosing the block with the largest expected next reference time) algorithms are address-dependent. Their priority lists depend on a transition probability matrix  $\Pi = \{\pi_{bc}\}$ , where  $\pi_{bc}$  is a fixed probability that block  $c$  is referenced right after block  $b$  is. The probability  $\pi_{bc}$  dictates that the address of the currently accessed block be taken into consideration. Most commonly used stack algorithms are address-independent algorithms.

From previous discussion, it is clear that in order to be able to determine all stack distances  $\{\Delta_t^\alpha \mid 0 \leq \alpha \leq k\}$  in one scan of the global stack  $S_{t-1}$ , a stack algorithm must satisfy the following:

**Property** At any access time  $t$ , scanning global stack  $S_{t-1}$  for  $x_t$  is equivalent to simultaneously scanning all substacks  $\{S_{t-1}(i, \alpha) \mid 0 \leq i \leq 2^\alpha - 1, 0 \leq \alpha \leq k\}$ .

LRU obviously has this property. We show this property necessitates that for any address-independent stack algorithm, reference to one block must not alter the relative ordering of any other blocks in its global stack during stack updating, and the only stack algorithm that preserves the relative ordering of non-accessed blocks in its global stack is LRU. Therefore, LRU is *the only* address-independent stack algorithm that satisfies the aforementioned property.

For easy exposition later on, we introduce a few more definitions.

**Definition 4.2** *A stack algorithm that can be evaluated in one-pass trace processing for all congruence-mapping set associativities, using a single stack, is called a congruence-mapping arbitrary set-associative stack algorithm.*

**Definition 4.3** For two blocks  $a, b$  in a stack  $S$ , denote  $a \xrightarrow{S} b$  if  $a$  is above  $b$  in stack  $S$ , i.e.,  $a$ 's stack distance is less than  $b$ 's stack distance.

Apparently,  $\xrightarrow{S}$  defines a partial relation on blocks in stack  $S$ . When stack  $S$  is changed by updating, the relation changes accordingly. It is undefined if one of the two blocks is not in  $S$ .

We first derive two conditions any congruence-mapping arbitrary set-associative stack algorithm has to obey, then strengthen them to obtain a necessary condition that an address-independent congruence-mapping arbitrary set-associative stack algorithm must satisfy, and finally conclude that LRU is the only address-independent stack algorithm that meets this necessary condition.

For brevity, we use  $\implies$  to denote “logically implying”.

First of all, for any set length  $\alpha$ , accessing one block should not alter the stack of any other set which the currently accessed block does not belong to:

**Lemma 4.1** Let  $[x_t]_\alpha$  denote the set that  $x_t$  is in, for set length  $\alpha$ :

$$[x_t]_\alpha \stackrel{\text{def}}{=} \{ p \mid p \equiv x_t \pmod{2^\alpha} \}$$

For any stack algorithm, after accessing  $x_t$ ,

$$S_t(i, \alpha) = S_{t-1}(i, \alpha)$$

for all combinations of  $0 \leq i \leq 2^\alpha - 1$ ,  $i \not\equiv x_t \pmod{2^\alpha}$ ,  $0 \leq \alpha \leq k$ .

**Proof.** Consider a fixed set length  $\alpha$ . For any stack algorithm, processing an access  $x_t$  at time  $t$  only requires examining the part of cache space allocated for set  $[x_t]_\alpha$ , as this is the only place where  $x_t$  can possibly be found. If  $x_t$  is there, access it; if  $x_t$  is not there, bring it in and access it, and in this case the stack algorithm makes a replacement decision if the part of cache space for set  $[x_t]_\alpha$  is already full. In both cases, none of the cache spaces allocated for other sets is searched. Hence accessing  $x_t$  should only change the substack  $S_{t-1}(x_t \bmod 2^\alpha, \alpha)$ ; it does not affect any other substack  $S_{t-1}(i, \alpha)$ , where  $i \not\equiv x_t \pmod{2^\alpha}$ . In other words,

$$S_t(i, \alpha) = S_{t-1}(i, \alpha)$$

for  $0 \leq i \leq 2^\alpha - 1$ ,  $i \not\equiv x_t \pmod{2^\alpha}$ . Since  $\alpha$  is chosen arbitrarily, this holds for every  $\alpha$  between 0 and  $k$ .  $\square$

Secondly, the ordering of blocks in any substack must always be preserved in the global stack:

**Lemma 4.2** *Suppose two different blocks  $p_1$  and  $p_2$  are in the same set  $[p]_\alpha$  of some set length  $\alpha$  ( i.e.,  $p_1 \equiv p_2 \equiv p \pmod{2^\alpha}$ ). For a congruence-mapping arbitrary set-associative stack algorithm, the relative ordering of  $p_1$  and  $p_2$  in the substack  $S_t(p \bmod 2^\alpha, \alpha)$  is preserved in the global stack  $S_t$  at any time  $t$*

$$p_1 \xrightarrow{S_t(p \bmod 2^\alpha, \alpha)} p_2 \implies p_1 \xrightarrow{S_t} p_2.$$

**Proof.** The proof is by contradiction. Suppose at some time  $t$  and for some specific  $\alpha$ , we have two different blocks  $p_1, p_2$  in set  $[p]_\alpha$  such that

$$p_1 \xrightarrow{S_t(p \bmod 2^\alpha, \alpha)} p_2, \text{ and } p_2 \xrightarrow{S_t} p_1.$$

Let  $x_{t+1} = p_1$ . Then the stack distance  $\Delta_{t+1}^\alpha$  obtained from scanning substack  $S_t(p \bmod 2^\alpha, \alpha)$  is different from that obtained from scanning the global stack  $S_t$ . The reason is obvious: in substack  $S_t(p \bmod 2^\alpha, \alpha)$  block  $p_2$  is not counted in calculating  $\Delta_{t+1}^\alpha$ , since  $p_2$  is below  $p_1$ ; while in the global stack  $S_t$  block  $p_2$  is counted, since  $p_2$  is above  $p_1$  and  $p_2$  does belong to the same set  $[p]_\alpha$  as  $p_1$ . In this case congruence-mapping arbitrary set-associative evaluation of the single global stack does not yield the correct stack distance for set length  $\alpha$ .  $\square$

With these two lemmas, we deduce that for an address-independent stack algorithm to be a congruence-mapping arbitrary set-associative algorithm, after an arbitrary access, the relative ordering of any non-accessed two blocks in the global stack must be preserved after stack updating:

**Theorem 4.1** *For any address-independent congruence-mapping arbitrary set-associative stack algorithm, if accessing block  $x_t$  at time  $t$ ,*

$$p_1 \xrightarrow{S_{t-1}} p_2 \implies p_1 \xrightarrow{S_t} p_2.$$

*holds for any two blocks  $p_1, p_2$  such that  $p_1 \neq x_t, p_2 \neq x_t$ .*

**Proof.** Since the case of  $p_1 = p_2$  is trivial, we assume that  $p_1, p_2, x_t$  are all different.

We first prove the theorem for the special case of  $p_1 = p_1^0, p_2 = p_2^0, x_t = x_t^0$ , where  $p_1^0 \neq p_2^0 \neq x_t^0 \neq p_1^0$ , and

$$\exists \text{ set length } \alpha \text{ with } 0 < \alpha \leq k, \text{ such that } [p_1^0]_\alpha = [p_2^0]_\alpha \neq [x_t^0]_\alpha$$

An example is  $p_1^0 = 1, p_2^0 = 3, x_t^0 = 2, \alpha = 1$ .

Since  $p_1^0 \xrightarrow{S_{t-1}} p_2^0$ , and  $p_1^0, p_2^0$  belong to the same set  $[p_1^0]_\alpha$ ,

$$p_1^0 \xrightarrow{S_{t-1}(p_1^0 \bmod 2^\alpha, \alpha)} p_2^0 \quad (4.1)$$

because if otherwise, by Lemma 4.2, we would have  $p_2^0 \xrightarrow{S_{t-1}} p_1^0$ , contradicting the assumption. Since  $x_t^0$  is not in the same set  $[p_1^0]_\alpha$  that  $p_1^0, p_2^0$  both belong to, by Lemma 4.1,

$$S_t(p_1^0 \bmod 2^\alpha, \alpha) = S_{t-1}(p_1^0 \bmod 2^\alpha, \alpha) \quad (4.2)$$

Therefore, from (4.1) and (4.2),

$$p_1^0 \xrightarrow{S_t(p_1^0 \bmod 2^\alpha, \alpha)} p_2^0 \quad (4.3)$$

Apply Lemma 4.2 to (4.3), we have

$$p_1^0 \xrightarrow{S_t} p_2^0 \quad (4.4)$$

Now let's consider arbitrary blocks  $p_1, p_2, x_t$  with  $p_1 \neq p_2 \neq x_t \neq p_1$ . Suppose the original access trace is  $X = x_1, x_2, \dots, x_L$ , and before accessing  $x_t$ , we have

$$p_1 \xrightarrow{S_{t-1}} p_2 \quad (4.5)$$

We want to show that  $p_1 \xrightarrow{S_t} p_2$ .

Define a permutation  $P$  on blocks:  $P = (p_1, p_1^0)(p_2, p_2^0)(x_t, x_t^0)$ . That is,  $p_1$  is switched with  $p_1^0$ ,  $p_2$  is switched with  $p_2^0$ , and  $x_t$  is switched with  $x_t^0$ ; all other



blocks remain unchanged:

$$P(b) = \begin{cases} p_1^0 & \text{if } b = p_1 \\ p_1 & \text{if } b = p_1^0 \\ p_2^0 & \text{if } b = p_2 \\ p_2 & \text{if } b = p_2^0 \\ x_t^0 & \text{if } b = x_t \\ x_t & \text{if } b = x_t^0 \\ b & \text{otherwise} \end{cases}$$

$P$  is a valid permutation, as both  $(p_1, p_2, x_t)$  and  $(p_1^0, p_2^0, x_t^0)$  are groups of three distinct numbers. Let the result of performing the permutation  $P$  on  $X$  be  $X^0 = P(X) = x_1^0, x_2^0, \dots, x_L^0$ . We do stack evaluation on the new trace  $X^0$ , with  $S_t^0$  being the global stack for trace  $X^0$  at time  $t$ .

Because the stack algorithm is address-independent, at any access time  $t'$ , stack  $S_{t'}^0$  is the result of performing permutation  $P$  on  $S_{t'}$ ,

$$S_{t'}^0 = P(S_{t'}) \text{ for } \forall t' > 0 \quad (4.6)$$

Equivalently,  $S_{t'}$  is the result of performing the inverse permutation  $P^{-1} = P$  on  $S_{t'}^0$ ,

$$S_{t'} = P(S_{t'}^0) \text{ for } \forall t' > 0 \quad (4.7)$$

Apply transformation  $P$  to Equation (4.5),

$$P(p_1) \xrightarrow{P(S_{t-1})} P(p_2) \quad (4.8)$$

by (4.6) and (4.8),

$$p_1^0 \xrightarrow{S_{t-1}^0} p_2^0 \quad (4.9)$$

from (4.4) and (4.9), accessing  $x_t^0$  preserves,

$$p_1^0 \xrightarrow{S_t^0} p_2^0 \quad (4.10)$$

apply transformation  $P^{-1} = P$  to (4.10),

$$P(p_1^0) \xrightarrow{P(S_t^0)} P(p_2^0) \quad (4.11)$$

from (4.7) and (4.11), we arrive at

$$p_1 \xrightarrow{S_t} p_2 \tag{4.12}$$

□

The logic of the above theorem proof can be described by the following diagram

$$\begin{array}{ccc}
 & \text{Theorem 4.1} & \\
 p_1 \xrightarrow{S_{t-1}} p_2 & \Longrightarrow & p_1 \xrightarrow{S_t} p_2 \\
 \Downarrow \text{Lemma 4.2} & & \Uparrow \text{Lemma 4.2} \\
 p_1 \xrightarrow{S_{t-1}(f(p_1),f)} p_2 & \Longrightarrow & p_1 \xrightarrow{S_t(f(p_1),f)} p_2 \\
 & \text{Lemma 4.1} & 
 \end{array}$$

where in order to prove the conclusion for the global stack, we find a substack (whose existence is guaranteed by the address-independency of the stack algorithm) which contains  $p_1$  and  $p_2$ , the two blocks under consideration, but does not contain  $x_t$ , the currently accessed block. According to Lemma 4.2, the relative ordering of  $p_1$  and  $p_2$  in this substack is the same as that in the global stack; using Lemma 4.1, their relative ordering in the substack is unchanged by the current reference to  $x_t$ ; again by Lemma 4.2, their relative ordering in the global stack is therefore unchanged by the current reference to  $x_t$ .

It is easy to see why some of the most familiar address-independent stack algorithms like LFU and MIN are not congruence-mapping arbitrary set-associative stack algorithms. The LFU stack updating may send its previous top entry block down the stack across one or more entries, reversing the relative ordering of the first two blocks on stack. The same thing may happen for MIN stack updating.

Using this theorem, it is a simple matter to conclude the following result.

**Corollary 4.1** *An address-independent, congruence-mapping arbitrary set associative stack algorithm is LRU.*

**Proof.** For any stack algorithm, the currently accessed block always goes to the top of the stack after stack updating. By Theorem 4.1, stack updating of an address-independent congruence-mapping arbitrary set-associative stack algorithm consists of moving the currently accessed block to stack top while keeping the orderings among all other blocks in stack unchanged. This is exactly the stack updating of LRU algorithm. Hence this address-independent congruence-mapping arbitrary set-associative stack algorithm is LRU.  $\square$

So far we have been concerned with congruence mapping set associative schemes, which constitute a rather restricted case. The above conclusion can be extended to more general set mapping schemes.

**Definition 4.4** *Let  $\mathcal{F}$  be a set of set-mapping functions, which are real-valued functions defined on block numbers. That is,  $\mathcal{F} = \{ f \mid f : N \mapsto R \}$ , where  $N = \{ 0, 1, 2, \dots, 2^k - 1 \}$ ,  $R =$  set of real numbers. A stack algorithm, which can be evaluated in one-pass trace processing for all set-mappings in  $\mathcal{F}$  while only using a single stack, is called an  $\mathcal{F}$ -mapping arbitrary set-associative stack algorithm.*

Corollary 4.1 says that LRU is the only  $\mathcal{F}$ -mapping arbitrary set-associative stack algorithm that is address-independent, with  $\mathcal{F} = \{ f_\alpha \mid f_\alpha(p) = p \bmod 2^\alpha, 0 \leq \alpha \leq k \}$ .

A set-mapping function puts the blocks into a specific arrangement of groupings or partitions. Constant-valued functions put all blocks into the same group, i.e., a single partition, so they are equivalent when used as set-mapping functions. When discussing a set of set-mapping functions, we are interested in those functions that group the blocks in different ways. Here we introduce a notion of equivalent class among set-mapping functions.

**Definition 4.5** *Let functions  $f_1$  and  $f_2$  belong to  $\mathcal{F}$ , a set of set-mapping functions. If for any blocks  $b_1$  and  $b_2$ ,  $f_1(b_1) = f_1(b_2)$  if and only if  $f_2(b_1) = f_2(b_2)$ , then  $f_1$  and  $f_2$  are called equivalent functions.*

We denote by  $[f]^{\mathcal{F}}$  the equivalent class of  $f \in \mathcal{F}$ , the functions in  $\mathcal{F}$  that are equivalent to  $f$ . Use  $[0]^{\mathcal{F}}$  for the equivalent class of constant-valued functions.  $\|\mathcal{F}\|$  denotes the number of equivalent classes in  $\mathcal{F}$ .

Notice that in the previous proofs, nothing intrinsic about congruence mapping is used. In fact, the lemmas can be extended to any set-mapping function, and the theorem and corollary can be extended to any set of set-mapping functions with *more than two* equivalent classes.

For brevity, we shall use  $[x]_f$  to represent the set of blocks whose values on function  $f$  is  $f(x)$ :  $[x]_f \stackrel{\text{def}}{=} \{ p \mid f(p) = f(x) \}$ ,  $S_t(r, f)$  to represent the stack of the blocks  $p$  such that  $f(p) = r$  at time  $t$ , and  $\Delta_t^f$  to represent the stack distance of  $x_t$  in the substack  $S_{t-1}(f(x_t), f)$ , for  $\forall f \in \mathcal{F}$ .

**Lemma 4.3** *For a  $\mathcal{F}$ -mapping arbitrary set-associative stack algorithm, where  $\mathcal{F} = \{ f \mid f : N \mapsto R \}$  is a set of set-mapping functions, accessing  $x_t$  should not alter any other set that  $x_t$  does not belong to. That is,*

$$\text{for } \forall f \in \mathcal{F} \text{ and } \forall r \neq f(x_t), S_t(r, f) = S_{t-1}(r, f)$$

**Proof.** Similar to Lemma 4.1. Consider a fixed  $f \in \mathcal{F}$ . Accessing  $x_t$  only needs to look into the part of cache space allocated to set  $[x_t]_f$ . None of the other substacks  $S_{t-1}(r, f)$ , where  $r \neq f(x_t)$ , is affected,  $S_t(r, f) = S_{t-1}(r, f)$  for  $\forall r \neq f(x_t)$ . Since  $f$  is chosen arbitrarily, this holds for  $\forall f \in \mathcal{F}$ .  $\square$

**Lemma 4.4** *For a  $\mathcal{F}$ -mapping arbitrary set-associative stack algorithm, if two blocks  $p_1$  and  $p_2$  are in the same set  $[p]_f$  of some set-mapping function  $f \in \mathcal{F}$ : i.e.,  $f(p_1) = f(p_2) = f(p)$ . then the relative ordering of  $p_1$  and  $p_2$  in substack  $S_t(f(p), f)$  is preserved in the global stack  $S_t$  at any time  $t$ :*

$$p_1 \xrightarrow{S_t(f(p), f)} p_2 \implies p_1 \xrightarrow{S_t} p_2.$$

**Proof.** Similar to Lemma 4.2. Suppose to the contrary, there exists a function  $f \in \mathcal{F}$  and time  $t > 0$ , such that

$$p_1 \xrightarrow{S_t(f(p), f)} p_2, \text{ but } p_2 \xrightarrow{S_t} p_1.$$

Let  $x_{t+1} = p_1$ , then there are two distinct values for the stack distance  $\Delta_{t+1}^f$  from scanning the substack  $S_t(f(p), f)$  and the global stack  $S_t$ . The  $\mathcal{F}$ -mapping arbitrary set-associative evaluation would yield wrong stack distance for set-mapping function  $f$ .  $\square$

We again note that Lemma 3 and 4 apply to any general stack algorithm, no matter whether it is address-independent or not.

Before proceeding further, we need another simple lemma

**Lemma 4.5** *Let  $\mathcal{F}$  be a set of at least three different set-mapping functions,  $\|\mathcal{F}\| \geq 3$ . Then  $\mathcal{F}$  has a function  $f$  such that, there exist blocks  $p, q, r \in N$  with*

$$p \neq q \neq r \neq p, \text{ but } f(p) = f(q) \neq f(r).$$

**Proof.** Clearly,  $\mathcal{F}$  has at least two different and non-constant equivalent classes  $[f_1]^\mathcal{F}, [f_2]^\mathcal{F}$ . That is,  $[f_1]^\mathcal{F} \neq [f_2]^\mathcal{F} \neq [0]^\mathcal{F} \neq [f_1]^\mathcal{F}$ . If both  $f_1$  and  $f_2$  are injective functions, i.e.,

$$\forall i, j \in N, i \neq j \implies f_1(i) \neq f_1(j) \text{ and } f_2(i) \neq f_2(j)$$

then we have

$$\forall b_1 \text{ and } b_2, f_1(b_1) = f_1(b_2) \text{ iff } f_2(b_1) = f_2(b_2)$$

By Definition 4.5, they are equivalent ( $[f_1]^\mathcal{F} = [f_2]^\mathcal{F}$ ), which is a contradiction. Hence at least one of them, say  $f_1$ , is not injective; i.e.,

$$\exists p, q \in N, p \neq q \text{ but } f_1(p) = f_1(q)$$

Since  $f_1$  is non-constant, there is another  $r \in N$  such that  $f_1(r) \neq f_1(p)$ . Obviously  $p \neq q \neq r \neq p$ .  $\square$

**Theorem 4.2**  *$\mathcal{F}$  is a set of set-mapping functions with at least three different equivalent classes:  $\|\mathcal{F}\| \geq 3$ . For an  $\mathcal{F}$ -mapping arbitrary set-associative stack algorithm that is address-independent, after accessing block  $x_t$  at time  $t$ ,*

$$p_1 \xrightarrow{S_{t-1}} p_2 \implies p_1 \xrightarrow{S_t} p_2.$$

*holds for any two blocks  $p_1, p_2$  such that  $p_1 \neq x_t, p_2 \neq x_t$ .*

**Proof.** The result is trivial if  $p_1 = p_2$ , so we consider  $p_1 \neq p_2$ .

Suppose  $p_1 \xrightarrow{S_{t-1}} p_2$ . As  $\|\mathcal{F}\| \geq 3$ , by Lemma 4.5,  $\mathcal{F}$  has a function  $f$  such that there are blocks  $p_1^0, p_2^0, x_t^0 \in N$  with

$$p_1^0 \neq p_2^0 \neq x_t^0 \neq p_1^0, \text{ and } f(p_1^0) = f(p_2^0) \neq f(x_t^0).$$

We first prove the theorem for a special case of  $p_1 = p_1^0, p_2 = p_2^0, x_t = x_t^0$ . Since  $p_1^0 \xrightarrow{S_{t-1}} p_2^0$ , applying Lemma 4.4 by contradiction,

$$p_1^0 \xrightarrow{S_{t-1}(f(p_1^0), f)} p_2^0 \quad (4.13)$$

Since  $x_t^0$  does not belong to set  $[p_1^0]_f$  that  $p_1^0, p_2^0$  are both in, by Lemma 4.3,

$$S_t(f(p_1^0), f) = S_{t-1}(f(p_1^0), f) \quad (4.14)$$

Therefore, from (4.13) and (4.14),

$$p_1^0 \xrightarrow{S_t(f(p_1^0), f)} p_2^0 \quad (4.15)$$

Applying Lemma 4.4 to Equation (4.15),

$$p_1^0 \xrightarrow{S_t} p_2^0$$

For the general case of  $p_1 \neq p_2 \neq x_t \neq p_1$ , do the same permutational transformation on block numbers as in the proof of Theorem 4.1. The address independency of the concerned stack algorithm leads to the conclusion for the general case.  $\square$

**Corollary 4.2** *LRU is the only address-independent  $\mathcal{F}$ -mapping arbitrary set-associative stack algorithm for any  $\|\mathcal{F}\| \geq 3$ .*

**Proof.** Similar to Corollary 4.1, by applying Theorem 4.2.  $\square$

Now we can prove Corollary 4.1 in another way: for two's power congruence-mapping,  $\mathcal{F} = \{ f_\alpha \mid f_\alpha(p) = p \bmod 2^\alpha, 0 \leq \alpha \leq k \}$ . As  $\|\mathcal{F}\| = k + 1 \geq 3$  for  $k \geq 2$ , Corollary 4.2 applies.

If a set mapping scheme is further relaxed to include arbitrary block groupings, the above result can be extended to address-dependent stack algorithms.

LRU will be the only stack algorithm, not just address-independent, that allows one-pass all set-associativity evaluation using a single stack. Here is the definition for a set mapping scheme with arbitrary block groupings:

**Definition 4.6** *Let  $\mathcal{F}$  be a set of set-mapping functions such that for  $\forall$  blocks  $p, q, r$  where  $p \neq r, q \neq r$ , there exists a set-mapping function  $f \in \mathcal{F}$  such that  $p, q$  are in a same set defined by  $f$ , but  $r$  is in a different set:  $f(p) = f(q) \neq f(r)$ .  $\mathcal{F}$  is said to contain universal set-mappings.*

Now we have a stronger result which applies to all general stack algorithms:

**Theorem 4.3** *Suppose  $\mathcal{F}$  contains universal set-mappings. For a  $\mathcal{F}$ -mapping arbitrary set-associative stack algorithm, after its accessing block  $x_t$ ,*

$$p_1 \xrightarrow{S_{t-1}} p_2 \implies p_1 \xrightarrow{S_t} p_2.$$

*holds for any two blocks  $p_1, p_2$  in the global stack such that  $p_1 \neq x_t, p_2 \neq x_t$ .*

**Proof.** Suppose  $p_1 \xrightarrow{S_{t-1}} p_2$ . Since  $p_1 \neq x_t, p_2 \neq x_t$ , and  $\mathcal{F}$  contains universal set-mappings, by Definition 4.6, there  $\exists f \in \mathcal{F}$  such that

$$f(p_1) = f(p_2) \neq f(x_t) \tag{4.16}$$

The rest is similar to what we did before:

Since  $p_1 \xrightarrow{S_{t-1}} p_2$ , applying Lemma 4.4 by contradiction,

$$p_1 \xrightarrow{S_{t-1}(f(p_1), f)} p_2 \tag{4.17}$$

As  $x_t$  is not in the set  $[p_1]_f$  that  $p_1, p_2$  belong to, by Lemma 4.3,

$$S_t(f(p_1), f) = S_{t-1}(f(p_1), f) \tag{4.18}$$

Therefore, from (4.17) and (4.18),

$$p_1 \xrightarrow{S_t(f(p_1), f)} p_2 \tag{4.19}$$

Applying Lemma 4.4 to Equation (4.19),

$$p_1 \xrightarrow{S_t} p_2$$

□

By this we get to the final conclusion:

**Corollary 4.3** *For a set  $\mathcal{F}$  of universal set-mappings, LRU is the only  $\mathcal{F}$ -mapping arbitrary set-associative stack algorithm.*

**Proof.** Similar to Corollary 4.1, using Theorem 4.3.  $\square$

### 4.3 Summary

To the question of whether other stack algorithms would permit an efficient all set-associativity stack evaluation like LRU, we have shown that among all the stack algorithms which do not base replacement decisions on the numerical values of block address, LRU is the only such stack algorithm. This result can be generalized to any set associative caching which has more than two different equivalent classes of set mapping schemes. If the set mapping schemes include all possible block groupings, then LRU is the only such algorithm among all stack algorithms, no matter how they make their replacement decisions.



# Chapter 5

## Multilevel Hierarchies

**Abstract.** We show an implementation of general (stack) replacement algorithms on multilevel memory hierarchies that maintains the staging properties and provides quick up-staging of data blocks in the hierarchy. This kind of multilevel hierarchy can be efficiently evaluated by using the stack evaluation techniques on two-level hierarchies.

### 5.1 Introduction

Multilevel hierarchies consist of an array of memory modules  $M_1, \dots, M_H$  that provide data storage. An upper level module (lower number) has smaller storage capacity than a lower level, but provides quicker response to a reference request and is more costly per unit storage. By effectively exploiting reference locality and keeping frequently used data in the upper levels, reference requests from the CPU are served quickly, while the cost of the whole system is kept low. Cache memories, commonly used in computer systems, are just two-level memory hierarchies, with the first level module being referred to as the *cache*.

Similarly, when a requested data is deep down the hierarchy, it needs to be brought into some upper level, so that future CPU access to that data can be served faster. If the upper level is already full at the time, then some old data will have to be replaced to make room for the newly requested data. The decision

of choosing an old data for replacement is often based on some stack replacement algorithm. To reduce overhead, data movement within a hierarchy is carried out in blocks, the size of which might vary from one level to another. A transfer between level  $M_i$  and any lower level  $M_j$  ( $i < j$ ) is done in  $B_i$ -byte blocks, with  $B_1 \leq \dots \leq B_{H-1}$ . The effectiveness of a memory hierarchy is measured as well by hit ratios, the percentage of references to each level. If the response time of the  $i$ -th level is  $T_i$  (including the time to transfer a block containing the data to some upper level(s) before giving the data to the CPU), and its hit ratio is  $p_i$ , then the expected response time of the hierarchy is  $\sum_{i=1}^H p_i T_i$ .

Stack evaluation of arbitrary multilevel hierarchies is difficult in general. This is because the reference pattern observed by the second or any lower level depends on the size and the replacement algorithm of all the levels above it. It seems difficult to apply an efficient evaluation method such as one-pass stack processing for a general multilevel hierarchy.

Mattson et al. [Mattson 70] proposed a multilevel hierarchy with a single block size and stack algorithm for all levels, with the limitation that only one copy of each block can reside in the hierarchy above the bottom level  $M_H$ . They showed that hit ratios can be obtained by stack evaluation for arbitrary configurations (the number of levels and their capacities) of such hierarchies.

Slutz and Traiger [Slutz 72a] introduced the concept of staging-hierarchies, which satisfy two properties: (1). inclusion: the data content of every level, except the bottom one, is a subset of the data content of its immediate lower level; (2). independence: the data content of every level is independent of the number of levels above it and their capacities. Staging-hierarchies are more realistic than the hierarchy of Mattson et al. because they allow the flexibility of multiple copies of the same data to reside in the hierarchy. The inclusion property between levels makes the implementation of certain cache coherence protocols easier for some tree-structured multiprocessor memory hierarchies [Baer 89]. The independence property makes hierarchy evaluation simple, since from the evaluation point of view, for a specific level, the existence of the upper level(s) are immaterial; it can

be viewed as the top level of a two-level hierarchy and evaluated using the stack processing method.

Gecsei[Gecsei 74] described one realization of staging hierarchies with different block sizes among the levels. Each CPU access is broadcast to all levels. The top level uses an arbitrary stack algorithm; a lower level's replacement policy is determined by those of its upper levels, forced by the containment (staging) property. Evaluation of such hierarchies is very expensive, as it requires a separate stack processing for each level of the hierarchy. Traiger and Slutz[Traiger 71] showed that if the top level uses the LRU replacement algorithm, then the replacement algorithms of the lower levels are all LRU. Such LRU multilevel hierarchies can be evaluated efficiently, as one-pass stack evaluation works for multiple block sizes[Slutz 72a]. Gecsei[Gecsei 74] later gave an implementation of this LRU hierarchy without using broadcasting; replacement decision at each lower level is made in a distributed fashion based on information passed down from its immediate upper level, and a requested data block is staged sequentially up the hierarchy to the top level before the data is given to the CPU. Silberman[Silberman 83] proposed a *delayed-staging hierarchy* in which the highest level with the requested data supplies it to the CPU directly, and the data block is "staged" into the top level sequentially through all intermediate levels, during which time a new CPU request may arrive and be concurrently served. Block replacement is also carried in a delayed way. It was shown that stack processing can be applied by considering the delayed staging times.

We consider applying a general stack algorithm to multilevel hierarchies in a way that both provides quick response to CPU requests and permits efficient evaluation. The multilevel hierarchy under consideration has a single block size for all levels (one-pass evaluation of an arbitrary stack algorithm for multiple block sizes seems out of reach). A common general stack algorithm is implemented on all levels of a staging hierarchy with minimal exchange of replacement information between adjacent levels. The staging of a missing data block into upper levels gets done fast by using broadcasting. One-pass stack processing of

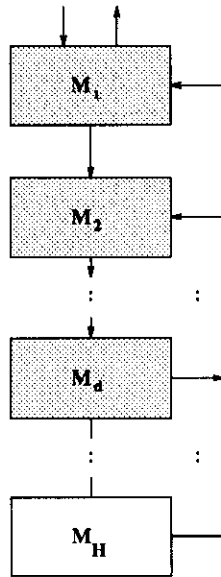


Figure 5.1: Multilevel Hierarchy Control

the stack algorithm on an access trace gives hit ratios for any configuration of such hierarchies.

## 5.2 Multilevel with Arbitrary Stack Algorithm

Consider the  $H$  level memory hierarchy  $M_1, \dots, M_H$  depicted in Figure 5.1. The center links connecting adjacent levels are for passing replacement-related information from the level above, and the bus on the right is used for up-staging a data block from a lower level to all upper levels. The CPU directly accesses the top level  $M_1$  only. Demand-fetching is assumed: only when the requested data is not in  $M_1$ , is it brought up from the upper-most level that contains it. Initially all data blocks reside in the bottom level  $M_H$ . All levels use the same stack preplacement algorithm  $A$  and the same block size  $B$ . The  $i$ -th level  $M_i$  ( $1 \leq i \leq H$ ) has a capacity of  $C_i$  blocks, with  $1 \leq C_1 < C_2 < \dots < C_H$ .  $P_t$  is the priority list of  $A$  for access time  $t$ .  $A$ 's stack is  $S_t$  after each access time  $t$ , when CPU access to  $x_t$  has just been fulfilled.

### 5.2.1 Implementation

Taking an approach similar to Gecsei's[Gecsei 74], we call a block residing in a level a *free* block for that level if there is no copy of that block in any upper level.

Let  $M_i(t)$  denote the content of level  $M_i$  after time  $t$ .

**Definition 5.1** *A block  $x$  is a free block for level  $M_i$  after access time  $t$ , if  $x \in M_i(t)$ , and  $x \notin M_j(t), 1 \leq j < i$ .*

All blocks in level  $M_1$  are free blocks for level  $M_1$ . Notice that to preserve the inclusion property of staging hierarchies, whenever a block replacement is needed at some level, only free blocks in that level should be considered.

Each level  $M_i$  keeps an ordered list of its free blocks. Let  $F_t^i$  denote the free block list of level  $M_i$  after access time  $t$ . If a referenced block  $x_t$  is first found in  $M_d$ , then for  $1 \leq i \leq d$  each  $M_i$  updates its free list  $F_{t-1}^i$  for  $x_t$ . A free list is updated in the same way as stack  $S_{t-1}$  is updated with priority list  $P_t$ , with the exception that  $x_t$  is not appended to the head of the new free list  $F_t^i$ . Specifically, let  $F_{t-1}^i = [f_{t-1}^i(0), \dots, f_{t-1}^i(m_i)]$  be the free block list of level  $M_i$  before access time  $t$ . First, compare  $f_{t-1}^i(0)$  with  $f_{t-1}^i(1)$ , let the one with higher priority in  $P_t$  be  $f_t^i(1)$ , and the one with lower priority be  $g_t^i(1)$ ; then compare  $g_t^i(1)$  with  $f_{t-1}^i(2)$ , yielding similarly  $f_t^i(2)$  with higher priority and  $g_t^i(2)$  with lower priority. Generally, for  $1 < k \leq m_i$ , compare  $g_t^i(k-1)$  with  $f_{t-1}^i(k)$ , producing  $f_t^i(k)$  with higher priority and  $g_t^i(k)$  with lower priority. This process stops at  $f_{t-1}^i(j)$  if  $f_{t-1}^i(j) = x_t$ ; then  $x_t$  is pulled out of the free block list, and the new free block list of  $M_i$  after access time  $t$  is  $F_t^i = [f_t^i(1), \dots, f_t^i(j-1), g_t^i(j-1), f_{t-1}^i(j+1), \dots, f_{t-1}^i(m_i)]$ . Figure 5.2 illustrates this case for level  $M_d$ . If  $x_t$  is not found in  $M_i$ , then in the end  $f_{t-1}^i(m_i)$  and  $g_t^i(m_i-1)$  are compared to produce  $f_t^i(m_i)$  and  $g_t^i(m_i)$ , and the new free block list of  $M_i$  is  $F_t^i = [f_t^i(1), \dots, f_t^i(m_i), g_t^i(m_i)]$ .

Suppose at time  $t$  the CPU issues to  $M_1$  a reference request of some byte in block  $x_t$ .  $M_1$  looks for it in its free list, doing the above free-list updating as it goes. If  $x_t$  is found, then it is pulled to the head of the new free list, and the requested byte is sent to CPU. If  $x_t$  is not found,  $M_1$  will send the request to

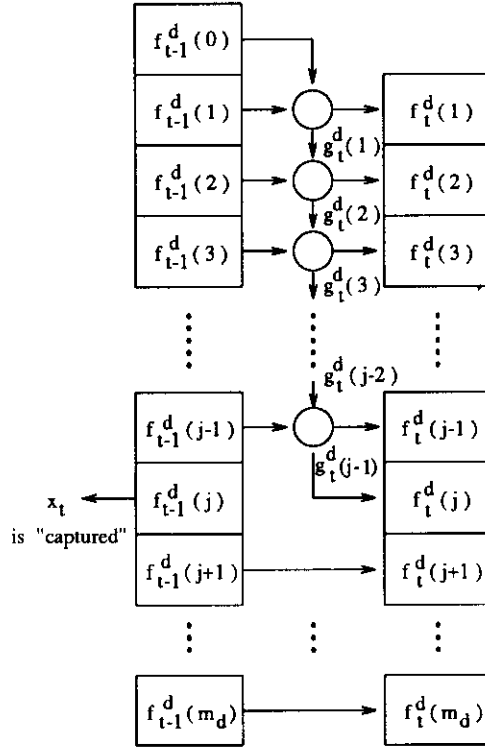


Figure 5.2:  $M_d$  Free List Updating

$M_2$ ; if  $M_1$  is already filled, the last block on its new free list is deleted.  $M_1$  sends a 3-tuple of numbers: the needed block address  $x_t$ , the current access time  $t$ , and the address  $y$  of the deleted block (if no deletion,  $y = nil$ ). Here  $t$  is necessary to inform the lower levels of the proper priority list  $P_t$  to use in updating their free lists, since lower levels do not observe every CPU access as  $M_1$  does and consequently need to know the current access time  $t$ .

Generally, a lower level  $M_i$  ( $i > 1$ ) on the search path for  $x_t$  receives the 3-tuple  $(x_t, t, y)$  from  $M_{i-1}$ . If  $y \neq nil$ , by Lemma 5.1 shown later in section 5.2.2, block  $y$  is guaranteed to be in  $M_i$ ;  $M_i$  appends block  $y$  to the head of its free block list  $F_{t-1}^i$ . Then  $M_i$  looks for  $x_t$  in  $F_{t-1}^i$ , updating it into  $F_t^i$  at the same time. If  $x_t$  is found,  $M_i$  pulls it out of its free list, and broadcast the block  $x_t$  to all upper levels. If  $x_t$  is not found, the request is propagated to  $M_{i+1}$ ; if in this case  $M_i$  is already full, the last block on its new free list  $F_t^i$  is deleted from  $M_i$ ,

and the new 3-tuple of numbers  $(x_t, t, y)$  is passed to  $M_{i+1}$ .

The action of an intermediate level  $M_i$  ( $1 < i < d$ ) is as follows:

<pre> Input: <math>(x_t, t, y)</math> from <math>M_{i-1}</math>.   if ( <math>y \neq nil</math> and block <math>y \in M_i</math> ) {     append block <math>y</math> to the head of <math>F_{t-1}^i</math>;     <math>y = nil</math>;   }   <math>F_t^i = \text{update\_free\_list}(F_{t-1}^i)</math>;   if (number of blocks in <math>M_i = C_i</math>) {     <math>y = \text{address\_of\_last\_block}(F_t^i)</math>;     <math>F_t^i = \text{delete\_last}(F_t^i)</math>;     delete <math>y</math> from <math>M_i</math>;   } Output: <math>(x_t, t, y)</math> to <math>M_{i+1}</math>. </pre>
--

By replacing a block out of each filled level along the way, it is guaranteed that every level above  $M_d$  has at least one open slot to store the data block  $x_t$  later.

When  $M_d$  finds  $x_t$  during updating its free list, it takes  $x_t$  out of its free list (shown in Figure 5.2), as block  $x_t$  will be residing in upper levels and will not be a free block to  $M_d$ , and broadcasts it to all upper levels. After the entire block  $x_t$  is received,  $M_1$  appends it to the head of  $F_t^1$  and provides the requested byte to the CPU. Now block  $x_t$  resides in every level of the hierarchy.

Figure 5.1 illustrates the operations of the hierarchy, with shaded rectangles representing the levels participating in current access. The down-arrows between levels are for sending the 3-tuple numbers  $(x_t, t, y)$ , sequentially passed from  $M_1$  through  $M_2, \dots, M_{d-1}$  to  $M_d$ . The right-arrow is for  $M_d$ 's broadcasting data block  $x_t$ . The left-arrows are for  $M_i$ 's ( $1 \leq i < d$ ) receiving data block  $x_t$ . Sequential search down the hierarchy is essential for a general stack algorithm, for a lower level can not make a replacement decision without knowing which block its immediate upper level has chosen to replace. The information about access request  $x_t$  is only sent down to level  $M_d$ , the highest level that has a copy of the data block before current CPU access.

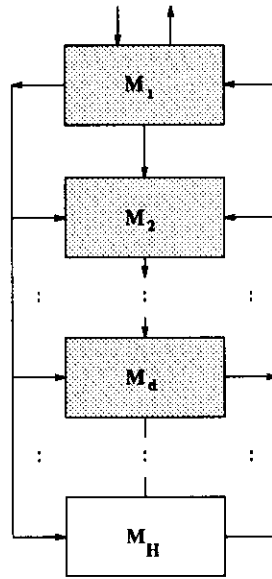


Figure 5.3: LRU Hierarchy Control

### Remarks

- Commonly used stack algorithms, such as LRU and LFU, have the characteristic that the relative ordering of the priorities of two blocks remains unchanged if neither of them is being accessed. These algorithms can be simply implemented by attaching a priority number to each block, and changing the priority accordingly whenever the block is accessed by the CPU. When a block is replaced from a higher level, its priority is passed down to be attached to the block's copy in the next lower level. For such algorithms, the current priority list  $P_t$  is stored with the blocks themselves, so it is unnecessary to pass down the time  $t$  while looking for  $x_t$ .
- For the LRU algorithm, the hierarchy can be made to operate even faster. There is no need to sequentially step through the intermediate levels to update their free block lists; instead, the top level broadcasts the address of the needed block via the bus on the left to all levels in the hierarchy, shown in Figure 5.3. The LRU algorithm has a very simple stack updating,



which merely shifts all stack entries down by one position until the stack entry containing the data is found. When reflected in the updating of the free block lists of  $M_1, \dots, M_d$ , this can be done in parallel by these levels. Each intermediate level  $M_i$  for  $i = 2, \dots, d-1$ , upon getting the broadcast address from  $M_1$ , checks that it doesn't have the block, deletes its last free block from its memory module if it is full, and sends the address of the deleted block down to the next level  $M_{i+1}$  via the center link; at the same time, it prepares to receive a deleted block address from its previous level  $M_{i-1}$ , which must happen by Lemmas 5.1 and 5.3 later, and appends that block to the head of its new free list.

A level above  $M_d$  decides that it is an intermediate level if it does not contain the sought-for block. Level  $M_d$  has the sought-for block as one of its free blocks and can be sure that it is the top-most level containing the data. A level below  $M_d$  decides that it is not going to participate in the current access if it has the block but it is not free. According to Lemma 5.5 below, if the requested data block is first found in a level other than  $M_H$  (i.e.,  $d < H$ ), then all upper levels  $M_1, \dots, M_{d-1}$  are full, and they all delete one free block to make room for the block to be broadcast by  $M_d$ . In particular,  $M_{d-1}$  will send the deleted block number to  $M_d$ , after which  $M_d$  can begin to broadcast the data block through the right bus. Since  $M_d$  is the slowest memory module, after it finishes receiving the block number from  $M_{d-1}$ , all other upper levels have done their free list updating; they are ready to receive the data block broadcast by  $M_d$  now.

If the requested data block is found only in  $M_H$ ,  $M_{H-1}$  may and may not be full. Since  $M_H$  will not broadcast the data block until it has received a block number from  $M_{H-1}$ , it is necessary that  $M_{H-1}$  always provide a block number to  $M_H$ . When a CPU request is not found in  $M_{H-1}$  and  $M_{H-1}$  is not full,  $M_{H-1}$  can send a fake replacement block number to  $M_H$ , so  $M_H$  knows when it can broadcast the data block.

- Only the levels above  $M_d$ , which receive the searching 3-tuples and which do not have the requested data, are informed to receive the broadcasting of the data block by  $M_d$ . All levels below  $M_d$  will not participate in receiving the broadcast. As all upper levels have faster response time than lower levels, there is no problem for the receiving speeds of upper levels  $M_1, \dots, M_{d-1}$  to keep up with the sending speed of  $M_d$ .
- The multilevel hierarchy can be reconfigured dynamically, using a special switch on each memory module to inform whether it is the top level module. The top level is different from all the other levels in that it gets requests from the CPU in the beginning of each access, and sends a byte to the CPU at the completion of the access; if the data is in  $M_1$ , there is no need to broadcast. Each module can decide whether it is  $M_1$  by checking if its special switch is set. Memory modules can be added to or deleted from the hierarchy, with smaller and faster modules above larger and slower ones.
- We assumed all capacities to be different among the levels. Two contiguous levels with the same capacity makes no practical sense; they pose no problem to evaluation either, as their contents are identical and can be treated as one level.

### 5.2.2 Properties

We study some properties of such multilevel hierarchies that are useful to their evaluation. For clarity of exposition, first define some notions. A *list* is an ordered sequence of distinct blocks  $L = [l_1, \dots, l_n]$ ,  $l_i \neq l_j$  if  $i \neq j$ .  $l_1$  is the leading block.  $x \in L$  if there is an equal block  $l_i = x$ ,  $1 \leq i \leq n$ .  $|L| = n$  is the size of the list. A sublist between entries  $i$  and  $j$  is denoted  $L[i, j] = [l_i, \dots, l_j]$ . A sublist from entry  $i$  to the end of the list is  $L[i, \cdot] = L[i, |L|]$ . The concatenation of two lists  $X = [x_1, \dots, x_n]$  and  $Y = [y_1, \dots, y_m]$  is  $X \bullet Y = [x_1, \dots, x_n, y_1, \dots, y_m]$ . A list is empty if it has no elements, denoted by  $[\ ]$ . If  $i > j$ ,  $L[i, j] = [\ ]$ .  $\text{Set}(L)$  is the set of blocks which are from the list  $L$ :  $\text{Set}(L) = \{x | x \in L\}$ .

**Lemma 5.1**  $M_{i-1}(t) \subset M_i(t)$  for  $1 < i < H$ .

**Proof.** Whenever a block  $x$  is to be brought into  $M_{i-1}$ , it must first be brought into  $M_i$ , and as long as  $x \in M_{i-1}(t)$ ,  $x$  is not a free block in  $M_i$  and will not be replaced, hence  $x \in M_i(t)$ .  $\square$

**Lemma 5.2** For  $1 < i < H$ ,  $\text{Set}(F_t^i) = M_i(t) - M_{i-1}(t)$ , hence  $|F_t^i| = |M_i(t)| - |M_{i-1}(t)|$ .

**Proof.** By Definition 5.1,  $\text{Set}(F_t^i) \subset M_i(t) - M_{i-1}(t)$ . For any  $x \in M_i(t) - M_{i-1}(t)$ ,  $x \notin M_{i-1}(t)$ ; by Lemma 5.1,  $x \notin M_j(t)$  for all  $1 \leq j \leq i-1$ ,  $x \in F_t^i$ . Thus  $M_i(t) - M_{i-1}(t) \subset \text{Set}(F_t^i)$ . Again by Lemma 5.1,  $|F_t^i| = |\text{Set}(F_t^i)| = |M_i(t)| - |M_{i-1}(t)|$ .  $\square$

**Lemma 5.3** For  $1 \leq i < H$ , if  $|M_i(t)| = C_i$ , then  $|M_i(t')| = C_i$  for  $t' > t$ .

**Proof.** Whenever  $M_i$  becomes full, it will stay full forever, since each time it replaces a block, it does so in order to get another one, hence the conclusion.  $\square$

Since  $C_{i-1} < C_i$ ,  $M_{i-1}$  gets full before  $M_i$  does; if  $|M_i(t)| = C_i$ , by Lemma 5.3,  $|M_{i-1}(t)| = C_{i-1}$ . This means that replacement always starts from the top level.

**Lemma 5.4** Define  $C_0 = 0$ , then  $|F_t^i| \leq C_i - C_{i-1}$  for  $1 \leq i < H$ .

**Proof.** As  $C_i - C_{i-1} > 0$ , consider  $|F_t^i| > 0$ , which means  $M_i$  gets a free block replaced by  $M_{i-1}$  after some access time  $t_0$ :  $t_0 \leq t$ , so  $|M_{i-1}(t_0)| = C_{i-1}$ . By Lemma 5.3,  $|M_{i-1}(t)| = C_{i-1}$ ; by Lemma 5.2,  $|F_t^i| = |M_i(t)| - |M_{i-1}(t)| = |M_i(t)| - C_{i-1} \leq C_i - C_{i-1}$ .  $\square$

**Lemma 5.5** If  $|F_t^i| > 0$ , then  $|M_j(t)| = C_j$  and  $|F_t^j| = C_j - C_{j-1}$  for any  $1 \leq j < i$ .

**Proof.** As  $|F_t^i| > 0$ , by the proof of Lemma 5.4,  $|M_{i-1}(t)| = C_{i-1}$ . By Lemma 5.2,  $|F_t^{i-1}| = |M_{i-1}(t)| - |M_{i-2}(t)| = C_{i-1} - |M_{i-2}(t)| \geq C_{i-1} - C_{i-2}$ . But Lemma 5.4 has  $|F_t^{i-1}| \leq C_{i-1} - C_{i-2}$ . Apply the same reasoning to each upper level  $M_j$  for  $j = i-1, \dots, 1$  in turn.  $\square$

**Lemma 5.6**  $F_t^1 = S_t[1, C_1]$ .

**Proof.** As  $M_1$  uses stack algorithm  $A$  for its replacement decisions, by the definition of stack algorithm, the content of  $M_1$  after access time  $t$  is the first  $C_1$  entries of  $A$ 's stack  $S_t$ . All blocks in  $M_1$  are its free blocks; from the free list updating procedure described in section 5.2.1, it is clear that  $F_t^1$  is exactly the first  $C_1$  entries of  $S_t$ .  $\square$

**Theorem 5.1** For  $1 \leq i < H$ ,  $F_t^i = S_t[C_{i-1} + 1, C_i]$ .

**Proof.** Due to Lemma 5.6, only need to prove the conclusion for  $i > 1$ .

The proof is by induction on time  $t$ . Initially, the stack is empty:  $S_0 = [ ]$ ; each level except the last one is also empty,  $F_0^i = [ ]$  for  $1 \leq i < H$ . The conclusion is true.

Suppose the conclusion holds after access time  $t - 1$  ( $t \geq 1$ ). Now the CPU references  $x_t$ ; let  $d_t$  be the stack distance of  $x_t$  in  $S_{t-1}$  ( $1 \leq d_t \leq \infty$ ). After updating, the stack remains unchanged for all entries beyond  $d_t$ :

$$S_t[d_t + 1, \cdot] = S_{t-1}[d_t + 1, \cdot] \quad (5.1)$$

There are three cases to consider:

- $d_t \leq C_1$ :  $x_t \in F_{t-1}^1 = S_{t-1}[1, C_1]$ ,  $x_t$  is found at the top level  $M_1$ . All other levels except  $M_1$  remain immune to the access,  $F_t^i = F_{t-1}^i$ ,  $1 < i < H$ . As  $C_i \geq C_2 \geq d_t + 1$ , by (5.1),  $F_t^i = F_{t-1}^i = S_{t-1}[C_{i-1}, C_i] = S_t[C_{i-1}, C_i]$ .
- $C_{l-1} < d_t \leq C_l$  for some  $1 < l < H$ :  $x_t \in F_{t-1}^l = S_{t-1}[C_{l-1} + 1, C_l]$ ,  $x_t$  is first found in level  $M_l$  and has to be brought into  $M_{l-1}, \dots, M_1$ . As  $|F_{t-1}^l| > 0$ , by Lemma 5.5,  $|M_i(t)| = C_i$  and  $|F_{t-1}^i| = C_i - C_{i-1}$  for  $1 \leq i < l$ , each level above  $M_l$  is full and needs to replace a block to make room for  $x_t$ . From the proof of Lemma 5.6, it is clear that  $M_1$  picks none other than  $y_t(C_1)$  as its replacement block, and passes the address of  $y_t(C_1)$  to  $M_2$ .

At each lower level  $i$  ( $2 \leq i \leq l - 1$ ),  $M_i$  gets from  $M_{i-1}$  the address of a block replaced by  $M_{i-1}$ . Suppose the address is  $y_t(C_{i-1})$ ; for an induction to hold, we want to show that the address passed from  $M_i$  to  $M_{i+1}$  will

in turn be  $y_t(C_i)$ .  $M_i$  will append  $y_t(C_{i-1})$  to the head of  $F_{i-1}^i$ , and get its new free list  $F_t^i$  and replacement block by using the free list updating procedure in section 5.2.1. As  $F_{i-1}^i = S_{i-1}[C_{i-1} + 1, C_i]$ , upon getting  $y_t(C_{i-1})$ ,  $M_i$  updates its free list in the way that exactly mimics the stack updating from  $S_{i-1}$  to  $S_t$ , for the range of entries from  $C_{i-1} + 1$  to  $C_i$ . Hence  $F_t^i = S_t[C_{i-1} + 1, C_i]$ , and the replacement block will precisely be  $y_t(C_i)$ . The address  $y_t(C_i)$  is passed down to  $M_{i+1}$ .

Finally, level  $M_l$  gets from  $M_{l-1}$  the address  $y_t(C_{l-1})$  of the block replaced by  $M_{l-1}$ .  $M_l$  mimics stack updating on its free list until reaching  $F_{l-1}^l[d_t - C_{l-1}] = x_t$ , then it deletes  $x_t$  from its free list. Hence  $F_t^l[1, d_t - C_{l-1}] = S_t[C_{l-1} + 1, d_t]$ . By (5.1),  $F_t^l[d_t - C_{l-1} + 1, C_l] = F_{l-1}^l[d_t - C_{l-1} + 1, C_l] = S_{l-1}[d_t + 1, C_l] = S_t[d_t + 1, C_l]$ , so  $F_t^l = F_t^l[1, d_t - C_{l-1}] \bullet F_t^l[d_t - C_{l-1} + 1, C_l] = S_t[C_{l-1} + 1, C_l]$ .

For all  $l < i < H$ ,  $M_i$  is unaware of the CPU access to  $x_t$ , so  $F_t^i = S_t[C_{i-1} + 1, C_i]$ .

- $d_t > C_{H-1}$ :  $x_t$  is only found in the bottom level  $M_H$ . If all upper levels have a full free list ( $|F_{i-1}^i| = C_i - C_{i-1}$ ), this is the same as the previous case except  $l = H$ , whose free list we don't need to consider.

Otherwise, let  $l$  ( $1 \leq l < H$ ) be the up-most level with  $0 \leq |F_{l-1}^l| < C_l - C_{l-1}$ . It must be that  $|S_{t-1}| < C_l \leq C_{H-1} < d_t$ . since otherwise  $|F_{l-1}^l| = |S_{t-1}[C_{l-1}, C_l]| = C_l - C_{l-1}$ . It means  $x_t$  has never been accessed before, so  $|S_t| = |S_{t-1}| + 1 \leq C_l$ . This is also similar to the previous case, except there is no block replacement at level  $l$ . At level  $l$ ,  $|M_l(t-1)| < C_l$ , no block needs to be replaced,  $F_t^l = S_t[C_{l-1} + 1, |S_t|] = S_t[C_{l-1} + 1, C_l]$ . For  $l < i < H$ ,  $F_t^i = F_{i-1}^i = [] = S_t[C_{i-1} + 1, C_i]$ .

So  $F_t^i = S_t[C_{i-1} + 1, C_i]$  holds at every level  $i$  after access time  $t$ . By inductive argument, it holds always.  $\square$

The entire stack is distributed among levels of the hierarchy in a disjoint and complementary manner. Each level does part of the stack updating by mimicking

the changes on its free list according to the common stack algorithm.

**Corollary 5.1**  $M_i(t) = \text{Set}(S_t[1, C_i])$  for  $1 \leq i < H$ .

**Proof.** With Lemmas 5.1 and 5.2,  $M_i(t) = \text{Set}(F_t^i) \cup M_{i-1}(t) = \text{Set}(F_t^i) \cup \text{Set}(F_t^{i-1}) \cup M_{i-2}(t) = \dots = \text{Set}(F_t^i) \cup \dots \cup \text{Set}(F_t^1)$ . For  $i > j$ , if  $x \in F_t^i$ , then  $x \notin M_j(t)$ ,  $x \notin F_t^j$ ; if  $y \in F_t^j$ , then  $y \in M_j(t)$ ,  $y \notin F_t^i$ . hence if  $i \neq j$ ,  $F_t^i$  and  $F_t^j$  do not have a common block, so  $M_i(t) = \text{Set}(F_t^i) \cup \dots \cup \text{Set}(F_t^1) = \text{Set}(F_t^1 \bullet \dots \bullet F_t^i) = \text{Set}(S_t[1, C_i])$ .  $\square$

After any access time  $t$ , the content of level  $M_i$  is composed of the first  $C_i$  entries of the stack  $S_t$ . This is illustrated in Figure 5.4. The hierarchy satisfies the independence property of staging hierarchies. Together with the inclusion property by Lemma 5.1, this multilevel hierarchy is a staging-hierarchy.

### 5.2.3 Evaluation

From theorem 5.1, it is clear that a request  $x_t$  is first found in level  $l$  if and only if its stack distance  $d_t$  satisfies  $C_{l-1} < d_t \leq C_l$ . Let  $r(s)$  be a counter for the number of times that stack distance  $s$  is found during the stack processing of a reference trace of length  $L$ , the total number of times an accessed block is first found in  $M_i$  is  $R_i = \sum_{s=C_{i-1}+1}^{C_i} r(s)$ , and the hit ratio of  $M_i$  is  $p_i = R_i/L$ .

#### Write Effects

Like cache memories, a write-back multilevel hierarchy writes a dirty block from the  $i$ -th level into the  $(i+1)$ -th level when that block is chosen for replacement in  $M_i$ . The difference between read-only hierarchies and write-back hierarchies is that for the former, when level  $M_i$  replaces a block, it only passes the block's address to  $M_{i+1}$ , while for the latter, when level  $M_i$  replaces a block that is dirty, it writes the block data to  $M_{i+1}$ .

Due to Theorem 5.1, evaluation of write-backs in the multilevel hierarchy is the same as the evaluation for cache memories. Denote by  $dl(x)$  the dirty level of block  $x$ . Each time  $x$  is written,  $dl(x)$  is reset to one. Let  $w(s)$  be a counter

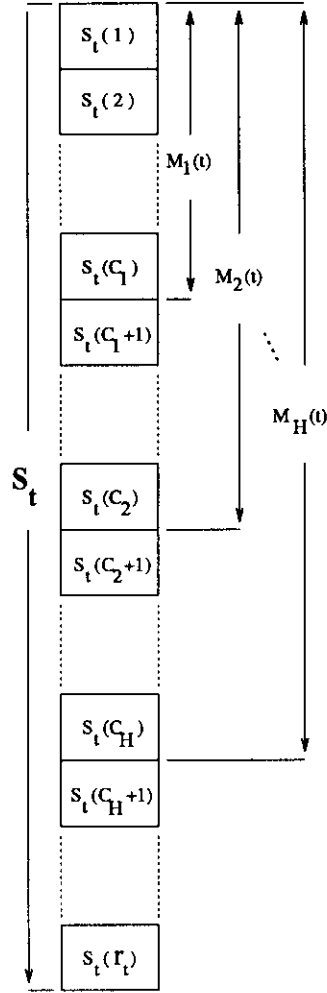


Figure 5.4: Level Contents v.s. Stack Contents

for the number of occurrence that  $s$  is the longest stack distance a dirty block has achieved between the time of its two consecutive writes, during the stack processing of a reference trace of length  $L$  of which  $W$  are write accesses. For level  $M_i$ , the number of writes it receives from  $M_{i-1}$  is  $W$  minus the number of writes saved by  $M_{i-1}$  and the number of distinct dirty blocks still residing in  $M_1, \dots, M_{i-1}$ . Hence the number of write-backs of  $M_i$  is  $W_i = W - \sum_{s=1}^{C_{i-1}} w(s) - |D_i(L)|$ , and the write ratio to level  $M_i$  is  $w_i = W_i/W$ . Here  $D_i(L) = \{x | x \in M_{i-1}(L) \wedge dl(x) \leq C_{i-1}\}$ , where  $dl(x) \leq C_{i-1}$  means that  $x$  has not been replaced

from the  $(i - 1)$ -th level  $M_{i-1}$  since last time it is written by the CPU.

### 5.3 Summary

We show an implementation of a general stack algorithm on multilevel memory hierarchies that maintains staging properties and provides quick up-staging of data blocks in the hierarchy. Due to the staging property, the multilevel hierarchies can be efficiently evaluated using the standard stack processing of reference traces. One-pass processing of the stack algorithm over an access trace gives hit ratios and write ratios for any configuration of such hierarchies.

This implementation has two apparent restrictions: that a lower level is at least as large as an upper level in capacity, and that all levels use the same block size. The first restriction reflects reality hence is not much a problem. The second restriction is probably inherent in the requirement of doing one-pass evaluation for any stack algorithm with any configuration.



# Chapter 6

## Multiprocessor Cache Analysis

**Abstract.** We model shared-memory multiprocessor caches on a bus architecture with a snoopy-based invalidation coherence protocol. Assuming an independent reference model, the general method of modeling the cache states as a Markovian chain is presented. Upper and lower bounds on cache performance are derived. Using recursion and conditional probabilistic reasoning, steady state analysis is applied to the states of LRU stack entries.

### 6.1 Introduction

Trace-driven simulation, stack evaluation in particular, is a viable tool in accurately predicting the performance of different memory hierarchy designs. Compared to full-scale, software-based system simulation, it runs at least a magnitude times faster[Chaiken 90].

However, there are several problems associated with trace-driven simulation: the validity of the used traces, the storage and computational requirements of the traces. Traces are usually gathered for a particular application. and to get some general result out of a simulation one must experiment with many traces from different application paradigms. Traces are expensive to gather and keep; because of the fast execution speed of modern computers, it takes a short time for a machine under tracing to produce a very large trace, requiring large storage

space and long simulation time.

On the contrary, while probably being less accurate than trace-driven simulation, analytical modeling can predict more general and long-term system behavior in a quick and relatively inexpensive way. It is a useful tool in the early stages of memory system design, which by identifying a sensible range of consideration for the design parameters can greatly narrow down design space.

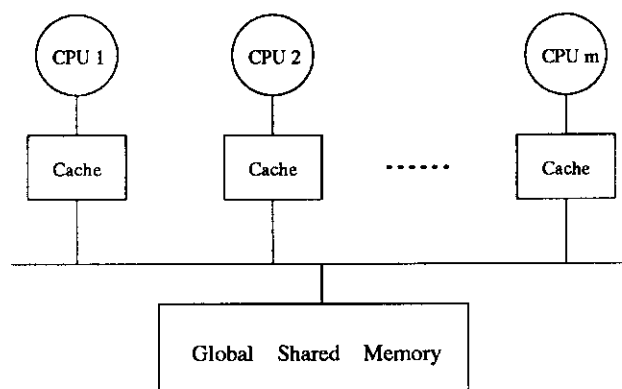


Figure 6.1: Shared-Memory Multiprocessor with Bus-Interconnection

In this chapter, we model a bus-based shared-memory multiprocessor caching system, shown in Figure 6.1. Each processor has a local cache. The caches and the shared main memory is connected by a bus interconnect. All caches use “*snoopy*” method to monitor bus transactions. Cache coherence is achieved by an invalidation-based protocol, whereby a cache invalidates its copy of a data block whenever it “*hears*” on the bus that some other cache has issued a write to that block. The actual write approach, whether write-through or write-back, is of little relevance and not specified in the analysis. Each cache employs LRU as its block replacement algorithm LRU is the mostly used algorithm in practice, and its stack updating strategy is relatively simple to analyze.

There are a number of models on reference string for the analysis of caches [King 72, Franaszek 74, Rao 78, Smith 79, Tzelnic 82, Dubois 82, Agarwal 89, Dan 93]. The most used is the independent reference model (IRM), which em-

bodies a simplifying assumption on CPU memory access patterns that makes analysis more tractable. Models based on IRM tend to underestimate memory system performance, due to its inadequacy in describing some important CPU access features such as localities. Baskett and Raffi (see [Flajolet 87]) showed that by appropriately converting observed reference probabilities into some virtual probabilities and use the virtual probabilities in modeling, the prediction on paging system performance agrees well with real performance. It demonstrated that dependent references can be modeled with independent reference model using modified probabilities for the main memory and disk hierarchy.

Under the IRM model, closed-form solution for miss ratios was obtained for fully associative LRU, FIFO, and RAND replacement algorithms on uniprocessor caches [King 72], by solving for the steady state distribution probabilities of each cache state. Some bounds on performance comparisons of LRU, FIFO, and RAND, with respect to MIN, were derived on uniprocessor caches [Franaszek 74], both for fixed memory size and for variations of performance with various memory size, using only mean values of the independent reference distribution. Set-associative caches were analyzed in [Rao 78], by solving the steady state distribution probabilities of cache states for a fixed cache size, extending the work in [King 72]. Smith used Poisson process assumption to analyze the buffering effects on a write-through cache with a buffer [Smith 79]. In [Tzelnic 82], the hit ratios of LRU algorithm on uniprocessor caches was analyzed for a first-order Markov chain reference distribution. Dubois and Briggs studied coherent LRU multiprocessor caches [Dubois 82], modeling each process and the interaction among processes, under the assumptions that the address space of each process is divided into a private cache and a shared cache, and the processes strictly alternate in their memory access. Agarwal used a hybrid of measurement and analysis [Agarwal 89] to study the effect of caching parameters on the performance of multiprogramming on uniprocessor caches. In [Dan 93], various buffer coherence policies are studied for LRU cache buffers under the IRM model, with an approximating assumption that all blocks stay in a stack entry for the same

amount of time.

We model the multiprocessor caches using two approaches. First, caches with fixed size are studied; their states are modeled by a first-order Markov chain, and their steady state probabilities found. This approach is similar to those in [King 72, Franaszek 74, Rao 78, Smith 79, Tzelnic 82], except that now we consider the cache interaction of block invalidation in the multiprocessor environment. Secondly, LRU stack entries are studied; their steady state probability distributions are formulated and solved with balance equations. [Dubois 82, Dan 93] also studied stack entry probabilities. Unlike [Dubois 82], we make no assumption about reference order of the CPUs; unlike [Dan 93], we determine approximately the steady-state probabilities.

The next section formally introduces the IRM model for the multiprocessor environment under study. The Markov chain model of cache states is described in section 3. Section 4 gives upper and lower bounds on cache performances. In section 5, a different approach is taken by analyzing the steady-state probabilities of each stack entry with recursive equations.

## 6.2 Independent Reference Model

Data are moved in blocks among the caches and the shared memory. We assume the block size is fixed. The blocks are numbered from 1 to  $n$ , with  $\Omega = \{1, \dots, n\}$  denoting the entire address space. The reference sequence observed by a typical cache is represented by  $X(1), \dots, X(L)$ , with  $X(t) \in \Omega$ .

Consider an arbitrary time epoch  $t_0$ , and suppose that the block referenced at  $t_0$  is  $X(t_0) = b$ . Let the next reference to  $b$  occur at time epoch  $t_1 > t_0$ . For LRU replacement without invalidation, we are interested in the number of distinct blocks referenced between  $t_0$  and  $t_1$ , since that determines whether block  $b$  still remains in cache at  $t_1$ . If the cache size is bigger than the number of distinct blocks referenced during the interval  $(t_0, t_1)$ , then  $X(t_1)$  is a hit; otherwise it's a miss.

Cache invalidation makes the analysis of multiprocessor caches difficult. An invalidation from a remote cache erases an existing block from the local cache, creating an empty slot for a new block referenced in the future. There are two possibilities upon which an access to block  $b$  is a miss: first,  $b$  itself has been invalidated since its last local access; second,  $b$  is not invalidated, but there have been more than  $m - 1$  ( $m$  being the cache size) other blocks that were accessed after  $b$ 's last access, and the number of valid blocks in the cache has exceeded the cache size. The last condition is necessary, and this is what makes the analysis difficult. Notice that even though the number of distinct blocks referenced during the intermission is more than the cache size, it may be that blocks are invalidated after being accessed and their slots reused for future new blocks during that time period, so there is never a necessity to replace a resident block, namely  $b$ , and  $b$  remains in the cache.  $X(t_1)$  is a hit if and only if  $b$  is not invalidated between the two accesses to it, and at any time during the intermission the number of distinct valid blocks (including  $b$ ) is no more than the cache size.

Denote by  $S(t)$  the set of valid blocks at time  $t$  since the last reference to block  $b$  at time  $t_0$ . Then  $S(t_0) = \{X(t_0)\}$ , and  $S(t+1) = S(t) - \{\text{blocks invalidated at time } t+1\} + \{X(t+1)\}$ . A necessary and sufficient condition for  $X(t_1)$  to be a hit in a cache of size  $m$ -blocks is  $X(t_1) \in S(t_1 - 1)$  and  $|S(t)| \leq m$  for  $t_0 \leq t < t_1$ .

We assume that the local CPU accesses and remote CPU invalidations to the local cache obey an Independent Reference Model (IRM). Each reference  $X(t)$  is a combination of block number  $b$  and access tag  $r$ :  $X(t) \in \{(b, r) | b \in \Omega, r \in \{-1, 1\}\}$ . A tag of 1 means a normal memory access from the local CPU, while the tag of -1 means an invalidation from a remote CPU. Each access is independent of others and has the probability distribution:  $P[X = (b, 1)] = p_b$ ,  $P[X = (b, -1)] = q_b$ . Here  $p_b > 0$ ,  $q_b \geq 0$ ,  $\sum_{b \in \Omega} p_b + q_b = 1$ . The  $p_b, q_b$  are observed by a typical CPU's cache and depend on the number of CPUs and their composite access pattern.

If  $X(i) = (b(i), r(i))$ , clearly

$$S(i) = \begin{cases} S(i-1) + \{b(i)\} & \text{if } r(i) = 1 \\ S(i-1) - \{b(i)\} & \text{if } r(i) = -1 \end{cases}$$

As a probabilistic model, IRM has been frequently applied in uniprocessor cache analysis [King 72, Franaszek 74, Flajolet 87]; under this assumption, analysis becomes tractable. It has also been widely used in analyzing a list searching problem, which is to some extent related to block replacement algorithms [Bitner 79, Mendelson 80, Sleator 85b].

We explicitly deal with a fully associative cache. Set associative caches can be treated as a group of disjoint, independent fully associative ones and the analysis is not much different.

### 6.3 Markov Chain Model

Let  $S_{t-1} = [s(1), \dots, s(m)]$  denote the state of the LRU cache before access time  $t$ . That is, the first  $m$  entries of the LRU stack. For  $1 \leq i \leq m$ ,  $s(i) \in \Omega$  or  $s(i) = \#$ .  $s(f)$  is the first marker entry “#” in  $S_{t-1}$ ; i.e..  $s(f) = \#$  and  $s(i) \neq \#$  for  $1 \leq i < f$ .  $f > m$  means  $S_{t-1}$  does not have a marker entry. An empty cache is represented by  $m$  markers. For the current access  $X(t) = (b(t), r(t))$ , the state transition from  $S_{t-1}$  to  $S_t$  is the following:

- if  $r(t) = 1$ ,
  1. if  $s(j) = b(t)$  and  $j < f$ , then  $S_t = [s(j), s(1), \dots, s(j-1), s(j+1), \dots, s(m)]$ .
  2. if  $s(j) = b(t)$  and  $j > f$ , then  $S_t = [s(j), s(1), \dots, s(f-1), s(f+1), \dots, s(j-1), \#, s(j+1), \dots, s(m)]$ .
  3.  $b(t) \notin S_{t-1}$ , then  $S_t = [b(t), s(1), \dots, s(f-1), s(f+1), \dots, s(m)]$ .
- if  $r(t) = -1$ ,

1. if  $s(j) = b(t)$ ,  $1 \leq j \leq m$ , then  $S_t = [s(1), \dots, s(j-1), \text{"\#"}, s(j+1), \dots, s(m)]$ .
2. if  $b(t) \notin S_{t-1}$  for  $1 \leq i \leq m$ , then  $S_t = S_{t-1}$ .

Given state  $S_{t-1}$  and access  $X(t)$ , the new state  $S_t$  can be determined without any additional information. Hence the state sequence  $\{S_t\}$  forms a Markov chain. Since the actual access time  $t$  plays no role in the model, the Markov chain is homogeneous. The state transition probability matrix  $P = (p_{S_1 S_2})$  is composed of either  $p_i$  or  $q_i$  and is quite straight-forward to determine following the above description of state transitions.

It is possible to apply such reasoning to any stack algorithm and model its (cache) state sequence as a Markov chain. For an arbitrary stack algorithm with markers in the stack, its stack updating procedure is as follows:

- if there is no marker above the currently accessed block in the stack, then stack updating, using the algorithm's priority list  $P_t$  for that access time, is done from stack top to the position of the data block. This is the same as the case with no markers in stack.
- if there is at least one marker above the accessed block in the stack, then stack updating is done from stack top to the position of the *first* marker, and then the marker is directly moved to the position of the accessed block, while the stack entries between them are unchanged.

Because of a stack algorithm's explicit priority list  $P_t$  at each time  $t$ , the state transition is deterministic, given the current state and the access. Therefore the state sequence is also Markovian. But since  $P_t$  can change with time, the Markov chain may not be homogeneous.

Actually, one can construct the model for non-stack algorithms as well – the difference being that one may need a different model for each cache size, if the replacement algorithm is not a stack algorithm.

Let  $Q$  be the state space:  $Q = \{[s(1), \dots, s(m)] | s(i) \in \Omega \cup \{\#\}\}$ . We have the following statement for the Markov chain for the LRU stack algorithm

**Lemma 6.1** *Suppose  $p_i > 0, q_i > 0$  for  $1 \leq i \leq n$ , then the Markov chain  $\{S_t\}$  is irreducible and aperiodic. Hence there is a unique equilibrium solution  $p(S)$  for the probabilistic distribution of steady state  $S$  in  $Q$ .*

**Proof.** To get the transition from state  $[s(1), \dots, s(m)]$  to state  $[s'(1), \dots, s'(m)]$ , we can first access  $m$  data blocks to fill up the cache, and then erase the unwanted entries. Specifically, suppose the indices of non-marker entries of the target state  $[s'(1), \dots, s'(m)]$  are  $j_1, \dots, j_d$ , i.e.,  $s'(i) = \#$  if  $i \notin \{j_1, \dots, j_d\}$ . Access sequence  $(b_m, 1), (b_{m-1}, 1), \dots, (b_1, 1)$ , where  $b_i$ 's are distinct blocks and  $b_{j_1} = s'(j_1), \dots, b_{j_d} = s'(j_d)$ , transforms the source state  $[s(1), \dots, s(m)]$  into

$$[b_1, \dots, b_{j_1-1}, s'(j_1), b_{j_1+1}, \dots, b_{j_2-1}, s'(j_2), b_{j_2+1}, \dots, b_{j_d-1}, s'(j_d), b_{j_d+1}, \dots, b_m]$$

Now a zapping access sequence  $(b_i, -1)$ , where  $i = 1, \dots, m$  and  $i \notin \{j_1, \dots, j_d\}$ , transforms it into the target state  $[s'(1), \dots, s'(m)]$ . The probability of going from the source state to the target state is at least  $(\prod_{i=1}^m p_{b_i, q_{b_i}}) / (\prod_{i=1}^d q_{s'(j_i)})$ . So the state space is irreducible. As each state goes to itself by an invalidation  $(b, -1)$  with any  $b$  not in its stack, it is aperiodic.  $\square$

The cardinality of  $Q$  is quite large:  $|Q| = \sum_{i=0}^m \binom{m}{i} \binom{n}{i} i!$  ( $i$  out of  $m$  entries are non-markers, selected from  $n$  blocks). Due to the large state space, one may have to settle with approximate solutions. Standard numerical methods for solving large Markov chains [Muntz 93] can be applied accordingly.

King[King 72] found explicit solution to the state probabilities of the above formulation for uniprocessor caches for LRU, FIFO, and  $A_0$  replacement algorithms.

Define the *kernel*  $kern(S)$  of a state  $S = [s(1), \dots, s(m)]$  in  $Q$  to be the set of non-marker elements in  $S$ :  $kern(S) = \{s(i) | 1 \leq i \leq m\} - \{\#\}$ . An access  $(b, r)$  is a miss if and only if its tag  $r = 1$  and the next state has a different kernel



than the current state. The miss ratio can be formulated by

$$P_{ms} = \sum_{S \in Q} P(S) \sum_{T \in Q: \text{kern}(T) \neq \text{kern}(S)} p_{ST} \quad (6.1)$$

where  $p_{ST} = p_b$  such that  $b \notin S, b \in T$ . An equivalent but simpler form is

$$P_{ms} = \sum_{S \in Q} P(S) \sum_{b \notin \text{kern}(S)} p_b. \quad (6.2)$$

## 6.4 Bounds

Since we lack a closed-form solution, we wish to find some bounds on  $P_{ms}$ . Under the condition that  $\{S_t\}$  is irreducible and aperiodic, i.e.  $P(S)$  is unique for each  $S \in Q$ , the steady state miss ratio can be reformulated from Equation (6.1) as

$$P_{ms} = \sum_{i=1}^n p_i z_i \quad (6.3)$$

where  $z_i = \lim_{t \rightarrow \infty} \Pr[i \notin S_t]$ , the steady state probability of finding block  $i$  absent from the cache, for  $1 \leq i \leq n$ . As above, we are assuming a specific cache size of  $m$  blocks.

In deriving the upper bound on the miss probability  $z_i$ , we use a method similar to that used by Franaszek and Wagner [Franaszek 74] and later Rao [Rao 78] in their work on uniprocessor caches. Assume without loss of generality that  $p_1 \geq p_2 \geq \dots \geq p_n$ . Let  $R_i = \sum_{j=i+1}^n p_j$ ,  $i = 1, \dots, n$ .

The event  $\{i \notin S_t\}$  can happen because either the last access to block  $i$  was an invalidation from a remote processor, or it was a local access but since then there has been a time when  $m$  blocks other than block  $i$  resided in the cache. Denote the event  $A_i =$  the last reference to block  $i$  is a local access, and at some time since the last reference to block  $i$ ,  $m$  distinct blocks other than block  $i$  resided in the cache. Let  $y_i = \Pr[A_i]$ , then  $z_i$ , the probability of finding block  $i$  missing from the cache, is the sum of the probability that the last access to block  $i$  is an invalidation (i.e.,  $q_i/(p_i + q_i)$ ), and the probability that the last access to block  $i$

is a normal access but there was a time since then that  $m$  distinct blocks other than block  $i$  resided in the cache ( $p_i y_i / (p_i + q_i)$ ),

$$z_i = q_i / (p_i + q_i) + [p_i / (p_i + q_i)] y_i$$

A necessary condition for event  $A_i$  is that there are at least  $m$  distinct blocks accessed locally since the last time block  $i$  was accessed. If  $i \leq m$ , i.e., the block number  $i$  is less than or equal to the cache size, then among the  $m$  distinct blocks accessed later, at least one of them (say block  $j$ ) has its block number greater than  $m$  ( $j > m$ ). The associated probability can be calculated as

$$\Pr[\text{some block } j > m \text{ is accessed after last access to } i \leq m] = R_m / (p_i + R_m), \quad (6.4)$$

therefore, for  $i \leq m$ ,

$$\begin{aligned} y_i &\leq [R_m / (p_i + R_m)] \Pr[\text{at least } m - 1 \text{ distinct blocks } (\neq i) \text{ are accessed}] \\ &\leq [R_m / (p_i + R_m)] \Pr[\text{some blocks } (\neq i) \text{ are accessed } m - 1 \text{ times}] \\ &= [R_m / (p_i + R_m)] \left( \sum_{k=1, k \neq i}^n p_k \right)^{m-1} \end{aligned}$$

From Equation (6.3), we have

$$\begin{aligned} P_{ms} &= \sum_{i=1}^n p_i z_i \\ &\leq \sum_{i=1}^m p_i z_i + \sum_{i=m+1}^n p_i \\ &= \sum_{i=1}^m p_i \{ q_i / (p_i + q_i) + [p_i / (p_i + q_i)] y_i \} + R_m \\ &\leq \sum_{i=1}^m p_i q_i / (p_i + q_i) + \sum_{i=1}^m p_i^2 R_m / [(p_i + q_i)(p_i + R_m)] \left( \sum_{k=1, k \neq i}^n p_k \right)^{m-1} + R_m. \end{aligned}$$

Therefore, we obtain

**Theorem 6.1**

$$P_{ms} \leq \sum_{i=1}^m p_i q_i / (p_i + q_i) + \sum_{i=1}^m p_i^2 R_m / [(p_i + q_i)(p_i + R_m)] \left( \sum_{k=1, k \neq i}^n p_k \right)^{m-1} + R_m.$$

A simplification gives

**Corollary 6.1**  $P_{ms} \leq \sum_{i=1}^m \min(p_i, q_i) + \sum_{i=1}^m p_i (\sum_{k=1, k \neq i}^n p_k)^{m-1} + R_m.$

Now let us consider an access sequence which, after the last local access to block  $i$  and before the next local access to  $i$ , consists entirely of local access (i.e., no invalidations) and each access is to a unique block. In order for this sequence to replace block  $i$ , its length has to be at least  $m$ . To get a lower bound, we count only those sequences that have exactly  $m$  local accesses to distinct blocks other than  $i$ , thus

$$\begin{aligned} y_i &\geq \sum_{j_1 \neq j_2 \neq \dots \neq j_m \neq i} \prod_{k=1}^m p_{j_k} \\ &\geq \sum_{j_1 \neq j_2 \neq \dots \neq j_m \neq i} \prod_{k=n-m+1}^n p_k \\ &\geq \binom{n-1}{m} m! \prod_{k=n-m+1}^n p_k. \end{aligned}$$

By Equation (6.3),

$$\begin{aligned} P_{ms} &= \sum_{i=1}^n p_i \{ q_i / (p_i + q_i) + [p_i / (p_i + q_i)] y_i \} \\ &\geq \sum_{i=1}^n [p_i / (p_i + q_i)] (q_i + p_i \binom{n-1}{m} m! \prod_{k=n-m+1}^n p_k). \end{aligned}$$

For a  $i \geq n - m + 1$ , the access sequence under consideration has to have at least one of its accesses made to a block  $j < n - m + 1$ ,

$$\begin{aligned} y_i &\geq p_{n-m} \sum_{j_1 \neq j_2 \neq \dots \neq j_{m-1} \neq i} \prod_{k=1}^{m-1} p_{j_k} \\ &\geq p_{n-m} \sum_{j_1 \neq j_2 \neq \dots \neq j_{m-1} \neq i} \prod_{k=n-m+2}^n p_k \\ &\geq p_{n-m} \binom{n-2}{m-1} (m-1)! \prod_{k=n-m+2}^n p_k \end{aligned}$$

hence,

$$P_{ms} \geq \sum_{i=1}^n p_i q_i / (p_i + q_i) + \sum_{i=1}^{n-m} p_i^2 y_i / (p_i + q_i) + \sum_{i=n-m+1}^n p_i^2 y_i / (p_i + q_i)$$

$$\begin{aligned} &\geq \sum_{i=1}^n p_i q_i / (p_i + q_i) + \binom{n-1}{m} m! \left( \prod_{k=n-m+1}^n p_k \right) \sum_{i=1}^{n-m} [p_i^2 / (p_i + q_i)] + \\ & p_{n-m} \binom{n-2}{m-1} (m-1)! \left( \prod_{k=n-m+2}^n p_k \right) \sum_{i=n-m+1}^n [p_i^2 / (p_i + q_i)]. \end{aligned}$$

Therefore, we have

**Theorem 6.2**

$$\begin{aligned} P_{ms} &\geq \sum_{i=1}^n p_i q_i / (p_i + q_i) + \binom{n-1}{m} m! \left( \prod_{k=n-m+1}^n p_k \right) \sum_{i=1}^{n-m} [p_i^2 / (p_i + q_i)] + \\ & \binom{n-2}{m-1} (m-1)! \left( \prod_{k=n-m+2}^n p_k \right) \sum_{i=n-m+1}^n [p_i^2 / (p_i + q_i)] \cdot \\ & \max\{(n-1)p_{n-m+1}, p_{n-m}\} \end{aligned}$$

## 6.5 Steady State Analysis of LRU Stack

In this section we look at the problem from a different angle. Instead of looking at caches of fixed size, we examine the steady state of the entire LRU stack. Our purpose is to derive the probability distribution for the state of each stack entry in equilibrium.

As before, let the LRU stack at time  $t$  be  $S_t = [s_t(1), \dots, s_t(\gamma_t)]$ . For  $i \in \Omega$ , define the steady state probability that the  $j$ -th stack entry contains the  $i$ -th block as  $p(i, j) = \lim_{t \rightarrow \infty} \Pr[s_t(j) = i]$ . Define the steady state probability of the  $j$ -th stack entry being empty as  $e(j) = \lim_{t \rightarrow \infty} \Pr[s_t(j) = \text{"#"}]$ . Obviously we have

$$\begin{cases} \sum_{j=1}^{\infty} p(i, j) \leq 1 \\ \sum_{i \in \Omega} p(i, j) + e(j) = 1 \end{cases}$$

where the first formula is less than one when block  $i$  is missing from the stack.

The miss ratio, in equilibrium, for a cache of  $C$  blocks is therefore

$$P_{ms}(C) = \sum_{i \in \Omega} p_i (1 - \sum_{j \leq C} p(i, j)) / \sum_{i \in \Omega} p_i.$$

Our goal is to solve for the defined probabilities. We derive a recursive equation for  $p(i, j)$  and  $e(j)$  in terms of  $p(k, l)$  and  $e(l)$  for  $l < j$ , and get solution for  $p(i, j), e(j)$  of all  $j$ 's.

For the first stack entry, it is simple. In order for block  $i$  to be at the top of the stack, the current access must be a normal access (not an invalidation) to block  $i$ , or block  $i$  was at the top of the stack and the current access is an invalidation directed to another block,

$$p(i, 1) = p_i + \sum_{k \neq i} q_k p(i, 1)$$

This gives

$$p(i, 1) = p_i / (1 - \sum_{k \neq i} q_k) \text{ for } i \in \Omega.$$

For the first entry to be empty, then either it was empty and the current access is an invalidation, or it was holding a block but the current access is an invalidation to that block,

$$e(1) = e(1) \sum_k q_k + \sum_k q_k p(k, 1)$$

that is,

$$e(1) = (\sum_k q_k p(k, 1)) / (1 - \sum_k q_k) = [\sum_k p_k q_k / (1 - \sum_{j \neq k} q_j)] / (1 - \sum_k q_k).$$

For an entry other than the first one, we make some observation about all the possible state changes:

- the entry is occupied by block  $i$ , either because
  1. it was occupied by  $i$ , and the current access or invalidation is directed to some other block, and no block is pushed down from above, or
  2. the immediately above entry was occupied by block  $i$ , and it gets pushed down.
- the entry is empty, either because
  1. it was empty and no block is pushed down, or

2. it was occupied by a block which is erased, or
3. it was occupied by a block which is accessed, and there was a empty entry above it.

Applying this to the second stack entry, assuming the event that block  $i$  resides in stack entry 2 is independent of the event that block  $k \neq i$  resides in stack entry 1, and also independent of whether stack entry 1 is empty:

$$p(i, 2) \approx p(i, 2) \left\{ \sum_{k \neq i} [q_k + p_k(p(k, 1) + e(1))] \right\} + p(i, 1) \sum_{k \neq i} p_k$$

from which we get

$$p(i, 2) \approx p(i, 1) \sum_{k \neq i} p_k / \left\{ 1 - \sum_{k \neq i} [q_k + p_k(p(k, 1) + e(1))] \right\}.$$

Similarly,

$$e(2) \approx e(2) \left[ \sum_k q_k + \sum_k p_k(e(1) + p(k, 1)) \right] + \sum_k p(k, 2)(q_k + p_k e(1))$$

which has

$$e(2) \approx \sum_k p(k, 2)(p_k e(1) + q_k) / \left( 1 - \sum_k q_k - e(1) \sum_k p_k - \sum_k p_k p(k, 1) \right).$$

To simplify the derivation, for  $j > 2$ , we define some auxiliary probability  $P(j) = \lim_{t \rightarrow \infty} \Pr[s_t(j) \neq s_{t-1}(j-1) \vee s_t(j) = \text{"\#"}]$ . That is to say,  $P(j)$  is the probability that no valid data block gets pushed down from stack entry  $j-1$  to  $j$ . Migration of an empty entry from above does not count as push-down. A necessary and sufficient condition for this to happen is that

- there is no push-down from entry  $j-2$  to  $j-1$ , or
- there is push-down from entry  $j-2$  to  $j-1$ , but it stops at entry  $j-1$  because
  1. entry  $j-1$  was empty, or
  2. entry  $j-1$  holds the currently accessed block.

Assuming that whether there is a block push-down from the  $(j - 1)$ -th entry to the  $j$ -th entry is independent of whether the  $j$ -th entry was empty, and also independent of whether a particular block was residing in the  $(j - 1)$ -th entry,

$$P(j) \approx P(j - 1) + [1 - P(j - 1)](e(j - 1) + \sum_k p(k, j - 1)p_k). \quad (6.5)$$

For boundary conditions, we have

$$P(1) = 1, \quad P(2) = 1 - \sum_k p(k, 1) \sum_{i \neq k} p_i.$$

Conditioning on the auxiliary probabilities, the probability of the  $j$ -th stack entry containing the  $i$ -th block can be calculated as follows. If there is no push-down from the  $(j - 1)$ -th entry, then block  $i$  was in the  $j$ -th stack entry before the current reference, and this reference is made to a block other than  $i$ ; if there is push-down from the  $(j - 1)$ -th entry, then block  $i$  was in the  $(j - 1)$ -th stack entry. So for  $j > 2$ ,

$$p(i, j) \approx P(j)p(i, j)(1 - p_i - q_i) + [1 - P(j)]p(i, j - 1)$$

which gives

$$p(i, j) \approx p(i, j - 1)[1 - P(j - 1)]/[1 - P(j)(1 - p_i - q_i)] \quad (6.6)$$

In order for a stack entry to be empty, there must not be a push-down from above during current stack updating; otherwise, this entry would be containing a valid block. So for the  $j$ -th stack entry to be empty, it must be that (1). there is no push-down from above, and (2). it was either empty before, or it contained a valid block but that block is either invalidated, or accessed hence has been moved to stack top (for this case, since there is no push-down from above, it must be that there was an empty entry somewhere above, which just migrated to the  $j$ -th entry). That is,

$$e(j) \approx P(j)[e(j) + \sum_k p(k, j)(q_k + p_k)]$$

hence

$$e(j) \approx P(j) \sum_k p(k, j)(q_k + p_k)/[1 - P(j)] \quad (6.7)$$

Using equations (6.5–6.7), all necessary probabilities can be solved in a systematic recursive manner. The process can stop at the depth that represents the maximum possible cache size.

## 6.6 Summary

In this chapter a Markov model of shared-memory multiprocessor caches on a bus-based architecture is provided. Under the assumption of independent references, the general method of solving the steady state distribution probabilities of cache states, formulated as a Markovian chain, is proposed. Simple upper bound and lower bound on the miss ratios of the LRU algorithm are derived. In another approach, instead of analyzing caches with specific sizes, the steady state probabilities of stack entries is studied, which can be solved by using recursion and conditional probabilities.



# Chapter 7

## Tree Cache Directories

**Abstract.** Shared-memory multiprocessors use caches on individual CPUs to reduce memory and network latency. Effective use of caches provides both better response to memory requests and lessens traffic demand on the interconnection network. Two approaches to the cache coherence problem are: for systems with a bus topology (hence inexpensive broadcasting mechanism), cache-snooping is an appropriate method to achieve cache coherence; for systems with a general interconnection network, a message-passing directory scheme is recommended. The most popular directory schemes can be characterized as *full-map directories*, *limited directories*, and *chained directories*. The full-map directory scheme is not scalable due to the storage required for the directories; the limited directory scheme restricts the number of caches having a copy of the same data block and limits data sharing; the chained directory scheme is slow in sequentially carrying out coherence operations of update or invalidation. In addition, some tree directory schemes were proposed that either had a fixed topology and longer tree height or could degenerate into linear lists.

We propose a novel directory scheme with a balanced binary-tree structure that dynamically changes with current data sharing. It is scalable in allowing unlimited number of caches to share the same data block, and can carry out coherence operations quickly, i.e., in logarithmic time. This scheme is especially appropriate for update-oriented coherence protocols where the sharing structure is preserved across writes. We demonstrate how the tree directories handle cache addition (a cache acquiring a data block copy) and cache deletion (a cache surrendering a data block copy due to its local block replacement), and their time complexities. We prove that this kind of tree structure is an optimal directory scheme for scalable architectures, which carries out all cache operations in minimum possible times. We also discuss some other tree-like

schemes and show that the proposed scheme has significant improvements in space and time complexities.

## 7.1 Introduction

Caches have been used as a buffering device in both uniprocessor systems and shared-memory multiprocessor systems to cope with the ever increasing disparity in speed between the CPU and the memory. For shared-memory multiprocessors, using caches is more appropriate and necessary. Here the delay of a memory access is a combination of memory latency and interconnection network (including bus-based interconnections) latency, of which the network latency is even more significant. Caches with low miss ratios can reduce the memory delay close to that of the cache and help realize satisfactory CPU utilization; they also put less demand on network traffic, alleviate network contention on those memory accesses which have to be serviced by the shared memory.

Having a cache for each CPU raises the prospect of multiple copies of a data block in the system. Write operations can then cause data in other caches to become out-of-date. To ensure correct parallel execution, data consistency must be maintained among the caches. Cache coherence schemes or protocols have been proposed to solve this problem. For bus-based shared memory multiprocessors, a *snoopy* scheme is most often used [Archibald 86, Goodman 83, Katz 85, Lee 87, Papamarcos 84], where each cache controller independently monitors bus activities. If there is a memory access on the bus that would make one of its local data block copies differ from those on other caches or the main memory, the cache controller takes appropriate actions such as invalidation or updating of all copies of the data block to preserve data consistency. Snoopy schemes rely heavily on the broadcasting capability of bus and are unfit for systems with arbitrary interconnection networks. Due to its limited bandwidth, bus communication topology can only accommodate a small number of processing elements and is insufficient for building large, scalable shared-memory systems.

Scalable shared-memory multiprocessors usually use point-to-point or multi-stage networks to connect processing elements and the main memory [Chaiken 90]. They have message-passing as their main communication mechanism; efficient broadcasting is not available due to inherent implementation difficulties (for details see [Chaiken 90, James 90]). For such general architectures, various *directory* schemes have been proposed in [Agarwal 89, Censier 78, Chaiken 90, Chaiken 91, James 90, Lenoski 90a, Tang 76, Thapar 90]. These schemes maintain a data structure called the *directory* to store the *sharing structure*, i.e. the locations of the copies of each cached data block. When a data block is written, the main memory may send an invalidation or update message, depending on the specific coherence protocol, to each cache that holds a copy of the effected data block.

For an invalidation-based coherence scheme, a writing cache first becomes the only cache with a copy of the data block (i.e. the exclusive owner of the data block) by invalidating all other cache copies; subsequent writes to the same cache only need to be done locally on the cache, improving the efficiency of coherence operations. Exclusive ownership is terminated whenever another cache reads from or writes to the same data block, at which time if the copy has been written (is dirty) by the previous owning cache since its exclusive ownership was established, the dirty copy is sent back to the main memory and then read into the currently requesting cache. Such an invalidation-based coherence protocol is usually called *write-back with ownership*.

For an update-based coherence scheme, a writing cache does not need to first obtain the exclusive ownership of a data block. Instead data is written directly into the main memory and all caches that hold a copy of the data block. An update-based coherence protocol is usually called *write-through*.

The directory entry may contain other state information on the data block, such as dirty bit and exclusive ownership bit [Archibald 84, Censier 78, Chaiken 90, Papamarcos 84, Tang 76]. The directory is either centralized within the main memory or distributed among the processing elements. Distribution can be

done in two ways: distributed with the main memory among the processing elements[Agarwal 89] in a so-called *distributed main memory*, or distributed among the caches of the processing elements[Chaiken 91, James 90, Thapar 90].

This paper proposes a new distributed directory scheme that can carry out coherence operations in an efficient manner and is suitable for scalable large shared-memory multiprocessors. Moreover, for each type of cache operation, it has provably optimal time complexity. In section 7.2 we first review the major existing directory schemes. Section 7.3 presents the binary-tree directory scheme and discusses its time complexity for each cache operation. In section 7.4 we examine some other tree-like directory schemes, including a redundant-pointer list structure proposed previously in the literature. Compared to these other schemes, our approach is superior in space and time complexities. Section 7.5 concludes with a summary.

## 7.2 Directory Schemes

First we discuss how each cache does read and write, which is common for all directory schemes. Usually, a cache has a few state bits for each cached block, such as a validity bit, an ownership bit, and a dirty bit. The validity bit indicates whether a local cache copy of a data block is up-to-date (valid). The ownership bit indicates whether the local cache is the (exclusive) cache which is currently allowed to write to the data block; for any data block, there is always at most one cache whose ownership bit of that block is set. The dirty bit indicates whether the local cache copy of a data block is newer than the copy in the main memory, which means upon giving up ownership of or replacing this data copy, the local cache must send this block copy back to the main memory. These bits, particularly the last two, are usually used by invalidation-based coherence protocols.

For any coherence protocol, a read always proceeds if the cache's validity bit for the target data block is set. For an invalidation-based coherence protocol, a write can proceed if the cache's ownership bit for the target data block is set. For

an update-based coherence protocol, a write is always sent to the main memory and the update is relayed to all other caches that have a copy of the target data block.

When a cache has a read-miss (its validity bit is not set), it sends a read request to the main memory or the memory module which controls the requested data block (we will not differentiate these two cases from now on), which sends back, or asks a cache with the most recent copy to send back, a copy of the data block. For an invalidation-based coherence protocol, when a cache has a write-miss (its ownership bit is not set), it sends a write request to the main memory, which sends back the permission to set its ownership bit, and possibly a copy of the data block if it is missing. For an update-based coherence protocol, when a cache has a write-miss (here its validity bit is not set), it sends a write request to the main memory, which updates everyone else, and send back a new copy of the data block.

Between the time the main memory gets a read-miss/write-miss request from a cache and the time it replies, the main memory usually carries out some cache coherence operation. For read-miss, this includes recording the requesting cache as one of the caches holding a copy of the requested data block; and for invalidation-based protocols, it also resets the exclusive ownership bit of some other cache which had ownership of the data block, if there is one. On a write-miss with an invalidation-based coherence protocol, it includes invalidating the data block in all current holding caches, and, if there exists a owner cache for this block, resetting the exclusive ownership bit of that cache. Normally the main memory waits to receive acknowledgement from all caches involved in the coherence operation before replying to the original requesting cache[Chaiken 90]. An update-based coherence protocol requires that each write be sent to the main memory, which in turn relays the update to all the other holding caches. The details of coherence operations are directory-scheme dependent and may differ for protocol variations.

When a full cache has a miss, it must replace one of its blocks. Information concerning the replacement block needs to be sent to the main memory along

with the miss request, so that the main memory is aware that the replaced block is no longer in that cache. For invalidation-based protocols, if the cache currently has exclusive ownership of the replaced block, and the corresponding dirty bit is set, then the dirty data block needs to be sent back to the main memory.

There are primarily three popular directory schemes[Agarwal 89, Censier 78, Lenoski 90b, James 90]: *full-map* directories, *limited* directories, and *chained* directories. Their data structures are essentially a linear list. Some tree schemes were proposed recently [Haridi 89, Maa 91, Wallach 92, Wilson 87, Yang 90]. There was a brief description of a tree-like structure using redundant pointers on a chained directory[James 90], proposed as a possible standard for scalable coherent interface. We will examine it closely in section 7.4 and point out its inadequacies compared to our scheme, in terms of structure simplicity and time complexity for handling a cache miss, i.e., adding a new cache ( $O(\log N)$  v.s.  $O(1)$ ).

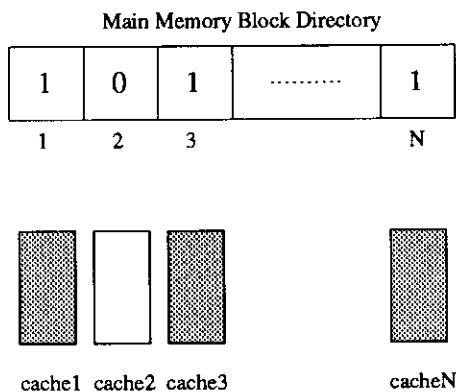


Figure 7.1: Full-map Directories

A full-map directory[Censier 78, Lenoski 90b] allows a data block to be cached simultaneously in all caches. It uses a presence vector of 0/1 bits in the main memory to represent the presence or absence of a data block in each cache, as is shown in Figure 7.1. For invalidation-based coherence protocols, a separate *dirty bit* is used to indicate if there is any cache with a more up-to-date copy of the data block than the one in main memory. When the dirty bit is set, exactly

one element in the presence vector can be set, which is for the cache with the most up-to-date version of the data block; that cache currently has the exclusive ownership of the data block. The presence vector and the dirty bit constitute the directory entry for that data block. If there are  $N$  caches in the system, the size of a directory entry is  $O(N)$ . Full-map has good performance for invalidation and update coherence operations. Its drawback is the large memory overhead of the directory entries.

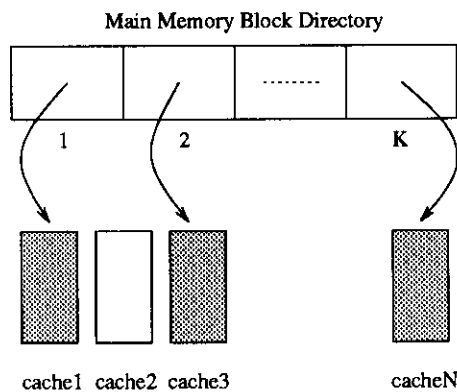


Figure 7.2: Limited Directories

To reduce the memory demand of directory entries, a limited directory scheme [Agarwal 89] was designed. It limits the number of caches that can simultaneously store a copy of the same data block to some number  $k < N$  by using  $k$  cache pointers, depicted in Figure 7.2. Its main memory overhead is  $O(k \log N)$ , as each element is now a cache ID. When a directory entry is already full and another cache requests to cache the data block, a previously cached copy must be invalidated, which can be chosen by a replacement policy. The limited directory scheme restricts sharing; highly shared data blocks can therefore experience poor performance. A combination of full-map and limited directory schemes is implemented in [Chaiken 91], where the full-map scheme is emulated in software. Both the full-map directories and the limited directories are centralized schemes in the main memory.

The chained directory scheme [James 90] distributes directory entry informa-

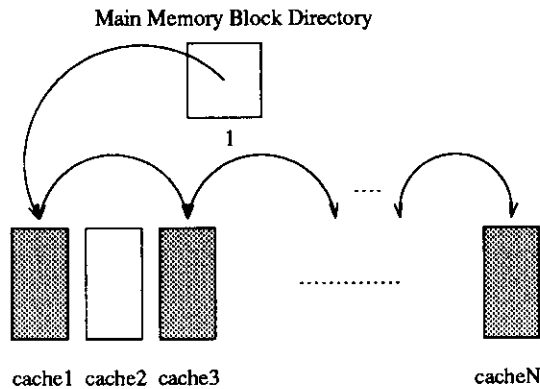


Figure 7.3: Chained Directories

tion among the caches that hold a copy of the concerned data block; these caches are chained linearly in a linked list, shown in Figure 7.3. The main memory has a pointer to the header cache of the list. There are minor variations between using singly-linked list[Chaiken 90] and doubly-linked list[James 90], and whether the main memory or the header cache on the list provides the most recent copy of the data block, but the main idea is the same[Chaiken 90, James 90, Thapar 90]. A cache always joins a linked list by becoming the new header. When a cache has a miss, it sends a request to the main memory; the main memory changes the header pointer to point to the new cache, and returns the ID of the old header cache to the requesting cache. Upon receiving the cache id, the requesting cache joins the cache list by making its forward pointer to the received cache ID. Coherence operations are done from the header through the linked list in serial fashion; invalidation removes the list. Cache replacement is handled by purging a cache from the list of the replaced block; with doubly-linked list, it is easily done in constant time  $O(1)$ [James 90]. Chained directory has only  $O(\log N)$  memory overhead for a cache pointer, making it applicable for large scalable systems. Compared to limited scheme, it does not impose limitation on sharing. The drawback is slow coherence operations: invalidation or update is done sequentially on the linked list, needing  $O(N)$  time.

The above directory schemes have the drawback of performance bottleneck



at the main memory, any cache missing a data block has to access the main memory to get a copy. To this end, Wilson[Wilson 87], Haridi[Haridi 89], Yang et al. [Yang 90] proposed bus-based hierarchical tree structures where each node is a group of processing or storing elements connected by a bus. Each element in the tree records which blocks are cached by its descendant elements[Wallach 92]. A processor in want of a data block sends its request up the tree until a copy is located. These schemes differ in whether the intermediate level only stores directory information[Haridi 89, Yang 90], and whether the main memory is attached at the top of the hierarchy[Wilson 87, Yang 90]. Wallach[Wallach 92] proposed a similar tree hierarchy scheme and its mapping onto k-ary n-cubes, with different hierarchies for different data blocks, further reducing the memory bottleneck.

All these hierarchical schemes have a fixed sharing structure containing all caches: at any time the tree may have many irrelevant caches. A requesting cache would have to send its message past several caches before finally locating a cache with a copy of the requested data block; cache invalidation/updating also may have to pass irrelevant caches on way to the destination caches. It is desirable to have a dynamic sharing structure that adjusts to changing data-sharing and contains only relevant caches.

Maa et al. [Maa 91] proposed two tree directory schemes that dynamically change and contain only the sharing caches. However, their tree structures are unchecked, hence can become unbalanced and in the worst case degenerate into a linear list, reducing to the chained or full-map scheme.

A good directory scheme should be free of these problems.

### **7.3 Balanced Binary-Tree Directory**

To achieve true scalability, the directory space must be bounded per block entry, i.e., only a limited number of pointers are employed in each directory entry, for both the main memory and for each individual cache. If we view the main memory and the caches as nodes in a directed graph, and the directory pointers

as arcs of the digraph, then the digraph of a scalable architecture has bounded out-degree for its nodes. The time complexity of message-broadcasting in such scalable architectures is that of broadcasting a message in the corresponding bounded-degree digraph, which clearly has a lower bound of  $O(\log N)$ .

**Lemma 7.1** *The time to propagate information in a bounded-degree  $N$ -node digraph is at least  $O(\log N)$ .*

**Proof.** Suppose a constant  $d > 1$  is the maximum value of the out-degrees of all nodes, and at time step 0 some node  $s$  is to send a message to  $N$  other nodes. Message-passing from one node to another directly connected node takes unit time. For simplicity, we assume that it takes zero time for a node to emit message to its out-bound channel, so there is no difference in time between sending a message to one out-bound channel and sending a message to all (at most  $d$ ) of its out-bound channels. Since this assumption could only underestimate the amount of time required by communication, it does not affect conclusions on the lower bound of communication time.

At time step 1, at most  $d$  distinct destination nodes have received the message, since  $s$  is one-hop away from at most  $d$  destination nodes. Using induction, at time step  $k$ , at most  $d^1 + d^2 + \dots + d^k = (d^{k+1} - d)/(d - 1)$  distinct destination nodes are reached. From  $(d^{k+1} - d)/(d - 1) \geq N$ , it gives  $k \geq \log_d((d - 1)N + d) - 1$ . As  $d$  is a constant, the lower bound on the number of time steps  $k$  to send messages to all  $N$  destination nodes is  $O(\log N)$ .  $\square$

In a shared-memory multiprocessor system with  $N$  caches, each cache pointer in the directories uses  $\log N$  binary bits to identify  $N$  caches. This is the minimal requirement on pointer bits. For any bounded-degree directory scheme, the minimum memory overhead per directory entry is therefore  $O(\log N)$ .

In order to obtain an  $O(\log N)$  communication time for invalidation and update in scalable shared-memory multiprocessors, a tree data-structure is a viable way to organize the sharing structure. Due to constant cache joining (a cache miss) and cache purging (a block replacement), i.e., addition and deletion on the sharing structure, the tree can degenerate into a linear list. To guarantee the

logarithmic time scale, the tree must be kept balanced. The issue is to efficiently update and balance the tree with continuous node addition and deletion. Deletion can be done simply in  $O(1)$  time, as shown below. The trick is with addition. A naive way of adding a node is to search from the tree-root down to a leaf node, taking  $O(\log N)$  time. Due to the high frequency of occurrence, node addition requires a more efficient method.

We will use a balanced binary-tree to represent data block sharing, in which both cache-miss and block-replacement take only  $O(1)$  time. Since the tree is balanced, invalidation and update only take  $O(\log N)$  time, which is optimal, by Lemma 7.1, for any directory scheme with bounded-degree for each directory entry. Therefore such balanced binary-tree scheme is an optimal directory scheme to represent the sharing structure for scalable shared-memory multiprocessors.

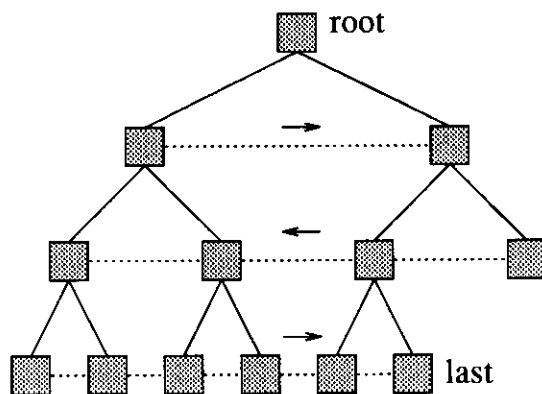


Figure 7.4: Balanced Binary-Tree

For each data block, there is a well-balanced binary tree structure whose nodes are caches that have a copy of this data block, as illustrated in Figure 7.4. In the directory entry for that data block, the main memory has a pointer to the root of the tree, and a pointer to the “last” leaf of the tree, which is the last leaf on the last level. The main memory directory entry also has an *oddness* bit to indicate whether the tree height is odd or even: 1 for odd height and 0 for even height. Each cache node has five cache pointers—parent, left child, right child, left sibling, right sibling. Sibling pointers are depicted by dashed lines

in the diagram. The memory overhead of each directory entry is  $O(\log N)$  for main memory and every cache; the directory entry information for a data block is distributed among the main memory and its resident caches.

We first explain how to add and delete one tree node, then discuss how to handle simultaneous node additions and deletions by locking the directory entry (root directory) on the shared main memory.

**Node Addition.** For brevity we will omit discussing the details of how to set various state variables such as validity bits and ownership bits, and whether most recent block copy is provided by the main memory or a cache. While variations on these points can yield many different protocols, they are orthogonal to the sharing data structure under study.

Initially the tree is empty: the data block is not cached anywhere. When a cache has a miss on this block, it sends a request to the main memory. A single-node tree is created with the main memory setting both its *root* pointer and *last* pointer to the requesting cache, and setting the oddness bit to 1. The data block and a null pointer are returned to the cache, which sets all five of its pointers to null. Subsequently, each added node becomes the new *last* node, and the binary tree grows level by level, filling up one level before advancing to the next one. The arrows in Figure 7.4 show the direction of node expansion on each level of the tree. The direction (left or right) to fill new nodes on the bottom level of the tree is easily determined by the tree's current oddness bit. For oddness bit 0 (even tree height), the nodes are added from left to right, while for oddness bit 1 (odd tree height), the nodes are added from right to left. In order to add a new node, the question is how to efficiently find the correct parent node.

If the bottom level is already full, addition is easy; the old *last* node is the parent of the new node. See Figure 7.5. In this case the main memory switches its oddness bit. If the parent node of the old *last* tree node has only one child, addition is simple too; that parent node is exactly the needed parent node for the new node, as shown in Figure 7.6. We describe in detail the general case where the old *last* node is neither on the boundary of the bottom level, nor does it share

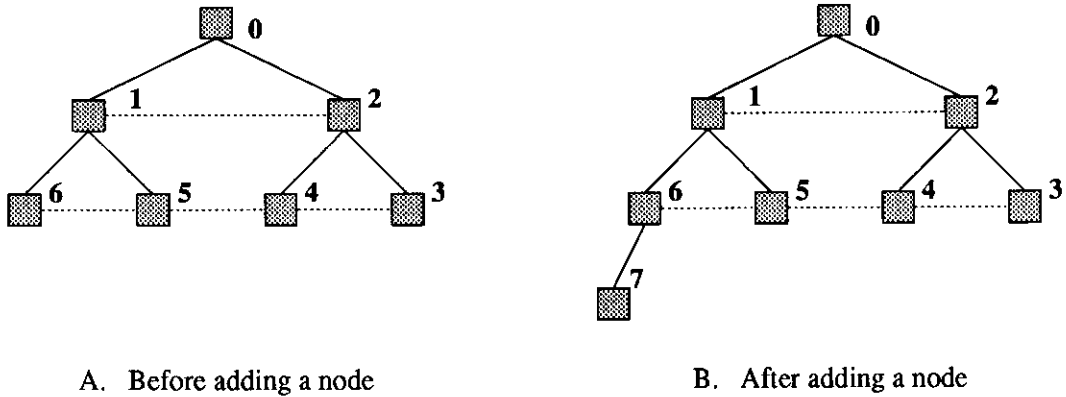


Figure 7.5: Node Addition: New Level

parent node with the new node. The basic idea is to find the proper sibling of the parent of the last node, which will be the parent of the newly added node.

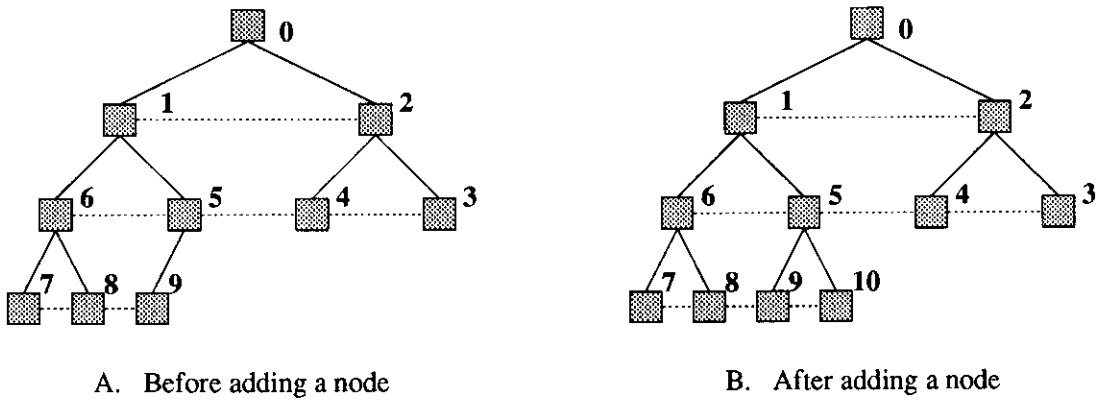


Figure 7.6: Node Addition: Common Parent

When a new cache (say cache 11) experiences a miss on this block, it sends a *miss* request to main memory; main memory sets the *last* node pointer to the new cache, and returns the old *last* node (cache 10) and the oddness bit of tree height (0) to the new cache; see Figure 7.7A. The new cache sets its left sibling pointer to received node pointer (cache 10), and sends a *parent* request and the zero oddness bit to that node (cache 10), which sets its right sibling pointer to the new cache (cache 11), and returns with a pointer to its parent (cache 5). Now the new cache sends a *sibling* request and the zero oddness bit to the received node

pointer (cache 5); that node (cache 5) sends back its right sibling pointer (cache 4). The new cache sets its parent pointer to the received node pointer (cache 4), and sends a *child* request and the zero oddness bit to its newly adopted parent node (cache 4). Upon receiving the message, the new parent node (cache 4) sets its left child pointer to the new cache (cache 11). The new cache completes the addition by sending an acknowledgement back to the main memory. Now a new tree is created, shown in Figure 7.7B. If the oddness bit was 1, then “left” and “right” should be switched in the above description.

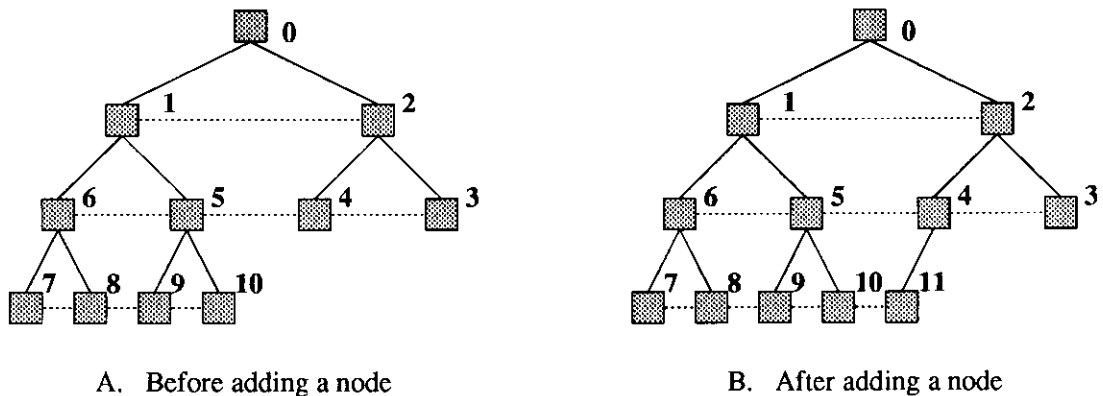


Figure 7.7: Node Addition: Distinct Parents

To add a cache to the binary-tree sharing structure, the worst case requires the new cache to communicate with four parties: main memory, last node, last node’s parent, and real parent. It may take up to a total of ten messages (the last two messages are to and from the main memory to release the lock on directory entry, see **Locking** below), thus the complexity in latency is  $O(1)$ . To avoid message-queue deadlocks, messages should not be directly forwarded by the main memory or the caches, as indicated in [James 90].

Notice that we did not differentiate between a read-miss and a write-miss. If the write protocol is invalidation-based, then a write-miss cache does not join the tree; instead the main memory nullifies the tree by sending an invalidation to the tree root, from which propagates it down to every descendant node, and acknowledgements are propagated upward in the tree back to the root, then the

root sends its acknowledgement to the main memory. If the write protocol is update-based, the write-miss cache first joins the tree, then sends each write to the main memory which relays it to every tree node.

**Node Deletion.** Deletion occurs when a cache voluntarily discards a cached data block, due to miss-incurred block replacement. When the node to be deleted is the *last* node, it is quite simple. If it is not the *last* node, we just move the *last* node to the position of the deleted node.

Specifically, the deletion node sends a *deletion* request to the main memory, which decides that it is not the *last* node, sends back to the requesting node a pointer to the *last* node. Now the deletion node sends a *substitute* message together with its five pointers (parent, left/right child/sibling) to the *last* node. The *last* node informs its parent and sibling (there is at most one) to cut their links, and substitutes its five pointers with the corresponding five pointer values just received, effectively taking the place of the deletion node in the tree. For all the non-null pointer values, the nodes they point to must properly adjust their pointers from the deletion node to the new substitution node; this can be done with the substitution node sending a *adjust* message, together with its node ID and the deletion node ID, to these adjacent nodes; these nodes can find the correct pointer to change by comparing with the deletion node ID, and change it to the new substitution node ID.

After that, the substitution node returns to the deletion node its sibling pointer if it is not null, or its parent pointer if it has no sibling; the deletion node in turn sends the new *last* node ID back to main memory. In the first case, the sibling is the new *last* node. In the second case, the old *last* node was the only node on the bottom level, and its parent becomes the new *last* node; the main memory switches the oddness bit for the tree height is now decremented by one. Now the main memory sends acknowledgement to the deletion node, informing the completion of deletion.

To remove a cache from the binary-tree sharing structure, the worst case requires the deletion cache to communicate with two partners: the main memory

and the *last* (substitution) cache. The substitution cache in the worst case needs to communicate with five partners: the parent, two siblings, and two children of the deletion node. The time complexity in latency remains  $O(1)$ . Like node addition, to avoid message-queue deadlocks, messages for node deletion are not forwarded by the main memory or caches.

In the chained directory scheme, multiple nodes can be deleted in parallel, as long as they are not adjacent with one another in the list, while in the binary-tree scheme, multiple deletions must be carried out sequentially. As each node deletion involves the main memory, this, however, does not improve the overall time complexity for the chained directory scheme. The reason is that node addition in both schemes is sequential (with the rare exception of *request combining* for chained directories [James 90], which is rather unlikely to occur); as the number of node deletions is bounded by the number of node additions, the time complexity of node addition dominates that of node deletion – meaning that any speed-up in node deletion does not improve the overall time complexity of the combination of node addition and node deletion.

**Locking.** To avoid data structure corruption due to race conditions, the main memory locks the directory entry of a data block on which there is a cache addition or deletion activity going on. Any request from other caches on the directory entry is rejected or queued to be serviced later. The binary tree updating messages (*parent, child, sibling*) have higher priority than normal cache operations. A cache waiting on an outstanding request still participates in tree updating: whenever it receives a tree updating message, it properly adjusts one of its tree pointers. Since any change to the tree goes through the main memory, concurrent addition and deletion are serialized.

**Drawbacks.** As any change in tree structure involves the main memory, it can become a performance bottleneck. Static distribution of main memory space among a number of memory modules or processing elements [Chaiken 91] may alleviate the bottle-neck problem, provided data usage is evenly spread among



all blocks in the address space. Skewed data usage is better served by some kind of dynamic distribution of “home” locations of the data blocks[Haridi 89, Wallach 92, Wilson 87, Yang 90], in order to balance the traffic demand on the memory modules, processing elements, or connecting networks. There are issues of when and how to do the redistribution under changing circumstances. For example, the Data Diffusion machine[Haridi 89] takes a simple approach: it relocates a data block only when its last copy is being replaced; the new home node is found at another leaf node inside the shallowest subtree that also contains the old home node, by traversing the tree bottom-up to its root. In general, these issues are hard to solve satisfactorily. Bus-based hierarchical architectures also suffer the performance hit due to bottleneck at or near the top of their hierarchies[Haridi 89, Wilson 87]. Using different hierarchies for different data blocks[Wallach 92] is analogous to distributing the main memory among many memory modules or processing elements.

While the asymptotic time complexities of the cache operations are optimal, in practice the accumulated cost of building up a big tree is still expensive. It may not be suitable for invalidation-based coherence protocols that completely nullify the tree each time some CPU issues a write to the data block. Instead, it is more appropriate for update-based coherence protocols, which do not change the sharing structure on writes but keep using the same tree structure for propagating new data values. In this way the cost of building the tree is amortized by the saving in coherence operations. An update coherence protocol is especially suitable for some event synchronization; for example, all processors waiting on a barrier variable are released efficiently with an update-based write to the barrier[Lenoski 90a]. Update is also perceived to have better performance than invalidation on normal data blocks[McCreight 84]. Applications with high degree of read-sharing and relatively infrequent write-sharing are good candidates for applying an update-based coherence protocol, and hence our binary tree directory scheme.

**Synchronization.** Goodman, et al. [Goodman 89] proposed an efficient method of FIFO access to synchronization variables by linearly queueing requests. The binary tree scheme provides efficient implementation for locks, in a manner similar to a linked list queue [James 90]. With the sibling and children pointers, a lock can be passed sequentially from the root to the last node then through each level to all tree nodes and finally back to the root, along a zig-zag and bottom-up path, reverse of the itinerary by which node addition is done. There is the danger of node-deletion causing the last node to miss its current turn to gain locking. To make matters worse, before the lock is passed to this poor node, it becomes last node and another node is being deleted, so it does substitution again and consequently loses its next turn, etc. However, this kind of lock-missing does not go on forever: each time a node (as the current last node) substitutes a deleted node, its distance to the root of the tree, measured by the length of the zig-zag bottom-up path, is strictly decreased. Since a path has finite length, any node can only do substitution for a finite number of times, before it finally becomes the root itself. Thus it will get its locking in finite time—no locking starvation. Notice that passing the lock in the opposite direction (i.e., from the root down through each level before finally reaching the last node and back to root) can indeed cause starvation: when the last two nodes on the bottom level alternate in a sequence of add-lock-unlock-delete operations, the lock can be hogged by them forever.

As mentioned earlier, event synchronization such as barrier release is done most naturally by our binary tree scheme.

**Performance Comparison With Full-Map Scheme.** Full-map directory provides a performance upper bound for centralized directory schemes [Chaiken 90]. We compare our binary tree with full-map in terms of invalidation/update speed. For both main memory and caches, let the average inter-transmission delay between sending a message to two out-bound channels be  $t_i$ , the average processing time of invalidation and update message at each cache be  $t_p$ , and the average point-to-point transmission delay among the main memory and the caches be  $t_x$ .

For simplicity, we shall assume that the message-processing time of acknowledgement is negligible; this will not fundamentally change the comparison.

Denote by  $T$  the average overall delay between the beginning (main memory issues the operation) and the end (main memory receives acknowledgements from all caches involved) of a invalidation or update coherence operation. For full-map directory with  $N$  caches,

$$T_{\text{full-map}} = (N - 1)t_i + t_x + t_p + t_x = (N - 1)t_i + 2t_x + t_p$$

where  $(N - 1)t_i$  is inter-transmission time before the message to the last cache is sent,  $t_x$  is the transmission time to the last cache,  $t_p$  is the processing time at the last cache, and  $t_x$  is the acknowledgement from the last cache. When the last acknowledgement arrives at the main memory, it has already received acknowledgements from the other  $N - 1$  caches. For binary tree directory with  $N$  caches, the longest delay occurs to the leaf nodes on the lower-right corner of the binary tree:

$$\begin{aligned} T_{\text{bin-tree}} &\leq t_x + \lceil \log N \rceil (t_p + t_i + t_x) + \lceil \log N \rceil t_x + t_x \\ &= \lceil \log N \rceil (t_i + 2t_x + t_p) + 2t_x \end{aligned}$$

Here the first  $t_x$  is the transmission time from the main memory to the tree-root cache;  $(t_p + t_i + t_x)$  is the delay between the receiving time of an internal node and that of its right child (the sum of the processing time at the parent, the inter-transmission time for the right child, and the transmission time), this happens at each of the  $\lceil \log N \rceil$  tree levels; last two terms are transmission delays for acknowledgements to go bottom-up to the root cache, and back to the main memory.

$T_{\text{full-map}}$  grows with  $O(N)$ , while  $T_{\text{bin-tree}}$  grows with  $O(\log N)$ , and

$$T_{\text{full-map}}/T_{\text{bin-tree}} \geq (N - 1)t_i / (\lceil \log 2N \rceil (t_i + 2t_x + t_p)).$$

When  $2t_x + t_p \leq (N - \lceil \log N \rceil - 1)t_i / \lceil \log N \rceil$ ,  $T_{\text{bin-tree}} \leq T_{\text{full-map}}$ . Compared to the full-map scheme, binary tree scheme could be competitive in performance for large systems.

**Performance Comparison With Chained Directory Scheme.** Obviously the chained directory scheme has the worst performance. For the sake of completeness, we compare our balanced binary-tree scheme with it as well. Use the same notation as above,

$$T_{\text{chained}} = N(t_x + t_p) + t_x$$

where  $N(t_x + t_p)$  is the total transmission and processing time from the main memory to the last cache on the chain, and the last  $t_x$  is the transmission time of the final acknowledgement from the last cache on the chain back to the main memory. Since  $t_i \ll t_x$ ,

$$\begin{aligned} T_{\text{bin-tree}} &\leq 3\lceil \log N \rceil (t_x + t_p) + 2t_x \\ &\leq 3(\lceil \log N \rceil + 1)(t_x + t_p) \\ &= 3\lceil \log 2N \rceil (t_x + t_p) \end{aligned}$$

$T_{\text{chained}}$  grows linearly with  $N$ , while  $T_{\text{bin-tree}}$  grows logarithmically with  $N$ , and

$$T_{\text{chained}}/T_{\text{bin-tree}} \geq N/3\lceil \log 2N \rceil.$$

Compared to the chained directory scheme, balanced binary-tree scheme is much better.

To sum up, like any directory scheme, our binary tree scheme is suitable for general interconnection networks without relying on a broadcasting mechanism. Unlike previously proposed tree schemes, which either has a fixed structure and contains many non-sharing caches, or can easily degenerate into linear lists, our scheme guarantees balanced structure without wasting any time on irrelevant caches. Compared to the chained directory scheme, it provides the same scalability by distributing directory entry information among caches with unlimited data sharing, while its logarithmic tree height allows much faster coherence operations than chained directory. For very large scale architectures with thousands of processing elements, our tree scheme could even compete with the full-map

scheme in performance, when the large value of the number of nodes  $N$  makes the point-to-point transmission time  $t_x$  among caches comparable with  $O(N/\log N)$  times the inter-transmission delay  $t_i$  at each cache node. We also prove that among all directory schemes with bounded-degree for each directory entry, our balanced binary-tree is an optimal scheme in terms of time complexity for each cache operation.

**Generalization to  $d$ -ary Tree.** In passing we note that there is nothing intrinsic about the tree structure being binary. The above discussion can be generalized to any  $d$ -ary tree with some constant integer  $d > 1$ . Each tree node now has  $d$  children pointers, while the numbers of parent pointer and sibling pointers remain unchanged. Node addition and deletion are carried out in exactly the same way, except that now each parent node can accommodate more children nodes, a new addition node is more likely to share parent with the old *last* node, making node addition faster by alleviating the usual searching sequence of “*last* to parent to sibling” in looking for the parent of a new node. On the other hand, node deletion becomes slower, as the substitution node has more pointers to update. Since the number of node deletions is at most the number of node additions, a bushy  $d$ -ary tree seems to have better performance than the binary-tree. Of course, their asymptotic time complexities are the same.

## 7.4 Discussion on Other Tree-Like Schemes

**Redundant List.** James, et al. briefly described a tree-like directory scheme [James 90], as depicted in Figure 7.8. It uses redundant pointers in the linked list to reduce coherence operation delays from linear to logarithmic. But the details of node addition and deletion were not given.

If we number the nodes by  $0, 1, \dots, N - 1$  from the header on the left to the tail on the right, the interconnection can be described as follows:

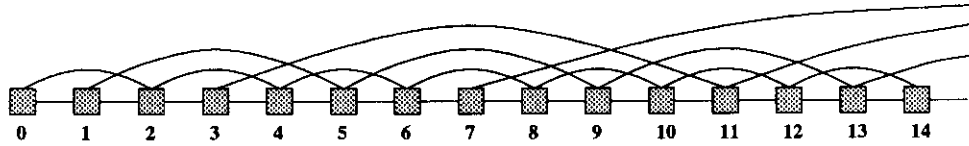


Figure 7.8: List with Redundant Pointers

1	0	1	2	3	...
2	0	2	4	6	...
4	1	5	9	13	...
8	3	11	19	27	...
16	7	23	39	55	...
32	15	47	79	111	...
⋮	⋮	⋮	⋮	⋮	⋮

Each row to the right of the vertical line is a sequence of nodes that are linearly connected; the first column is the difference in ID numbers between consecutive nodes. Generally, nodes  $2^{m-1} - 1, 2^{m-1} - 1 + 2^m, 2^{m-1} - 1 + 2^m 2, 2^{m-1} - 1 + 2^m 3, \dots$  are linearly connected. A node  $n = 2^{m-1} - 1 + 2^m l$  has up to four connected neighbors:  $n - 1, n + 1, n - 2^m, n + 2^m$ ; here  $m$  is one plus the number of consecutive 1's in the lower order of binary  $n$ .

Node deletion can be done by substitution with the last node, as we did above. Using double links for each connection, this takes  $O(1)$  time. To add a new node, backward links need to be created to two predecessors. One is to the tail of the list, which is easy; the other one is to some node in the middle of the list, which is difficult. To add, say, node  $2^m + 2^{m-1} - 1$ , one has to find node  $2^{m-1} - 1$ , for which the searching takes at least  $O(m)$  time. The time complexity for adding a node to such a structure with  $N$ -node is therefore  $O(\log N)$ . However, in order to decide which existing nodes are to be connected with a new node, the main memory has to keep track of the number  $N$  of current nodes in the list, requiring a counter of  $\log N$  bits per directory entry. While this does not change the  $O(\log N)$  space complexity, it does incur more directory overhead on the main memory than the

balanced tree scheme does.

Another way might be to keep pointers to all the “loose” nodes in the current list, that do not have four neighbors yet, in the main memory; when a new node is to be added, the farthest predecessor node can be found in  $O(1)$  time. However, this requires a memory overhead of  $O(\log^2 N)$  for each directory entry in the main memory (for up to  $O(\log N)$  loose node pointers, each using  $\log N$  bits), which is less scalable.

**Perfect Shuffle.** One might consider the perfect shuffle structure[Stone 71] as another alternative. Shown in Figure 7.9, the interconnections are[Kindervater 91]  $(0, 1)$ ,  $(N-2, N-1)$ , and  $(i, 2i)$ ,  $(i+N/2, 2i+1)$ ,  $(2i, 2i+1)$ , for  $i = 1, \dots, N/2-2$ . It has a  $O(\log N)$  bound on the length of its paths with no repeating nodes, so invalidation and update only take time  $O(\log N)$ . The difficulty is with node addition and deletion, which requires the changing of  $O(N)$  pointers on all the connections  $(i+N/2, 2i+1)$ , due to the change of  $N$ 's value. In addition, the number  $N$  of current nodes must also be kept by the main memory, using  $O(\log N)$  bits per directory entry.

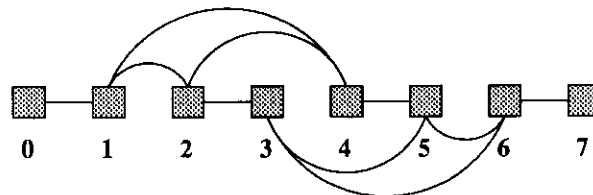


Figure 7.9: Perfect Shuffle List

Therefore, our tree scheme, compared to the above tree-like schemes, has the following advantages:

1. it has a simpler structure that is easy to understand and implement.
2. it needs  $O(\log N)$  time for invalidation/update, the minimum for bounded-degree directory entry schemes.
3. it needs minimum time  $O(1)$  for addition and deletion. Without  $O(\log^2 N)$

memory overhead, the redundant-pointer list scheme cannot achieve  $O(1)$  addition time. The perfect shuffle scheme requires  $O(N)$  addition time.

4. it uses the bounded-degree space of  $O(\log N)$  for each directory entry on both main memory and caches. The redundant-pointer list scheme requires  $O(\log N)$  space for  $O(\log N)$  addition time, and  $O(\log^2 N)$  space for  $O(1)$  addition time.
5. it only uses one oddness bit for each directory entry on the main memory, while the redundant-pointer list scheme requires a  $\log N$ -bit counter to keep track of the number of nodes in the list.

## 7.5 Summary

We propose a balanced binary-tree directory scheme as an alternative to represent the block sharing structure of shared-memory multiprocessors. It is scalable such that unlimited number of caches can share the same data block, with the bounded-degree directory space  $O(\log N)$  for each block entry of the directory table in the main memory and each cache. All cache operations are done with provably optimal time complexity: cache-miss and cache replacement use  $O(1)$  time, while coherence operations of invalidation and update use  $O(\log N)$  time. Compared to some previous tree directory scheme, it never degenerates into a linear list; since only the sharing caches are contained in its sharing structures, its trees are smaller (hence shorter) in comparison to those of some previous tree scheme, making faster coherence operations. Compared to other tree-like schemes such as redundant list, its structure is simpler and easier to implement, and cache-misses are handled much more efficiently. For update-based coherence protocols where the sharing structure is not destroyed by invalidation, our tree scheme is a promising choice.



# Chapter 8

## Conclusions

This chapter summarizes our results in analysis, design, and simulation of cache memories, and proposes future areas of research in the simulation of cache systems.

### 8.1 Summary

Performance evaluation is very important to the effective design of memory systems. The parameters considered in designing memory hierarchies include the number of levels, level capacity, placement algorithm (set associativity), block size, replacement algorithm, fetching method (pre-fetching v.s. demand-fetching). This thesis has concentrated on two-level cache memories and the LRU replacement algorithm. Cache memory is the most successfully implemented memory hierarchy in practice, and the LRU is the most interesting and studied replacement algorithm, and is known to perform well in many applications.

We try to extend previous stack evaluation methods to new areas, including one-pass evaluation of write-back LRU caches for multiple block-sizes, and one-pass evaluation of multiprocessor LRU caches with invalidation-based coherency protocols for multiple set-associativities. For the former, a vector of dirty level variables is used on each entry in the minimum block-size LRU stack. Each

individual dirty level variable is always attached to the corresponding top-most stack entry. Using this approach, stack evaluation of a memory reference trace produces hit ratios and write ratios of write-back LRU caches for multiple block-sizes in a single pass over the reference trace. For the latter, special marker entries are used in the stack to represent data blocks deleted by an invalidation-based cache coherence protocol. A method of marker-splitting is employed when a data block below a marker in the stack is accessed. Using this method, stack evaluation of a memory reference trace yields hit ratios for all cache sizes and set associativities of multiprocessor caches in a single pass over the reference trace.

To the question of whether other stack algorithms would permit an efficient all set-associativity stack evaluation like LRU, we formally show that LRU is the only such stack algorithm that do not base replacement decisions on the numerical value of a block address. The conclusion can be generalized to any set associative caching, as long as it has more than two different set mapping schemes. If the set mapping schemes include all possible block groupings, then LRU is the only such algorithm among all stack algorithms, even if they do consider the value of a block address in their replacement decisions.

We propose a special realization of multilevel staging hierarchies with an arbitrary stack algorithm. It maintains staging properties, and provides quick up-staging of data blocks from lower level to higher level in the hierarchy. The staging property permits a multilevel hierarchy to be evaluated by standard stack evaluation methods on reference traces, originally only applied to two-level hierarchies. One-pass processing of the stack algorithm over a reference trace gives hit ratios and write ratios for multilevel hierarchies with any number of levels and any capacity on each level.

In analytical modeling, we use the Independent Reference Model to study bus-based shared-memory multiprocessor LRU caches. A general method of modeling the cache states as a Markovian chain is presented, whose solution for steady state distribution probabilities can be solved by standard numerical solution methods. Because of cache interactions through block invalidations, it is difficult to get

a closed solution form; some simple bounds on the solution are derived. We approach the problem in another way by looking at the LRU stack of a cache; the distribution of state probabilities of each stack entry in equilibrium can be solved systematically with the help of recursion and conditional probabilities. It is desirable to do simulation experiments to verify the analysis.

In the area of cache system design, we propose using a balanced binary-tree directory scheme for the sharing structure of shared-memory multiprocessors. It is truly scalable without any limitation on the number of caches that can share the same data block, using only a bounded-degree directory space per block entry of directory table. Cache operations have provably optimal time complexities. Compared to previously proposed tree schemes, its shape is uniform and its height is minimum, achieving coherence operations more efficiently. Compared to a tree-like redundant-list scheme proposed as a possible standard for scalable coherent interface, its structure is simpler and easier to implement, handling cache-misses more efficiently. For any update-based coherence protocol where the sharing structure is not destroyed by a write operation, it is a promising choice.

## **8.2 Future Work**

Cache memory evaluation is a well trodden research area, yet there are still many open and very hard questions yet to be resolved. The important role that caches continuously play in computers makes their satisfactory solution essential.

### **8.2.1 Simulation methodology**

The traces that we use often correspond to a relatively short period of execution. Due to the high speed of present computers, the length of a trace tends to be extremely long even for a short period of running time [Smith 82]. In order to study the reference behavior for a reasonable amount of running time, very long traces are needed. The time needed to collect these traces and to process them

will in turn grow very long, so will the disk space needed to store them become very large. Qualitative but more efficient simulation methods such as on-the-fly and statistical sampling therefore require further study[Baer 91].

### **8.2.2 Validity of multiprocessor trace simulation**

As to the validity of using trace data collected on one architectural configuration to simulate another configuration, no definite study has been done yet. For stack evaluation, it boils down to how “universal” the trace data is; i.e., how accurately the trace data, collected on a target machine with specific cache sizes, may be used to predict the performance of the same target machine but with different cache size configurations.

It is worthwhile to do detailed timing simulations and find out under what conditions the traces collected with different cache sizes may be used by stack evaluation to accurately predict the performance of one another[Smith 93]. The outcome could very well depend on the type of application program being traced. Characterizing which application programs are suitable for stack evaluation needs extensive data collection and experimentation.

### **8.2.3 Parallel stack simulation**

Though an efficient simulation method, stack evaluation still takes a lot of time on very long traces. It is interesting to explore parallelism in stack evaluation and look into possibilities of using multiprocessor computers or multicomputers in carrying out parallel simulation.

One approach is to divide a long trace into  $m$  equal parts and run stack simulation on a  $m$ -PE multicomputer. The combination of their hit ratios are lower bounds on the true hit ratios of the original trace, since except the first partition, every later partition begins with a “cold-start” and will overestimate its miss ratios. On the other hand, one can convert the  $i$ -th infinite stack distances of each later partitions to stack distance  $i$ , then the final hit ratios will be an

upper bound on the true hit ratios of the original trace, since every later partition begins with the assumption that all blocks to be referenced in this partition have all been referenced in some earlier partition, and the distance conversion renders the *best-case* “warm-start” for this partition.

The problem here is trace partition and error estimation. Statistical methods such as sampling may be combined with this kind of approach to get better estimation.

How to effectively run simulations on multiprocessors, in view of their ever-increasing availability, is an important topic and deserves further research.

#### **8.2.4 Application paradigms**

There exist an array of cache evaluation methods: analytical, trace-driven, and software-based emulation. When and how to best use each method remains *ad hoc* at best. One method may very well be more suitable for some type of applications. Some applications may require a hybrid approach that combines several different methods to compensate one another and get more accurate estimations. Categorizing various classes of applications according to their suitability for different evaluation methods is an important problem that has to be addressed in the future.

# Bibliography

- [Agarwal 86] A. Agarwal, R. L. Sites, M. Horowitz. "ATUM: A New Technique for Capturing Address Traces Using Microcode", *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 119-127, June 1986.
- [Agarwal 88] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence", *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 280-289, June 1988.
- [Agarwal 89] A. Agarwal, M. Horowitz, J. Hennessy. "An Analytical Cache Model", *ACM Transactions on Computer Systems*, Vol. 7, No. 2, pp. 184-215, May 1989.
- [Aho 74] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.
- [Archibald 84] J. Archibald, J.-L. Baer, "An Economical Solution to the Cache Coherence Problem", *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 355-362, June 1984.
- [Archibald 86] J. Archibald, J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM*

*Transactions on Computer Systems*, Vol. 4, No. 4, pp. 273-298, November 1986.

- [Baer 89] J.-L. Baer, W. Wang, "Multilevel Cache Hierarchies: Organizations, Protocols, & Performance", *Journal of Parallel and Distributed Computing*, **6**(3), pp. 451-476, May 1989.
- [Baer 91] J.-L. Baer, *private communication*, December 1991.
- [Bechtolsheim 90] A. V. Bechtolsheim, E. H. Frank, "Sun's SPARCstation 1: A Workstation for the 1990's", *Proceedings of the Spring '90 IEEE COMPCON*, pp. 184-188, February 1990.
- [Belady 74] L. Belady, F. Palermo, "On-Line Measurement of Paging Behavior by the Multivalued MIN Algorithm", *IBM Journal of Research and Development*, Vol. 18, No. 2, pp. 2-19, April 1974.
- [Birrell 84] A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, February 1984.
- [Bitar 91] N. Bitar, E. Shienbrood, "Mach: Architecture and Implementation", *Lecture Notes*, University of California, Los Angeles, November 1991.
- [Bitner 79] J. R. Bitner, "Heuristics That Dynamically Organize Data Structures", *SIAM J. Computing*, Vol. 8, No. 1, pp. 82-110, February 1979.
- [Censier 78] L. M. Censier, P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, Vol. C-27, No. 12, pp. 1112-1118, December 1978.

- [Chaiken 90] D. Chaiken, C. Fields, K. Kurihara, A. Agarwal, "Directory-Based Cache Coherence in Large-Scale Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 49-59, June 1990.
- [Chaiken 91] D. Chaiken, J. Kubiatoicz, A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme", *Proceedings of the ASPLOS IV*, pp. 224-234, April 1991.
- [Cheong 88] H. Cheong, A. V. Veidenbaum, "A Cache Coherency Scheme with Fast Selective Invalidation", *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 299-307, May 1988.
- [Cherian 89] M. M. Cherian, "A Study of Backoff Barrier Synchronization", MIT/LCS/TR-452, June 1989.
- [Courtois 86] P. Courtois, P. Semel, "Bounds on Conditional Steady-state Distributions in Large Markov and Queueing Models", *Teletraffic.Analys.& Comp.Perf.Eval.*, O. Boxma (Eds), pp. 499-520, North-Holland, 1986.
- [Cross 93] D. Cross, R. Drefenstedt, J. Keller, "Reduction of network cost and wiring in Ranade's butterfly routing", *Information Processing Letters*, Vol. 45, No. 2, pp. 63-67, February, 1993.
- [Dan 90] A. Dan, D. Dias, P. S. Yu, "The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment", *Proceedings of the 16th VLDB*, pp. 419-431, August, 1990.
- [Dan 93] A. Dan, P. S. Yu, "Performance Analysis of Buffer Coherence Policies in a Multisystem Data Sharing Environment", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 3, pp. 289-305, April 1993.



- [Denning 68] P. J. Denning, "The Working Set Model for Program Behavior", *CACM*, Vol. 11, No. 5, pp. 323-333, May 1968.
- [Denning 70] P. J. Denning, "Virtual Memory", *ACM Computing Surveys*, 2(3), pp. 153-190, September 1970.
- [Dubois 82] M. Dubois, F. Briggs, "Effects of Cache Coherency in Multiprocessors", *IEEE Transactions on Computers*, Vol. C-31, No. 11, pp. 1083-1099, November 1982.
- [Dubois 86] M. Dubois, C. Scheurich, F. Briggs, "Memory Access Buffering in Multiprocessors", *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 434-442, June 1986.
- [Dubois 88] M. Dubois, C. Scheurich, F. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors", *IEEE Computer*, pp. 9-21, February 1988.
- [Dygas 86] M. Dygas, "A Singular Perturbation Approach to Non-Markovian Escape Rate Problems", *SIAM J. Appl. Math.*, Vol. 46, No. 2, pp. 265-298, April 1986.
- [Edenfield 90] R. Edenfield, B. Ledbetter, R. McGarity, "The 68040 On-Chip Memory Subsystem", *Proceedings of the Spring '90 IEEE COMPCON*, pp. 264-269, February 1990.
- [Feller 71] W. Feller, *An Introduction to Probability Theory and its Applications*. Vol. 1, 3rd Ed., John Wiley & Sons, New York, 1971.
- [Flajolet 87] P. Flajolet, D. Gardy, L. Thimonier, "Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search", *T.R. 87-1176*, DCS, Stanford University, August 1987.

- [Franaszek 74] P. A. Franaszek, T. J. Wagner, "Some Distribution-Free Aspects of Paging Algorithm Performance", *JACM*, Vol. 21, No. 1, pp. 31-39, January 1974.
- [Fricker 91] C. Fricker, P. Robert, "An Analytical Cache Model", *Rapports de Recherche No. 1496*, INRIA, Chesnay Cedex, France. September 1991.
- [Gecsei 74] J. Gecsei, "Determining Hit Ratio for Multilevel Hierarchies", *IBM Journal of Research and Development*, Vol. 18, No. 4, pp. 316-327, July 1974.
- [Goodman 83] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 124-131, June 1983.
- [Goodman 89] J. R. Goodman, M. K. Vernon, P. J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors", *Proceedings of the ASPLOS III*, pp. 64-75, 1989.
- [Gopal 90] M. Gopal, B. Kadaba, R. Sultan, "Analysis of Caching in Distributed Directory Algorithms", *IBM Res.Rep.RC 16277*, November 1990.
- [Guibas 78] L. J. Guibas, R. Sedgewick, "A Dichromatic Framework for Balanced Trees", *Proceedings of the 19th Annual Symposium on foundations of Computer Science*, pp. 8-21, October 1978.
- [Gupta 91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, W.-D. Weber, "Comparative Evaluation of Latency Reduction and Tolerating Techniques", *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 245-263, May 1991.

- [Greenberg 86] A. Greenberg, A. Weiss, "A Lower Bound for Probabilistic Algorithms for Finite State Machines", *J.Comp.Syst.Sci.*, Vol. 33, No. 1, pp. 88-105, August 1986.
- [Guibas 78] L. J. Guibas, R. Sedgewick, "A Dichromatic Framework for balanced Trees", *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pp. 8-21, 1978.
- [Haridi 89] S. Haridi, E. Hagersten, "The Cache Coherence Protocol of the Data Diffusion Machine", *PARLE'89: Parallel Architectures and Languages Europe*, Vol. I, pp. 1-18, June 1989.
- [Hennessy 84] J. L. Hennessy, "VLSI Processor Architectures". *IEEE Transactions on Computers*, Vol. C-33, No. 12, pp. 1221-1246, December 1984.
- [Hill 89] M. D. Hill, A. J. Smith, "Evaluating Associativity in CPU Caches", *IEEE Transactions on Computers*, Vol. C-38, No. 12, pp. 1612-1630, December 1989.
- [Hoare 78] C. A. R. Hoare, "Communicating Sequential Processes", *CACM*, Vol. 21, No. 8, pp. 666-677, August 1978.
- [Hwang 93] K. Hwang, *Advanced Computer Architecture with Parallel Programming*. Preliminary Edition, 1993 by McGraw-Hill, Inc..
- [James 90] D. V. James, A. T. Laundrie, S. Gjessing, G. Sohi, "Scalable Coherent Interface", *IEEE Computer*, Vol. 23, No. 6, pp. 74-77, June 1990.
- [Karlin 81] S. Karlin, H. Taylor, *A Second Course in Stochastic Processes*. AP, 1981.

- [Katz 85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, R. G. Sheldon, "Implementing a Cache Consistency Protocol", *Proceedings of the 12th International Symposium on Computer Architecture*, pp. 276-283, June 1985.
- [Kindervater 91] G. A. P. Kindervater, *Exercises in Parallel Combinatorial Computing*, pp. 6-7, CWI Tracts, Stichting Mathematisch Centrum, Amsterdam, 1991.
- [King 72] W. F. King, III "Analysis of Demand Paging Algorithms" *Proceedings of IFIP congress 71*, Freiman Ed., Vol.1, pp. 485-490, North-Holland, 1972.
- [Kleiman 86] S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Usenix Summer 86*, pp. 238-247, 1986.
- [Kleinrock 75] L. Kleinrock, *Queueing Systems*, Vol. I: *Theory*. John Wiley & Sons, New York, 1975.
- [Kleinrock 76] L. Kleinrock, *Queueing Systems*, Vol. II: *Computer Applications*. John Wiley & Sons, New York, 1976.
- [Knuth 73] D. E. Knuth, *The Art of Computer Programming*, Vol. III: *Sorting and Searching*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.
- [Koldinger 91] E. J. Koldinger, S. J. Eggers, H. M. Levy. "On the Validity of Trace-Driven Simulation for Multiprocessors", *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 244-253, May 1991.
- [Lamport 78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *CACM*, Vol. 21, No. 7, pp. 558-565, July 1978.

- [Lamport 79] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocessor programs", *IEEE Transactions on Computers*, Vol. C-28, No. 9, pp. 690-691, September 1979.
- [Lee 87] R. L. Lee, P. C. Yew, D. H. Lawrie, "Multiprocessor Cache Design Considerations", *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 253-262, June 1987.
- [Leighton 92] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*. Morgan Kaufmann, 1992.
- [Lenoski 90a] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, M. Lam, "Desing of Scalable Shared-Memory Multiprocessors: The Dash Approach", *Proceedings of the Spring '90 IEEE COMPCON*, pp. 62-67, February 1990.
- [Lenoski 90b] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, "The Directory-Based Cache Coherence protocol for the DASH Multiprocessor", *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 148-159, May 1990.
- [Leung 83] C. H. C. Leung, "Analysis of Disk Fragmentation Using Markov Chains", *Computer Journal*, Vol. 26, No. 2, pp. 113-116, May 1983.
- [Levelt 90] W. G. Levelt, M. F. Kaashoek, H. E. Bal, A. S. Tanenbaum, "A Comparison of Two Paradigms for Distributed

Shared Memory”, Report IR-221, Vrije Universiteit, Amsterdam, August 1990.

- [Li 89] K. Li, P. Hudak, “Memory Coherence in Shared Virtual Memory Systems”, *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321-359, November 1989.
- [Maa 91] Y.-C. Maa, D. K. Pradhan, D. Thiebaut, “Two Economical Directory Schemes for Large-Scale Cache Coherent Multiprocessors”, *Computer Architecture News*, Vol. 19, No. 5, pp. 10-18, September 1991.
- [Maltese 86] G. Maltese, “A Simple Proof of the Fundamental Theorem of Finite Markov Chains”, *Am.Math.Monthly*, Vol. 93, No. 8, pp. 629-630, October 1986.
- [Mattson 70] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, “Hierarchical Storage Evaluation Techniques”, *IBM Systems Journal*, Vol. 17, No. 2, pp. 78-117, February 1970.
- [McCreight 84] E. McCreight, “The Dragon Computer System: An Early Overview”, Technical Report, Xerox Corporation, September 1984.
- [Mendelson 80] H. Mendelson, U. Yechiali, “A New Approach to the Analysis of Linear Probing Schemes”, *JACM*, Vol. 27, No. 3, pp. 474-483, July 1980.
- [Metcalf 76] R. M. Metcalfe, D. R. Boggs, “Ethernet: Distributed Packet Switching for Local Computer Networks”, *CACM*, Vol. 19, No. 7, pp. 395-404, July 1976.
- [Meyer 80] C. Meyer, “The Condition of a Finite Markov Chain and Perturbation Bounds for Limiting Probabilities”, *SIAM Journal*

- on Algebraic and Discrete Methods*, Vol. 1, No. 3, pp. 273-283, September 1980.
- [Muntz 74] R. R. Muntz, H. Opderbeck, "Stack Replacement Algorithms for Two-Level Directly Addressable Paged Memories", *SIAM Journal on Computing*, Vol. 3, No. 1, pp. 11-22, March 1974.
- [Muntz 93] R. R. Muntz, E. S. Silva, *Computational Solution Methods for Markov Chains: Application to Computer and Communication Systems*, In preparation.
- [Needham 78] R. M. Needham, M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers", *CACM*, Vol. 21, No. 12, pp. 993-999, December 1978.
- [Nelson 88] M. N. Nelson, B. B. Welch, J. K. Ousterhout, "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 134-154, February 1988.
- [Nielsen 91] M. J. Nielsen, "DECstation 5000 Model 200", *Proceedings of the Spring '91 IEEE COMPCON*, pp. 220-225, February 1991.
- [Ousterhout 87] J. K. Ousterhout, A. Cherenson, F. Douglass, M. N. Nelson, B. B. Welch, "The Sprite Network Operating System", Report No. UCB/CSD 87/359, June 1987.
- [Owicki 89] S. Owicki, A. Agarwal, "Evaluating the Performance of Software Cache Coherence", *Proceedings of the ASPLOS III*, pp. 230-242, April 1989.
- [Papamarcos 84] M. S. Papamarcos, J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 348-354, June 1984.

- [Quarterman 86] J. S. Quarterman, A. Silberschatz, J. L. Peterson, "4.2BSD and 4.3BSD as Examples of the Unix System", *ACM Computing Surveys*, Vol. 17, No. 4, pp. 379-418, December 1985.
- [Rao 78] G. S. Rao, "Performance Analysis of Cache Memories", *JACM*, Vol. 25, No. 3, pp. 378-395, July 1978.
- [Ritchie 74] D. M. Ritchie, K. Thompson, "The Unix Time-Sharing System", *Comm. of ACM*, Vol. 17, No. 7, pp. 365-375, July 1974.
- [Rivest 78] R. L. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *CACM*, Vol. 21, No. 2, pp. 120-126, February 1978.
- [Roberts 90] P. Roberts, T. Layman, G. Taylor, "An ECL RISC Microprocessor design for Two Level Cache", *Proceedings of the Spring '90 IEEE COMPCON*, pp. 228-231, February 1990.
- [Rudolph 84] L. Rudolph, Z. Segall, "Dynamic Decentralized Cache Consistency Schemes for MIMD Parallel Processors", *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 340-347, June 1984.
- [Ruschitzka 83] M. Ruschitzka, "A Markov Model for Evaluating Synchronization Algorithms", *Parallel and Large-scale Computers: performance, architecture, application*, IMACS Trans.Sci.Comp., Vol. 2, pp. 53-66, North-Holland, 1983.
- [Sandberg 85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem", *USENIX Summer 85*, pp. 119-130, 1985.



- [Schwartz 87] M. Schwartz, *Telecommunication networks: Protocols, Modeling and Analysis*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.
- [Seitz 84] C. L. Seitz, "Concurrent VLSI Architectures", *IEEE Transactions on Computers*, Vol. C-33, No. 12, pp. 1247-1265, December 1984.
- [Seitz 85] C. L. Seitz, "The Cosmic Cube", *CACM*, Vol. 28, No. 1, pp. 22-33, January 1985.
- [Shoemaker 90] K. Shoemaker, "The i486<sup>TM</sup> Microprocessor Integrated Cache and Bus Interface", *Proceedings of the Spring '90 IEEE COMPCON*, pp. 248-253, February 1990.
- [Silberman 83] G. M. Silberman, Stack Processing Techniques in Delayed-Staging Storage Hierarchies, *Communications of ACM*, Vol. 26, No. 11, pp. 999-1007, Nov. 1983.
- [Singh 91] J. P. Singh, W. D. Weber, A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory". CSL-TR-91-469, Stanford University, April 1991.
- [Slater 92] *A Guide to RISC Microprocessors*, M. Slater, Editor, Academic Press, Inc. 1992.
- [Sleator 85a] D. D. Sleator, R. E. Tarjan, "Self-Adjusting Binary Search Trees", *JACM*, Vol. 32, No. 3, pp. 652-686. July 1985.
- [Sleator 85b] D. D. Sleator, R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules", *CACM*, Vol. 28, No. 2, pp. 202-208, February 1985.

- [Slutz 72a] D. R. Slutz, I. L. Traiger, "Determination of Hit Ratios for a Class of Staging Hierarchies", *IBM Res.Rep. RJ 1044*, May 1972.
- [Slutz 72b] D. R. Slutz, I. L. Traiger, "Evaluation Techniques for Cache Memory Hierarchies", *IBM Res.Rep. RJ 1045*, May 1972.
- [Smith 79] A. J. Smith, "Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write-Through", *JACM*, Vol. 26, No. 1, pp. 6-27, January 1979.
- [Smith 82] A. J. Smith, "Cache Memories", *ACM Computing Surveys*, Vol. 14, No.3, pp. 473-530, September 1982.
- [Smith 93] A. J. Smith, *private communication*, April 1993.
- [Smith 91] M. D. Smith, "Tracing With Pixie", CSL-TR-91-497, Stanford University, November 1991.
- [So 86] K. So, A. S. Bolmarcich, F. Darema-Rogers, V. A. Norton, "SPAN - A Speedup Analyzer for Parallel Programs", *IBM T.R. RC12186*, September 1986.
- [Spencer 85] J. Spencer, "Probabilistic Methods", *Graphs and Combinatorics*, Vol. 1, No. 4, pp. 357-382, 1985.
- [Spencer 87] J. Spencer, *Ten Lectures on the Probabilistic Methods*. CBMS-NFS Regional Conference Series of Applied Mathematics, No. 52, SIAM, 1987.
- [Stenström 90] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 12-25, June 1990.

- [Stewart 83] G. Stewart, "Computable Error Bounds for Aggregated Markov Chains", *JACM*, Vol. 30, No. 2, pp. 271-285, April 1983.
- [Stone 71] H. S. Stone, "Parallel Processing with the Perfect Shuffle", *IEEE Transactions on Computers*, Vol. C-20, No. 1, pp. 153-161, January 1971.
- [Taksar 91] M. Taksar, W. Grassman, "Probabilistic Approach to Computational Algorithms for Finding Stationary Distribution of Markov Chains", *J.Comp.Appl.Math.*, Vol. 36, No. 2, pp. 131-136, August 1991.
- [Tang 76] C. K. Tang, "Cache Design in the Tightly Coupled Multiprocessor System", *AFIPS Conf. Proc., National Computer Conf.*, pp. 749-753, June 1976.
- [Tarjan 78] R. E. Tarjan, "Complexity of Combinatorial Algorithms", *SIAM Review*, Vol. 20, No. 3, pp. 457-491, July 1978.
- [Tarjan 83] R. E. Tarjan, *Data Structures and Network Algorithms*. CBMS-NFS Regional Conference Series of Applied Mathematics, No. 44, SIAM, 1983.
- [Teller 89] P. J. Teller, "The TLB Consistency Problem", Technical Report RC 15156, T. J. Watson Research Center, November 1989.
- [Thapar 90] M. Thapar, B. Delagi, "Stanford Distributed-Directory Protocol", *IEEE Computer*, Vol. 23, No. 6, pp. 78-80, June 1990.
- [Thompson 87] J. G. Thompson, "Efficient Analysis of Caching Systems", TR UCB/CSD 87/374, Ph.D. dissertation, Univ. of California, Berkeley, Oct. 1987.

- [Thompson 89] J. G. Thompson, A. J. Smith, "Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories", *ACM Transactions on Computer Systems*, Vol. 7, No. 1, pp. 78-116, February 1989.
- [Traiger 71] I. L. Traiger, D. R. Slutz, "One-Pass Techniques for the Evaluation of Memory Hierarchies", *IBM Res.Rep.RJ 892*, San Jose, California, July 1971.
- [Tzelnic 82] P. Tzelnic, "The Length of Path for Finite Markov Chains and its Application to Modeling Program Behavior and Interleaved Memory Systems", *Applied Prob.-CS: the Interface*, R. Disney, T. Otto (Ed), Vol. 2, pp. 375-403, Birkhauser Boston Inc., 1982.
- [Wallach 92] D. A. Wallach, "PHD: A Hierarchical Cache Coherent Protocol", Master Thesis, EECS, MIT, 1992.
- [Wang 89] W. Wang, "Multilevel Cache Hierarchies", DCS TR 89-09-13, Ph.D Dissertation, Univ. of Washington, Seattle, September 1989.
- [Wang 91] W. Wang, J.-L. Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis", *ACM Transactions on Computer Systems*, Vol. 9, No. 3, pp. 222-241, September 1991.
- [Weber 89] W.-D. Weber, A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors", *Proceedings of the ASPLOS III*, pp. 243-256, April 1989.
- [Wilson 87] A. W. Wilson, Jr. "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors", *Proceedings of the*

*14th International Symposium on Computer Architecture*, pp. 244-252, June 1987.

- [Wu 92] Y. Wu, J. Popek, R. R. Muntz, "Efficient Evaluation of Arbitrary Set-Associative Caches on Multiprocessors", *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pp. 507-514, December 1992.
- [Wyk 88] C. J. Van Wyk, *Data Structures and C Programs*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
- [Yang 90] Q. Yang, G. Thangadurai, L. N. Bhuyan, "An Adaptive Cache Coherence Scheme for Hierarchical Shared-Memory Multiprocessors", *Proceedings of the 2th IEEE Symposium on Parallel and Distributed Processing*, pp. 318-325, December 1990.
- [Yew 87] P. C. Yew, N. F. Tzeng, D. H. Lawrie, "Distributed Hot-Spot Addressing in Large-Scale Multiprocessors", *IEEE Transactions on Computers*, Vol. C-36, No. 12, pp. 388-395, April 1987.