CACHE DESIGN SIMULATION IN MULTIPROCESSOR BASED ENVIRONMENT

By
Captain Tahir Javaid
NC Behram Khan
Captain Tasadduq Mahmood
Captain Ahmar Raza

Supervisor: Brigadier S.M. Salahuddin Tariq

Project report for partial fulfilment of the requirement of MCS/NUST
for the
award of the B.E degree in Software Engineering

Department of Computer Science
Military College of Signals
Rawalpindi

April 2003

DECLARATION

"No portion of the work presented in this dissertation has been submitted in support of another award or qualification either at this institution or elsewhere"

Acknowledgements

We are thankful to Almighty Allah who enabled us to complete this project. We show gratitude to our parents, whose love and care has enabled us to be what we are. Our deepest appreciation is extended to all whose unfeigned help and encouragement made the present work a reality.

We gratefully acknowledge the help and guidance provided by our project advisors Brigadier S.M.Salahuddin Tariq and Mr. M. Mohsin Rahmatullah. Without their personal supervision, advice and valuable guidance, completion of this project would have been doubtful. We are deeply indebted to them for their encouragement and continual help during this work.

Our very special thanks are extended to Prof. Dr. Mark D Hill and Ms. Carrie Pritchard at University of Wisconsin-Madison, and Ms. Weikel at Virginia State University for their very useful help extended during the project which made us more clear about whole project.

We would like to express our gratitude to all faculty members of the Department of Computer Science for their cooperation and healthy academic environment throughout our career at Military College of Signals Rawalpindi.

Abstract

The gap between processor and memory speeds is increasing day by day. This situation makes it imperative to use effective caches or other structures between CPUs and DRAMs. The traditional measures of the quality of a caching strategy have been the aggregate hit rate and the execution time of a benchmark, but these measures are no longer sufficient. They provide no insight into dynamic programme behaviour and little guidance in designing a multi-level memory hierarchy. Current caches are designed primarily using ad hoc experimentation and commonly accepted rules-of-thumb: there is no systematic experimental methodology, and there is only fragmented theory to guide the design.

This demands evolution of new methods for evaluating memory system designs before they are implemented in hardware. One such method, trace-driven memory simulation, has been the subject of intense interest among researchers and has, as a result, enjoyed rapid development and substantial improvements during the past decade.

This project report surveys and analyzes these developments by establishing criteria for evaluating Cache Designs for Multiprocessor based Environments using trace-driven simulation method, and then applies these criteria to describe, categorize and recommend optimal combinations of Cache Designs basing on user discretion.

Besides, it provides an analysis methodology that supports cache hierarchy design theoretically and subsequently leads towards the development of mathematical and software tools that accompany this methodology.

In doing so it discusses the strengths and weaknesses of different approaches and uses One Pass Stack based Trace Driven Simulation method for application of Cache Coherence Protocols, to recommend best, appropriate and user defined design, when all criteria, including accuracy, speed, memory, flexibility, portability, ease-of-use and expense are considered.

Contents

List of Figures

Chapter 2

Chapter 3

Chapter 4

Chapter One

Introduction

1.1 Cache and System Performance in Retrospect.

Cache: a safe place for hiding or storing things.

> Webster's New World Dictionary of the American Language, Second College Edition (1976)

Cache is the name generally given to the first level of the memory hierarchy encountered once the address leaves the CPU. Since the principle of locality applies at many levels, and taking advantage of locality to improve performance is so popular, the term cache is now applied whenever buffering is employed to reuse commonly occurring items: examples include file caches, name caches, and so on. The memory hierarchy is given the responsibility of address checking: hence protection schemes for scrutinizing addresses are also part of the memory hierarchy.

The importance of the memory hierarchy has increased with advances in performance of processors. For example, in 1980 microprocessors were often designed without caches, while in 1995 they often came with two levels of caches. Microprocessor performance improved 55% per year since 1987, and 35% per year until 1986.

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a memory hierarchy, which takes advantage of locality and cost/performance of memory technologies. The principle of locality says that most programs do not access all code or data uniformly. This principle, plus the guideline that smaller hardware is faster, led to the hierarchy based on memories of different speeds and sizes. Since fast memory is expensive, a memory hierarchy is organized into several levels each smaller, faster and more expensive per byte than the next level. The goal is to provide a memory system with cost almost as low as the cheapest level of memory and speed almost as fast as the fastest level. The levels of the hierarchy usually subset one another: all data in one level is also found in the level below, and all data in that lower level is found in the one below it, and so on until we reach the bottom of the hierarchy. Note that each level maps addresses from a larger memory to a smaller but faster memory higher in the hierarchy. As part of address mapping, where miss rate is the fraction of accesses that are not in the cache and miss penalty is the additional clock cycles to service the miss, a block is the minimum unit of information that can be present in the cache (hit in the cache) or not (miss in the cache).

1.2 **Trace Driven Simulation Methods for Cache Performance.**

Caches have been widely used in most computer systems for the last two decades. They are expected to play an increasingly important role in future high-performance computer systems. Very soon we will see machines with an off-chip cache miss penalty of over 100 instruction times (undoubtedly, the performance of these machines will be substantially influenced by their cache performance). A careful cache design choice is therefore crucial to the design of future computer systems. The most accurate way of assessing cache performance before a machine is built would be a thorough simulation of the whole system, which captures not only the detailed behaviour of the cache but also its subtle interactions with the rest of the system. Unfortunately, a thorough simulation of a complete system generally takes too long to allow coverage of the vast design space.

A more realistic approach would loosely couple the cache with the rest of the system via high-level analytical model. In this approach, detailed simulations are performed on the cache subsystem to produce performance metrics such as miss ratios and write-back traffic. These results are then incorporated into the high-level model to generate the system performance under different cache design choices.

Still, even cache simulation itself is not an easy task. The widely used trace-driven cache simulation technique generally requires a large amount of disk space to store the program traces, and simulations that produce results which would cover a sufficient portion of the design space are very time consuming. Therefore, methods which can quickly and cheaply produce cache performance results are desirable.

A large number of techniques for trace-driven cache simulation have been reported in the literature. Related work is briefly reviewed in following section along with a description of trace characteristics that we use throughout this paper.

1.3 **Background and Related Work in Trace Driven Simulations.**

Trace driven simulations can be further subdivided under three major headings, which are:

- Trace Collection
- Trace Reduction
- Trace Processing

Out of these three each one is a complete subject within itself. Though the main focus of our project is on *Trace Processing* however following sections of this chapter briefly discuss ‘trace collection’ and ‘trace reduction’ for the sake of completeness, whereas a detailed discussion on trace processing is generated in chapter 2.

1.3.1 **Trace Collection**

To ensure accurate simulations, collected address traces should be as close as possible to the actual stream of memory references made

by a workload when running on a real system. Trace quality can be evaluated based on the *completeness* and *detail* in a trace, or on the degree of *distortion* that it contains. Ideally speaking a *complete* trace should include all memory references made by each component of the system, including all user-level processes and the operating system kernel. User level processes should include not only applications, but also OS server and daemon processes that provide services such as a file system or network access. Complete traces should also include dynamically-compiled or dynamically-linked code, which is becoming increasingly important in applications such as processor or operating-system emulation. An ideal *detailed* trace is one that is annotated with information beyond simple raw addresses. Useful annotations include changes in VM page-table state for translating between physical and virtual addresses and tags that mark each address with a reference type (read, write, execute), size (word, half word, byte) and a timestamp. Traces should be *undistorted* so that they do not include any additional memory references, or references that appear out of order relative to the actual reference stream of the workload had it not been monitored. Common forms of distortion include *trace discontinuities*, which occurs when tracing must stop because a trace buffer is not large enough to continue recording workload memory references, and *time dilation* and *memory dilation*, which occur when the tracing method causes a monitored workload to run slower, or to consume more memory than it normally would.

In addition to the three aspects of trace quality described above, a good trace collector exhibits other characteristics as well. In particular, *portability*, both in moving to other machines of the same type and to machines that are architecturally different is important. Finally, an ideal trace collector should be *fast*, *inexpensive* and *easy to operate*.

Address traces have been extracted at virtually every system level, from the circuit and microcode levels to the compiler and operating-system levels. (see Figure 2). We organize the following discussion accordingly, starting at the lower hardware levels.

1.3.1.1 *External Hardware Probes*

A straightforward method for collecting address traces is to record signals from electrical probes physically connected to the address bus of a host computer while it runs a workload. The address and control signals are fed into an external memory buffer at the full speed of the monitored host system, and when the buffer fills, its contents are transferred to a standard storage device, such as tape or disk, so that it can be processed at a later time. If a long, continuous address trace is desired, then the buffer must either be very large or there must be some way to stall the host whenever the buffer becomes full. It is usually only possible to stall the processor — external I/O devices, such as

disks or network controllers will must usually be permitted to continue operating. If there is no way to stall the system, then several discontinuous address-trace samples can be acquired and concatenated together. In either case, the resulting trace exhibits a form of distortion that is called *trace discontinuity*.

The main advantage of the probe-based trace collectors is their ability to capture trace sequences complete with both user and kernel memory references, and free of most forms of trace distortion, provided that the trace buffer is deep enough. Although the traces are complete, this does not necessarily mean that they are easy to interpret. Hardware events such as cache misses, integer- and floating-point-unit stalls, exceptions and interrupts all must be separated from run cycles to determine the actual type (read, write, execute) and size (word, half word, byte) of the memory references made by a monitored processor. In processors that implement hardware pre-fetching or speculative execution, it may be difficult or impossible to separate "true" memory references from those that occur due to a pre-fetch that might not actually be used. Some of these problems can be overcome by implementing the inverse function of the processor sequencer, either in the trace-collecting hardware, or in a trace post-processing tool. Because the addresses captured by a probe-based monitor are usually physical addresses, special methods that may require cooperation from the host OS must be used to reverse-translate addresses to their matching virtual addresses. These problems all follow from the fact that probe-based trace collectors are external to the monitored system and therefore do not have easy access to operating-system data structures.

A common misconception regarding trace collection using hardware probes is that the technique is very fast. While it is true that acquisition of the trace proceeds at the full speed of the monitored system, it is important to account for the overhead of managing trace-buffer overflow as well as the time required to empty the buffer. This overhead is typically not reported in published papers, but because most systems can unload these buffers only through some form of relatively low-bandwidth channel, this overhead is necessarily high. For a system where overhead data is available, approximately 12 hours are required to obtain 11 seconds of real-time system activity.

Although trace collection with hardware probes is time consuming, once the traces have been captured and stored to a permanent file they require no special hardware to use, and can be used repeatedly to achieve reproducible simulation results. Hardware probe-based methods share other common disadvantages. The first is expense. Logic analyzers with deep

trace memories cost from $50,000 to $200,000 .These amounts are probably low compared to the engineering costs associated with designing custom hardware. A second problem is portability. Although logic analyzers support probes for most popular microprocessors, it is often necessary to physically modify the motherboard or chassis of the monitored system to enable probe access to the signals of interest. These systems also require an understanding of the electrical issues concerning the connection of probes to running hardware, and are therefore typically fragile, sensitive to their operating environment, and difficult to learn and operate.

As noted above, the advent of on-chip caches is making it increasingly difficult to build trace collection hardware as an afterthought. The future of probe-based trace collection therefore depends mainly on the level of support *designed* into systems for this task. A small, on-chip trace buffer that traps to the operating-system kernel whenever it becomes full is an example of the sort of support that could be provided. However, even a very small buffer of 2048 entries with 32-bits per entry (8 K-bytes) is about the size of on-chip caches in current microprocessors and thus would be relatively costly in terms of chip area. An alternative approach would be to send certain key internal signals through the microprocessor package pins so that they can be monitored externally.

1.3.1.2 ***Microcode Modification***

The high cost of circuit-level probing has motivated many researchers to develop methods for collecting traces at higher levels of system abstraction. One such alternative is to collect traces at the borderline between the hardware and software levels of a system in microcode. From the beginnings of the IBM 360 series (1964) until the DEC VAX machines, the most common method for implementing control logic was microcode. When implemented off-chip, a microcode memory was often writable or could be modified through replacement, making it possible to change the behaviour of instructions, or to support multiple instruction sets. Agarwal (1986, 1988) realized that this mechanism made it possible to collect address traces. He modified the microcode on a VAX 8200 to cause all instructions to deposit the addresses of their memory references into a reserved area of main memory as a side effect of their execution. This method, which Agarwal called *address tracing using microcode (ATUM)*, offers a number of advantages. The first is completeness. Because the microcode runs beneath the operating system, all user and kernel references are captured, as well as those from dynamically-compiled and dynamically-linked code. Because ATUM has access to internal system state,

it is easily able to annotate traces with access-type tags and page-map information. Another advantage is speed. ATUM acquires address traces with a slowdown of only about 10 to 20, and because the addresses can be processed directly out of the trace buffer in main memory, there is no overhead of buffer unloading as with external probe-based trace collection. Finally, no additional hardware is required. The only cost associated with ATUM is the engineering effort required to modify microcode to produce the desired results. The ATUM method suffers a few minor disadvantages and one major one. First, ATUM traces exhibit some discontinuity distortion because the processor is not stalled when the trace buffer becomes full. Buffer size could be increased only up to a certain point because it took away from the usable memory of the host system. Agarwal has developed a method, called *trace stitchin*g, to counter this problem. Microcode modification also introduces another form of trace distortion, commonly called *time dilation*. Because instructions take 10 to 20 times as long to execute as they normally would, external devices such as disks and network controllers appear to the workload to be faster than they actual are, and interrupts from the system clock occur more frequently, thus changing the workload's behaviour.

The primary disadvantage of the microcode-modification technique is that the technique is now effectively obsolete because most new microprocessors use hardwired control or have an on chip microcode memory that is not easily modified. The fundamental idea behind microcode modification — augmenting the interpretation of instructions to generate trace addresses as a side effect of their execution — can, however, be implemented at other levels in a system. This has been made easier by some of the very trends that have made microcode modification obsolete. Hardwired control, for example, has been made possible (or at least easier) with the advent of RISC instruction sets. The relatively simple and uniform coding of RISC instruction sets has also made it easier to develop fast instruction-set emulators and binary-rewriting tools for annotating executables to produce traces as a side effect of their normal execution.

1.3.1.3 ***Instruction-set Emulation***

An instruction-set architecture (ISA) is the collection of instructions that defines the interface between hardware and software for a particular computer system. A microcode engine, as described in the previous section, is an ISA interpreter that is implemented in hardware. It is also possible to interpret an instruction set in software through the use of an *instruction-set emulator*. Emulators typically execute one instruction set (the

target ISA) in terms of another instruction set (the *host* ISA) and are usually used to enable software development for a machine that has not yet been built, or to ease the transition from an older ISA to a newer one. As with microcode, an instruction-set emulator can be modified to cause an emulated program to generate address traces as a side-effect of its execution.

Conventional wisdom holds that instruction-set emulation is very inefficient, with slowdowns estimated to be in the range of 1,000 to 10,000 Agarwal (1989); Wall (1989); Borg(1989); Stunkel(1991); Flanagan(1992). The degree of slowdown is clearly related to the level of emulation detail. For some applications, such as the verification of a processor's logic design, the simulation detail required is very high and the corresponding slowdowns may agree with those cited above. In the context of this review, however, we consider an instruction-set emulator to be sufficiently detailed for the purposes of address-trace collection if it can produce an accessible trace of memory references made by the instructions that it emulates. Given this minimal requirement, there are several recent examples of instruction-set emulators that have achieved slowdowns much lower than 1,000; they work by fetching, decoding and then dispatching instructions one at a time in an iterative emulation loop, re-interpreting instructions each time they are encountered. Instructions are fetched by reading the contents of the emulated program's text segment, and are decoded through a series of mask and shift operations to extract the various fields of the instruction (opcode, register specifiers, etc.). Once an instruction has been decoded, it is emulated (*dispatched*) by updating machine state, such as the emulated register set, which can be stored in memory as a *virtual register* data structure , or which may be held in the actual hardware registers of the host machine . An iterative interpreter may use some special features of the host machine to speed instruction dispatch, but this final step is more commonly preformed by simply jumping to a small subroutine or *handler* that updates machine state as dictated by the instruction's semantics. The reported slowdowns for iterative emulators range from 20 to about 600, but these figures should be interpreted carefully because larger slowdowns may represent the time required to emulate processor activity that is not strictly required to generate address traces.

Some interpreters avoid the cost of repeatedly decoding instructions by saving *predecoded* instructions in a special table or cache. A predecoded instruction typically includes a pointer to the handler for the instruction, as well as pointers to the memory locations that represent the registers on which the

instruction operates. The register pointers save both decoding time as well as time in the instruction handler, because fewer instructions are required to compute the memory address of a virtual register.

1.3.1.4 ***Static Code Annotation***

The fastest instruction-set emulators *dynamically* translate instructions in the target ISA to instructions in the host ISA, and optionally annotate the host code to produce address traces. Because these emulators perform translation at run time they gain some additional functionality, such as the ability to trace dynamically-linked or dynamically-compiled code. This additional flexibility comes at some cost, both in overall execution slowdown and in memory usage. For the purposes of trace collection, it is often acceptable to trade some flexibility for increased speed. If the target and host ISAs are the same and if dynamically-changing code is not of interest, then a workload can be annotated *statically*, before run time. With this technique, instructions are inserted around memory operations in a workload to create a new executable file that deposits a stream of memory references into a trace buffer as the workload executes. Static code annotation can be performed at the source (assembly) level, the object-module level, or the executable (binary) level, with different consequences for both the implementation and the end user Stunke(l991); Wall(1992); Pierce(1994).

.

The main advantage of annotating code at the source level is ease of implementation. At this level, the task of relocating the code and data of the annotated program can be handled by the usual assembly and link phases of a compiler, and more detailed information about program structure can be used to optimize code-annotation points. Unfortunately, annotation at this level may render the tool unusable in many situations because the complete source code for a workload of interest is often not available and annotation at this level is also the most difficult to implement because executable files are often stripped of symbol-table information. A significant amount of analysis may be required to properly relocate code and data after trace-generating instructions have been added to the program Pierce(1994). Despite these difficulties, there exist several program-annotation tools that operate at the executable level. A common problem with many code annotators is that they produce traces with an inflexible level of detail, requiring a user to select the monitoring of either data or instruction references (or both) with an all-or-nothing switch. Many tools are similarly rigid in the mechanism that they use to communicate addresses, typically forcing the trace through a file

or pipe interface to another process containing the trace processor. Some more recent tools overcome these limitations and offer a flexible interface that enables a user to specify how to annotate each individual instruction, basic block and procedure of an executable file; at each possible annotation point the user can specify the machine state to extract, such as register values or addresses, as well as an analysis routine to process the extracted data. If no annotation is desired at a given location, they do not add it, thus enabling a minimal degree of annotation to be specified for a given application. For I-cache simulation, for example, a simulator writer can specify that only instruction references be annotated, and that a specific I-cache analysis routine be called at these points. In general, code annotators are not capable of monitoring multi-process workloads or the operating system kernel, but there exist some exceptions as well.

1.3.1.5 ***Single-step Execution***

The highest level of system abstraction for collecting address traces is the operating system. Most operating systems support some form of debugging utility that enables a programmer to step through a program one instruction at a time to expose errors. This form of debugging is usually supported in hardware through a single-step execution mode, where the processor traps into the OS kernel after the execution of each instruction or basic block Digital 86; AMD91; AMD93; Motorola93; HP90; Motorola90 or by breakpoint instructions that cause kernel traps whenever they are executed Kane(1992); Intel(1990). A debugger that supports single-step execution and examination of processor state, such as registers, can be modified to generate both instruction-address and data-address traces. Instruction-address traces are produced by simply recording the value of the program counter at each execution step. Data-address traces require instruction emulation to determine if the current instruction generates a memory reference and, if so, the value of that reference.

The main advantages of this method are low expense, high portability, and ease of use. With the exception of debugger data structures, little additional host memory is used. Unfortunately, slowdowns for this technique are high, with estimates varying widely from 100 Agarwal (1988) to 1,000 Flanagan (1992) to 10,000 (Holliday1991). High slowdowns are usually due to debugger implementations that rely on the UNIX `ptrace ()` facility which, in turn, is implemented using UNIX exception-signal handlers. Although there is nothing inherent in this approach that limits traces to a single process, or to user-only references, debuggers typically do impose these

limitations. Similarly, dynamically compiled and dynamically-linked code is usually not supported by debuggers. Because only address-trace information is desired, a single-step trace-collection tool could, in principle, be written from scratch to avoid the overheads and single-process limitations of program debuggers. We are not aware of any existing trace-collection system that uses this approach.

1.3.2 **Trace Reductions.**

Since the space and time needed for trace-driven cache simulation are approximately proportional to the trace length, several early studies have focused on reducing the trace length to reduce the cost of cache simulation. Smith (1977) pioneered this work by proposing a trace deletion technique for memory-paging studies. He used Least Recently Used (LRU)-model of memory references and produced a reduced trace by deleting references that accessed the top D levels of the LRU stack of data. The resulting trace, if used for simulating memory larger than D pages under the LRU replacement algorithm, would produce almost the same number of misses (page faults) as the original long trace, provided that the page size is kept constant.

An extension of Smith's (1977) technique was proposed by Puzak (1985). He called it trace stripping. This approach focuses on reducing traces for simulating set-associative caches. A direct-mapped cache (serving as a filter) is simulated and the miss references are recorded; they form a reduced trace. This reduced trace, if used to simulate caches with a larger number of sets, would result in the same number of misses as the original trace, provided that the block size is kept the same.

Other approaches have also been proposed which substantially reduce the trace length but cannot guarantee that exact performance metrics would be obtained by using the reduced trace. For example, Smith's(1977)'s snapshot method records memory references at regular time intervals; a set selection method described by Puzak(1985) keeps only references that access some specific sets; Agarwal's(1987) trace compaction combines Puzak's(1985) reduction technique and Smith's(1977)'s snapshot method ; Laha et al.(1988), proposed another sampling-based method.

The main focus is to produce exact cache performance cheaply and quickly. To this end, sampling approaches cannot be used. Even Puzak's (1985) and Smith's (1977)'s original exact reduction techniques are not sufficient for two reasons. First, the reduced trace method can only produce a count of misses (i.e., hit ratios) but not the number of write-backs. Second, the existing methods apply only to

uniprocessor caches and are inadequate for multiprocessor caches. Finally, the reduced trace cannot be used to simulate caches which have block sizes different from that of the cache filter used for the trace reduction.

1.3.2.1 *Trace Reduction for Write-Back Cache Simulation.*

As stated earlier, the objective of trace reduction is to produce a reduced trace which can be used to generate exactly the same number of misses and write-backs as the original trace. Wang(1991) used small direct-mapped cache as a filter but instead of keeping only references that cause misses, he also kept those that were first-time writes.

The intuitive idea behind this reduction is that if a reference causes a hit in a small direct-mapped cache, it will also hit in a larger cache. Furthermore, if a block is dirty in a small direct-mapped cache it will also be dirty in a larger cache. Thus, the miss references to a small cache would be a superset of misses to larger caches. Also, references that cause blocks to be dirty in a small cache would be a superset of references that make blocks dirty in larger caches. Therefore, the reduced trace can be used to produce exact cache performance metrics for same size or larger caches (which implies some criteria for the parameters of the small cache).

Formal description of reduction is as follows.

Assume all caches under study have a power-of-two integer number of sets, simulate a direct-mapped cache with S sets using a program trace and record only references that cause misses or writes on clean data to form a reduced trace. This reduced trace, if used to simulate caches with a larger number of sets, would produce exactly the same number of misses and write-backs as the original trace, provided that the same block size is used.

Now we see the proof of the above statement,
Proof. The proof on the exact number of misses was given by Puzak (1985). So, it is sufficient to prove that using the reduced trace would produce the same number of write-backs as the original trace. Since the number of misses is exactly the same, any block that is replaced is the same independently of whether the reduced trace or the original trace is used. For the purpose of proof by contradiction, let us assume that there is a block, when replaced using the reduced trace that has a dirty state different from the original trace. That is, we have two cases:

(1) Case 1: the block is clean using the reduced trace, but is dirty using the original trace. The reasons for a block to be dirty are either the block is brought into the cache upon a write miss or upon a read miss followed by a write-on-clean. In the case of a write miss, there will be a write miss in the filter cache too, and the reduced trace will retain the reference. That is, using the reduced trace will make the block dirty too. In the case of a write-on-clean, it can be either a write-on-clean or a write miss on the filter cache. In either case, the write reference is recorded in the reduced trace. Thus, using the reduced trace will make the block dirty as well.

(2) Case 2: the block is dirty using the reduced trace, but is clean using the original trace. This case is easily dismissed as follows. Since the block is clean using the original trace, the block must have been brought into the cache on a read miss and possibly followed by a number of read references. The first read miss and possibly a subset of the followed read references will be recorded in the reduced trace. None of them will cause the block to be dirty, since they are read references. Thus, using the reduced trace will make the block clean as well. For both cases, the block will be in the same dirty state using either the original trace or the reduced trace. This contradicts the assumption that the block be in different states. Since a block being replaced cannot have different dirty states and since the number of replacements is the same, using the reduced trace will produce the same number of write-backs as the original trace.

Wang (1991) used the traces mentioned above to measure the effectiveness of trace reduction. By using an 8K-byte cache filter with a 4-byte block size obtained reduced traces which are between 10 and 22 times shorter than the original traces.

The reduced traces were used to simulate (i.e., record misses and number of write-backs) a 32 K-byte 4-way set-associative cache. Table I shows that the simulation times are between 7 and 15 times faster using the reduced traces. We notice from Table I that the speedups are not as good as the space reduction ratios. This is due to the fact that the reduced traces have fewer localities than the original traces.

1.3.2.2 ***Multiblock-Size Trace Reduction.***

The reduction techniques we have discussed so far only work if reduced traces are used to simulate caches with the same block size as that of the filter cache used for reduction. A reduced trace cannot be used to simulate a large cache with a different block size because the contents of a larger cache are

not necessarily a superset of those of a smaller cache with different block sizes. Thus, the reduced trace may not capture every miss that can occur in a larger cache. In other words, the reduced trace cannot be used to simulate larger caches and still produce the exact results.

What this lack of consideration implies is that we need to produce reduced traces for every possible block size under study or we take the risk of getting incorrect results. Unfortunately, the disk spaced needed to store every reduced trace would take away the space-saving benefits of the trace reduction. A "universal" reduced trace that can be used for all different block sizes but does not take up too much space is therefore highly desirable.

Table I Time and Space Saving Due to Trace Reduction

traces	Pero	Thor	Pops
original length	14,198kB	16,415kB	16,087kB
reduced trace length	624kB	1,045kB	1,519kB
space reduction	22.7	15.7	10.6
simu. time for orig. trace	208.7 sec	246.6 sec	268.3 sec
simu. time for redu. trace	14.4 sec	26.8 sec	39.6 sec
speed-up	14.5	9.2	6.8

In an attempt to produce a universal reduced trace, we observe that most misses to a cache filter with one block size tend to be misses in another filter with a different block size. Therefore, Wang(1991) produce universal reduced traces by collecting the superset of misses that occur on every cache filter with different block sizes. His results (Table II) show that with 40-48% of additional space, i.e., 1.8-4.5% of the original traces, we can have universal reduced traces for 5 block sizes. We notice from TableII that there is a jump on the length of the reduced trace from 4 block sizes to 5 block sizes for the Pops trace. The reason this happens is that the Pops trace has the worst locality among the three traces and the 8 K-byte cache filter experiences lots of misses due to thrashing when the block size is 64 bytes.

1.3.2.3 ***Trace Reductions for Parallel Multiprocessor Cache Simulation.***

Recently, there have been many performance (trace-driven simulation) studies on cache coherence protocols for shared-bus multiprocessors. In this section we describe how multiprocessor traces can be reduced and still be used to provide exact performance figures. The performance metrics of interest for multiprocessor caches are miss ratios, number of write-backs and cache-coherence interferences from other caches. We start our discussion with a straightforward sequential simulation and describe how traces can be reduced for this type of simulation.

We then consider extensions of the reduction technique to parallel cache simulations.

A few assumptions are in order before we present the reduction technique, these assumptions were originally made by Wang(1991) and we apply them here for our discussion. We assume that each processor has a private cache and that the cache size is the same for every cache in the system. We also assume that the relative order of reference streams from each processor is kept the same across different simulations; similar assumptions are presented by Thompson (1989) and Lin et al. (1989). To simplify our discussions we further assume that an invalidation protocol Sweazey (1986) is used for cache coherence, although our results can easily be extended to a larger class of coherence protocols. For the MOESI class of invalidation protocols, each cache block can be in one of four states: invalid, private, shared and dirty. On a read miss, the block is brought in and the state is set to private if the other caches do not have this block; otherwise the state is set to shared. On a write miss the block is brought in and any copy of this block, if present in other caches, is invalidated before the write is done. The state is set to dirty. On a write hit on a shared block, other copies also get invalidated before the write is done and the state is set to dirty. A write hit on a private block does not require a bus transaction except that the state of the block is changed to dirty.

Table II Length of Reduced Traces with up to 5 Different Block Sizes

traces	Pero	Thor	Pops
original length	14,198kB	16,415kB	16,087kB
redu. length (4 byte block size)	624k (4.4%)	1,045k (6.4%)	1,519k (9.4%)
redu. length (4b and 8b)	731k (4.9%)	1,186k (7.2%)	1,643k (10.2%)
redu. length (4b, 8b and 16b)	770k (5.4%)	1,270k (7.7%)	1,719k (10.7%)
redu. length (4b, 8b, 16b and 32b)	828k (5.8%)	1,353k (8.2%)	1,795k (11.2%)
redu. length (4b, 8b, 16b, 32b and 64b)	878k (6.2%)	1,453k (8.9%)	2,247k (13.9%)

A straightforward simulation method represents a cache as a table and takes a serialized reference stream as input. For each reference a table look-up is performed to determine whether there is a hit and whether any coherence action needs to be done. Although this straightforward serial method is slow, we use it as the basis for comparing the speed-up with parallel methods to be discussed later.

The trace reduction for this method works as follows. Simulate small caches (serving as filters) under the chosen coherence protocol using a multiprocessor trace and record only references that cause misses or result in writes on clean blocks. These references form a reduced trace and, if used for simulating large multiprocessor caches, would produce the same

number of misses, write-backs, and invalidations as the original trace, provided that the block size is kept the same and the same coherence protocol is used.

The proof of the above statement is similar to that of the uniprocessor case, except we need to prove that the reduced traces preserve the references that cause invalidations i.e. if the original trace is used to simulate a larger cache (more sets) it should produce the same number of invalidations as the reduced trace. To show this, let us assume, for the purpose of a proof by contradiction, that there are invalidations in the original trace but not in the reduced trace. Then, there must be a first such invalidation. According to our chosen protocol (to mention again this protocol was originally designed by Wang (1991) and we are only discussing it here), this invalidation can either be caused by a write miss or a write hit on shared data (i.e., a subset of write on clean) in the large cache. If this is a write miss, then there will also be a write miss in the filter cache, so this reference will be recorded in the reduced trace. If this is a write hit on shared data, then it can only fall into the following three categories in the filter cache: namely, a write hit on shared data, a write hit on private, or a write miss. It cannot be a write hit on a modified block in the filter cache, since before this first mismatched invalidation a block that is modified in the filter cache will also be modified in the larger cache. That is, the reference causes either a write on clean or a write miss in the filter cache, and it will be recorded in the reduced trace as well. This is a contradiction to the assumption that the reduced trace does not contain the reference that causes the invalidation. Therefore, the number of invalidations will still be the same for the reduced trace and for the original trace.

Table III Time and Space Saving due to Trace Reduction on Simulation of a 4-Processor System

traces	Pero	Thor	Pops
original length	14.198k	16.415k	16.087k
reduced trace length	641k	1.117k	1,718k
space reduction	22 1	14 7	9 4
simu. time for orig. trace	95.4 sec	115 9 sec	118.9 sec
simu. time for redu. trace	5 6 sec	10.5 sec	15 8 sec
speed-up	17	11	7 5

Using the above reduction method Wang (1991) produced reduced traces for the three multiprocessor traces of Table IV. These traces are used to simulate 4-cpu multiprocessor caches, 128 K-byte, direct-mapped with a 4 byte block size. Table III gives the space and time comparison in using long traces vs. reduced traces. It shows that the reduced traces are between 9 to 22 times shorter than the original traces and the simulation times using the reduced traces are between 7 to 17 times shorter.

1.4 **Trace Characteristics.**

Our traces are on a set of three 4-processor VAX traces and four post-processed single processor traces as shown in Table IV. These traces were collected by sites and Agarwal (1987). We describe below a summary of these traces. The reader is referred to Sites and Agarwal (1987) for more information about these traces and the post-processing details.

Abaqus is a parallel finite-element analysis program. It is manually decomposed to run on multiple processes, one per CPU, June9 is a batch multiprogramming workload, with no shared data except in the operating system. Cayenne is a parallel version of Spice, a circuit simulation program.

Make is a trace of two X-window network processes plus a disk copy and a make. Pero is a parallel VLSI layout routing program. Thor is a parallel logic simulation program and Pops is a parallel rule-based production system. We use these traces to simulate each individual cache of each CPU in the same way as Sites and Agarwal (1987).

Table IV Characteristics of Traces

trace	# refs	instr	data read	data write	cntxt switches
Pero	2840k	1812k	783k	245k	8
Thor	3283k	1517k	1390k	376k	21
Pops	3286k	1718k	1285k	283k	7
Abaqus	1196k	514k	599k	82k	292
June9	981k	550k	305k	126k	230
Cayenne2	3483k	1599k	1571k	312k	892
Make	853k	402k	251k	199k	16

Chapter Two

Trace Processing

2.1 **Trace Processing: An Overview**.

The ultimate objective of trace-driven simulation is, of course, to estimate the performance of a range of memory configurations by simulating their behaviour in response to the memory references contained in an input trace. This final stage of trace-driven simulation is often the most time consuming component because at this level we are often interested in hundreds or thousands of different memory configurations in a given design space. As an example, the space of simple caches defined by sizes ranging from 1 K-bytes to 1024 K-bytes (in powers of two), line sizes ranging from 1 word to 32 words (in powers of two), and associativities ranging from 1-way to 8-way, contains 264 possible design points. Adding the choice of different replacement policies (LRU, FIFO, Random), different set-indexing methods (virtually- or physically-indexed) and different write policies (write-back, write-through) creates thousands of additional possibilities. These design options are for a single cache, but actual memory systems are typically composed of multiple caches that cooperate and interact in a multi-level hierarchy. Because of these interactions different components cannot be considered in isolation. This leads to a further, combinatorial expansion of the design space. Our exploration of the subject material reveals that there are two basic approaches of dealing with this problem:

- Parallel distributed simulations
- Multiconfiguration simulation algorithms.

Since first approach doesn't fulfil the requirements of this project due to some inherent limitations so their discussion is out of scope. Algorithms that enable the simulation of multiple memory configurations in a single pass of an address trace offer another solution to the compute-intensive task of exploring a large design space. We use several criteria to judge a multi-configuration simulation algorithms in this survey. First, it is desirable that the algorithm be able to vary several simulation *parameters* (cache size, line size, associativity, etc.) at a time and, second, that it be able to produce any of several different *metrics* for performance, such as miss counts, miss ratios, misses per instruction (MPI), write backs and cycles per instruction (CPI). The *overhead* of performing a multiconfiguration simulation relative to a single-configuration simulation is also of interest because this value can be used to compute the effective simulation speedup relative to the time that would normally be required by several single-configurations simulations.

2.2 **Efficient Cache Simulation Using Multiconfiguration Algorithms.**

Almost two decades ago, Mattson (1970) and his colleagues presented an algorithm that can determine the performance (i.e., hit ratios) of all cache sizes under certain replacement policies with only a single pass through the trace file. Their techniques work because the replacement policies they studied guarantee inclusion, the property that, after any sequence of references; the contents of a cache are always a subset of any larger cache. This class of replacement algorithms is called stack algorithms, and the performance evaluation method is therefore known as stack simulation. This stack simulation technique was later extended to cover a wider range of cache organizations TRAIGER, I L., AND SLUTZ, D. R (1991).

The basic idea behind stack simulation is as follows. A common tag stack which holds the reference history is shared by caches of different sizes and with a different number of sets. When a block is requested, a search in the stack is performed until the block is found or the end of the stack is reached. Each element of the stack being visited is compared with the block's tag to determine whether this element is in the same set as the block. This is done for various tag lengths corresponding to the number of sets under study. An array of distance counters is used to keep track of the stack distances for caches with a different number of sets. At the end of the simulation, the distances are used to calculate the hit ratios of interest.

The stack simulation technique works well if the hit ratio is the only parameter of interest. For write-back caches, the frequency of replacing a dirty block is an important parameter since it substantially affects the bus and memory traffic. Unfortunately, this parameter cannot be obtained by using the original technique. This is because a block could be clean for some small caches but dirty for larger ones. Upon replacement, there is no way to tell whether the displaced block needs to be written back to memory. Thompson and Smith's (1989) addressed the write-back frequency problem in the context of fully associative caches. Their method is to attach a dirty level to each block in the stack. This allows them to count the number of writes that can be avoided when the dirty block is still resident in the cache.

Thompson and Smith (1989) were the first to propose efficient simulation algorithms which obtain useful performance measures other than hit ratio, although they considered only fully associative caches. Their technique was extended to a one-pass simulation method for more widely used set-associative write-back caches.

2.3 **Stack Algorithms.**

Design and research questions regarding memory hierarchies are often investigated with trace-driven simulation of several design alternatives. Mattson et al. (1970) developed the stack simulation technique for simulating many caches with one pass through an address trace. Stack simulation can evaluate alternative caches of many sizes if all have the same number of sets, the same block size, do no prefetching, and use a stack replacement algorithm

(e.g., LRU and RANDOM). Caches in a single stack simulation all have different associativities, however, since associativity is cache size in block (which varies in a stack simulation) divided by the number of sets (which is fixed). Design and research questions regarding CPU caches often examine caches of differing sizes, but fixed associativity Smith(1978), Clar(1983), Good(1983), Haik(1984), Hill(1987) and Puzak(1985). Consequently the evaluation of alternative CPU cache designs can require numerous stack simulations.

To reduce the number of simulations required, Hill (1987) has developed efficient one-pass trace-driven simulation algorithms for evaluating caches having differing numbers of sets. In some cases he reduces simulation time by using inclusion. He says cache C2 includes cache C1 if cache C2 contains a superset of the blocks in cache C1 after any series of references. A simulation of alternative cache designs can take advantage of inclusion by searching for a reference in cache C1 first, cache C2 second, and then in other caches that include cache C2. When a reference is found, a hit can be reported for that cache and (implicitly) for all caches that include that cache.

Hill (1987) shows when inclusion holds for caches having differing number of sets. He finds inclusion holds between practical direct-mapped (one-way set-associative) CPU caches, but that it does not hold in general between practical set-associative CPU caches. Since direct-mapped caches are important. He develops and algorithm, called forest simulation, for simulating alternative direct-mapped caches that takes advantage of inclusion. He allows alternative caches to use arbitrary functions to map references to sets. He also shows that faster simulation times can be achieved when the functions that map references to sets obey a property called set hierarchy. His algorithm is a generalization of an algorithm for simulating set-associative caches that map references to set with bit selection Mattson(1970), Trai(1971). A cache that uses bit selection contains a power of two number of sets and selects the set of a reference with the least-significant bits of the reference's block number.

Subsequent discussion is based on original work of Hill (1987) and Mattson (1991), and reviews set-associative caches, formally introduces stack algorithms, describes and analyzes linked-list stack simulation and describes more efficient methods of stack simulation.

2.3.1 **Set Associative Caches.**

A fully-associative cache allows any block to reside in any block frame. An n-way set-associative cache of c blocks uses a set-mapping function f to partition all blocks in main memory into a number of equivalence classes, and allows at most n blocks from each equivalence class to be simultaneously resident. The block frames that hold blocks from one equivalence class are called a set. The number of block frames in a set, n, is called the associativity (or degree of associativity or set size). The number of equivalence classes in the image of f, called the number of sets, is always equal to c/n, the number of blocks in a cache

divided by its associativity. The advantage of a set-associative cache with respect to a fully-associative cache of the same size is that n block frames rather than c block frames must be searched on each reference. The disadvantage of a set-associative cache is that it restricts which blocks can be simultaneously resident. For example, an n-way set-associative cache cannot contain the n + 1 most-recently-referenced block that map to one set. Figure 2-1 illustrates set-associative mapping and discusses Hill's notation for caches.

The most-commonly used set-mapping function is bit selection, because it can be implemented with no logic or delay. In bit selection, several low order bits of the block number are used to select the set. Bit selection requires that the number of sets be a power of two. For example, the set of block x is a cache with 21 sets that uses bit selection is $f(x) = \text{rem } 2^i$ is the remainder of dividing x by 2^i.

2.3.2. Stack Algorithms: Formal Definition.

The seminal paper on memory hierarchy simulation is Mattson et al. (1970). It introduces stack simulation as an efficient technique for evaluating a series of fully-associative caches and obeys the inclusion property. Since a set-mapping function partitions blocks into equivalence classes and set-associative caches do not allow blocks from different classes to interact, each set of a set-associative cache operates as an independent fully-associative cache. For this reason stack simulation can be applied to set-associative caches that use the same set-mapping function.

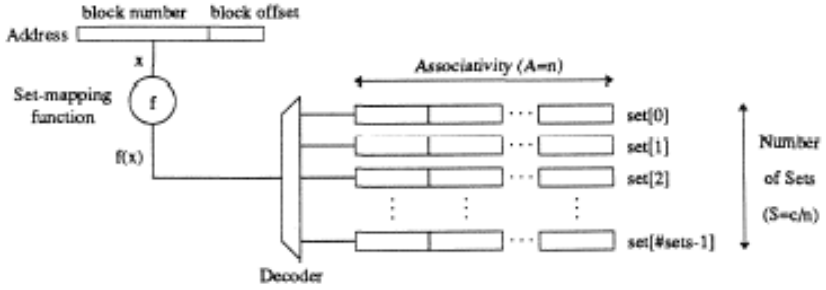


Figure 2-1. Set-Associative Mapping.

This figure illustrates set-associative mapping in an n-way set-associative cache of c blocks with set-mapping function f. If a block x is present, it is in one of the n block frames in set f (x) (one row). The number of elements in a single set is the associativity (degree of associativity, set size, A). The number of values in the image of f (number of rows) is the number of sets in the cache (S=c/n). The associativity times the number of sets is always equal to the cache's size in blocks. A cache is direct-mapped if A=1; it is fully-associative if S =1.

Hill(1987) denotes the above cache with "C(A=n, S=c/n, F=t)." where A,S and F are cache parameters "associativity." "number of sets" and "set-mapping function". When comparing caches he omits listing parameters

that do not vary. For example, I use "C(A=1)" and "C(A=2)" to contrast a direct-mapped and a two-way set-associative cache that are otherwise similar. When differences are clear, I use subscripts for distinguishing caches (e.g, "C 1", and :C_2"). Finally I use "c_i" (lower case) to represent the number of blocks in cache "C_i" (upper case). Thus "C_i" represents all attributes of cache C_i while "c_i" represents only the number of blocks in cache C_i..

Stack simulation is efficient because it takes advantage of inclusion, which is the property that, after any series of references, each larger cache simulated contains a superset of the blocks resident in all smaller caches. Inclusion may seem trivially true, but it is not. For example, a series of caches managed with FIFO (first in first out) replacement do not always obey inclusion. Consider a series of references to blocks 1,2,3,1, and 4. At the end of this sequence, a two-block cache will contain blocks 1 and 4 while a three-block cache will contain 2,3 and 4, but not block 1.

Assuming no perfecting and fixed block size, Mattson et al (1970). show that inclusion holds between caches using the same set-mapping function for a class of replacement algorithms called stack algorithms. LRU and RANDOM are the principal, interesting stack algorithms.

A stack simulation of caches C (A=*k*, F=*f*) for $k = 1$ to ***n*** uses a stack of ***n*** nodes for each set in the image of f, and an array of n distance counters. If we assume LRU replacement, each stack conceptually lists the most-recently-referenced n blocks for its set. Stacks in simulations of other stack replacement algorithms list blocks in order of descending priority, where priorities are defined so that blocks with a lower priority are preferred for replacement with respect to blocks with a higher priority. Each counter distance (k) contains the number reference so far to the k-th most-recently-referenced block. For each reference x, stack simulation performs three steps: *FIND, UPDATE and METRIC*.

FIND Locate block x in stack *f (x)*. we say a reference is found at distance k if it is the k-th element in the stack and at distance infinity (∞ if it not found.

METRIC Increment counters distance *[k]* and *N*, where *N* is the number of reference. At the end of simulation, the miss ratio of cache C (A=*k*, F=*f*) is 1- $\sum_{j=1}^{k}$ distance [k]/N. Metrics can be also be maintained by keeping counters only for specific cache sizes of interest. This will save space, but increase the time required to determine what counter(s) to increment.

UPDATE Update the stack to reflect the contents of all caches after the reference to *x*. See Mattson et al. (1970). for what is required with an arbitrary stack algorithm. For LRU, *x* must be moved from it old

position (if any) to the top of stack *f (x)*, all blocks x passes must be moved down one position, and all other blocks must not move, if *x* was not previously referenced, moved down one position.

2.3.3. **Linked-List Stack Simulation.**

Hill(1987) describes stack simulation with the stack for each set implemented with a linkedlist. This is commonly done for CPU cache simulations, because it is simple to implement and has adequate performance since the referenced block is usually found in the first few elements of the stack. He assumes LRU replacement, because it is commonly used; the arguments that follow can also be extended to other stack replacement algorithms. Figure 2-2 shows an example eight-entry stack before and after a reference.

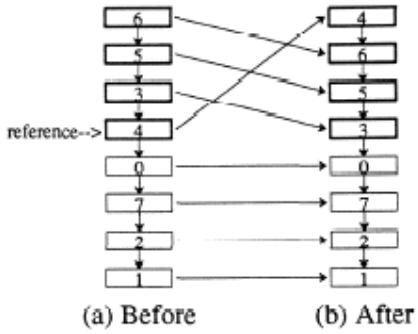


Figure 2-2. Stack Simulation Example.

The left stack (a) shows an LRU stack for one sort after a sequences of references to that set, Information in the stack reveals that block 6 is in this set of a direct-mapped cache (one block per set); blocks 6 and 5 are in a two way set-associative cache; blocks 6, 5 and 3 are in a three way cache;…and blocks 0 through 7 are in an eight way cache. Let the next reference that maps to this set be to block 4. The blocks in bold are examined to find block 4. The search stops when block 4 is found or the stack is exhausted. Since block 4 is located (coincidentally) at stack dept 4, a miss is recorded for all caches smaller than four blocks, and a hit is recorded for all caches 4 blocks or larger. The right stack (b) shows the stack after it has been updated with LRU replacement; the blocks in bold have moved.

The pseudo-code in Figures 2-3 and 2-4 illustrate the storage and the per-reference processing required by linked list stack simulation. The implementation of FIND (not shown) merely walks down the link-list *f(x)* until reference x is found or the linked list is exhausted. If *x* found, the implementation of UPDATE (also not shown) changes two pointers to move x to the head of the linked list *f (x)*. Otherwise, it allocates a new node for *x*, either from a free list or by reclaiming the last block in the list, and puts the node at the head of the linked list.

The analysis of the time to simulate each reference is some constant, *O* (I), that include the time to read the reference, plus the

number of iterations within FIND. Let δ_k be the probability that a reference is found at stack depth k, let δ_∞ be the probability that the reference is not found, i.e this is the first reference to that block, and let N be the number of references in the trace. FIND uses k iterations to find a reference at stack distance k, and $O(N\ \delta_\infty)$ iterations for stack distance ∞ where $O\ (N\ \delta_\infty)$ is the size of the entire stack. In practice, the average stack size is much smaller than the number of unique blocks in the trace, $N\ \delta_\infty$, because the unique blocks are distributed across a 100 or more sets.

```
 max assoc …. Maximum stack size
 function f (x) – a set-mapping function
integer number of stacks --- number of sets induced by f (x)
integer N---- number of references
----distance counts so that m (C(F=f,A=k)=1-j=1∑k  distance[j]/N
integer distance (1 max assoc)
define stacknode type {
        integer block number
        stacknode type next
}
stacknode type stack (0: number of stacks-1) --- top of stack pointers
---pool of dynamically linked stacknodes
stacknode type stacknodes (1 number of stacks max assoc)
```

figure 2-3 Stack Simulation Storage

```
 For each reference x (
        Read (var x)
        N + +
        Stack number = f (x)
        ---Walk down stack unitl x is found or stack is exhausted.
        ---If found, return stack distance and pointers to stacknode
        containing x.
        ---Otherwise set stack distance to max assoc + 1 and point to
        LRU stacknode.
        Found =FIND (x, stack number, var stack distance,  Var
        previous node pointer, var node pointer
        --METRIC
        ---if (found) distance (stack distance)++
--If was found, move the stack node of x to the top of its stack.
---Otherwise, store x in LRU stacknode and move it to the top
UPDATE(x, stack_ number found, previous node pointer node pointer)
```

Figure 2-4 Stack Simulation

The time integer to process a reference is of order:

$$\sum_{k=1}^{\infty} k\,\delta_k + O(N\,\delta_\infty)\delta_\infty + O(1).$$

The first term, called the mean stack distance, is the average number of distinct blocks since the last reference to the referenced block. If one is simulating only caches with associativity skmax, then no stack node need to be retained beyond distance kmax. This reduces the simulation time to;

$$\sum_{k=1}^{kmax} k\,\delta_k + kmax^* \sum_{k=kmax+1}^{\infty} \delta_k + kmax^*\,\delta_\infty + O(1).$$

Bounding stack size can significantly reduce simulation time of set associative CPU caches, where kmax rarely exceeds eight. However, for fully associative caches, kmax is equal to the number of block in the largest cache simulated. The run-time of linked list stack simulation of fully associative caches will be poor if either $\sum_{k=kmax+1}^{\infty} \delta_k$ or δ_∞ is large.

An analysis of the exact storage required for bounded linked list stack simulation of even large CPU caches is uninteresting, because the storage required is small relative to modern main memories. For example, the storage required by the linked list stack simulation pseudo code in figure 2-3 for simulating a direct mapped 128K-btye cache, a two way set associative 256 K byte cache, a four way512 K cache and eight way 1M cache with 32 byte blocks is approximately equal to the number of blocks in the 1M byte cache (32K) times 8 bytes per block, and is less than 300 K bytes.

2.3.4 **Other Stack Simulation Implementation.**

Bennett and Kruskal(1975) examine the paging behaviour of a large data base. They find mean stack distances of 1 to 328 entries for varying page sizes. Bennett and Kruskal propose an algorithm for stack simulation using an m-ary tree and a hash table where the run time per reference is approximately logarithmic in the number of block since the last reference to the current block. In contrast, the time per reference for linked list stack simulation is linear in the number of distinct blocks since the last reference of the current block. Bennett and Kruskal conclude that their algorithm is of order ten times faster than linked list stack simulation for mean stack distances of 150 entries. The storage requirements of the algorithm are large, but this is not important since the memory required is small relative to modern main memory sizes. The tree size is linear in the length of the address trace, N, and the hash table must be larger than the number of distinct blocks ($N\ \delta_\infty$). A simulation of 10 million references with 200.000 unique blocks requires only 3M bytes of storage if it uses two bits per reference and two words

per unique block. Olken (1981) changes Bennett and Kruskal's algorithm by replacing their m-ary tree with an AVL tree.

Bennett and Kruskal's algorithm and Olken's algorithm use a hash table to learn about a block's history. A hash table can also be used in linked list stack simulation to see if a block has ever been referenced. This reduces the time to process a previously unreferenced block from *Kmax* to a constant, reducing simulation time to:

$$\sum_{k=1}^{kmax} k\,\delta_k + kmax^* \sum_{k=kmax+1}^{\infty} \delta_k + O(\delta_\infty) + O(1).$$

This change will significantly improve performance only if both kmax and δ_∞ are large, that is both the degree of associatively and the fractions of the references to previously un-referenced blocks are large.

Thompson et al.(1989) examine each of these algorithms, and conclude that linked list stack simulation performs best for most CPU cache simulations. Consequently, Hill(1987) compared the performance of forest and all associativity simulation with linked list stack simulation only, and used stack simulation to linked list stack simulation.

2.4 **Inclusion in Set Associative Caches .**

Hill(1987) proves several theorems about inclusion for set associative caches using (possible) differing set mapping functions. Recall that Mattson et al, (1970) discuss inclusion only in caches that use the same set mapping function, and hence have the same number of sets (e.g. all are fully associative). In this section, as in the rest of this chapter, Hill assumes that all caches have the same block size, do no pre-fetching, and use LRU replacement. He wants to use inclusion to rapidly simulate alternative single level cache designs. Consequently when he discusses a large and small cache, he is considering using one or the other in a memory system, not using both as components in cache hierarchy.

Consider two caches, C,(A=n_1, F=f_1) and C (A=n_2, f_2), with blocks, associativities of n and set –mapping functions f_i, for i = 1,2 . An important condition necessary for cache C_2 to include (the blocks of) cache C_1 is that all blocks mapping to the same set in C_2 map to the same set in C_1,. That is, for all blocks *x* and *y:*

$$f_2(x)=f_2(y) \text{ implies } f_1(x)=f_1(y).$$

Hill (1987) calls this condition set hierarchy, because it means that f_2 induces a finer partition on all blocks than does f_1. Assume also that each set mapping function maps a large number of blocks(>=2* max(n_1 , n_2))to each set.

Set mapping functions used in real caches, including bit selection, trivially meet this restriction.

For cache C_2 to include cache C_1, C_2 must be at least as large as C_1 otherwise inclusion will be violated as soon as C_1, is full. For cache C_2 to include a different cache C_1, C_2 must be strictly larger than C_1. Hill(1987) considers two caches to be equivalent if they always contain the same blocks, i.e. , are identical up to placement of sets. Suppose cache C_1 and C_2 are the same size. For cache C_2 to include cache C_1 . It must always contain a superset of cache C_1blocks. Since cache C_2 contains the same number of blocks as C_1. It must always contain exactly the same blocks, and therefore is not a different cache. For this reason he sometimes refers to cache C as the larger cache.

2.5 **Simulating Direct Mapped Caches with Inclusion.**

The section introduces forest simulation for evaluating direct mapped caches that have the same block size and obey inclusion. Like stack simulation, forest simulation takes advantage of inclusion by searching for a block from the smallest to largest cache. When a block is found, a hit can be implicitly recorded in all larger caches. Forest simulation is so named because it uses a forest (a set of disjoint trees) rather than a stack to store cache blocks.

Let the direct mapped caches be named C_1 C_2… C_L. Assume that each cache C_i has c_i block frames and uses set mapping function rem c_i. While forest simulation works for arbitrary set mapping functions of the form rem c_i. Let $1 \leq c_1 < c_2 < \ldots < c_L$ and c_i divided c_{i+1} for i=1,L-1. By the argument presented after Theorem 3, inclusion holds for these caches.

The key data structure in forest simulation is a forest of L levels. The number of trees in the forest is equal to the number of blocks in the smallest cache, c. The c nodes of level I represent the blocks in cache c_1. The branching factor between two levels is equal to the cache size of the larger level, divided by the cache size of the smaller level c_{i+1} / c_i . The levels represent the blocks in the largest cache, c. This forest can be implemented as a heap containing twice as many nodes as there are blocks in the largest cache, since $c_{i+1} / c_i <= 2$ for all I implies $\sum_{i=1}^{L} c_i$ is less than 2* c_L . For example, the heap location of block x a cache of c blocks using set mapping function f can be calculated with *f (x)* + c. Figure 2-5 shows an example forest simulation forest.

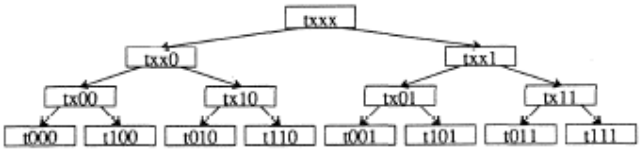
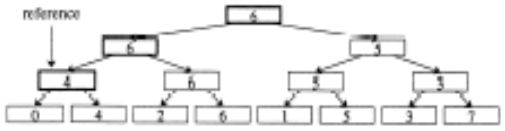


Figure 2-5. Forest Simulation Forest.

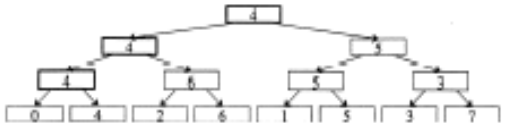
This figure displays the forest for caches of size 1,2,4 and 8 blocks. This forest contains only one tree, because the smallest cache contains only one block. This tree is

a binary tree, because each cache in this example is twice as large as the next smaller cache. In this example we assume blocks are mapped to block frames with bit selection. Each node holds the information for one block frame in direct mapped cache. The block at the root of the tree has no block number bits constrained, because a one block direct mapped cache can hold any block. This illustrated with a ***t*** *representing arbitrary high order bits of the block number and three x's representing don't cares for the three low order bits. The tags lxx0 and xx1 in the nodes of level two indicate that the blocks that can reside in these nodes are constrained to have even and odd block numbers, respectively. Similar rules with more bits constrained apply to the rest of the levels.*

Forest simulation works as follows and as is illustrated in Figure 2.6. On each reference, the algorithm selects the tree corresponding to the set of the reference in the smallest cache.



(a) Before



(b) After

Figure 2-6. Forest Simulation Example.

The top tree (a) depicts the forest of Figure 2-5 after a series of references. Information in the tree tells us that block 6 is in a cache of size one block: blocks 6 and 5 are in a direct mapped cache of size two: blocks 4.6,5, and 3 are in a direct mapped cache of size four: and blocks 0 through 7 are in a direct mapped cache of size eight.
Let the next reference be to block 4. A path from the root to a leaf is determined using the set mapping function for each cache (here bit selection is assumed). A search begins at the root and stops when block 4 is found. All nodes encountered in the search that do not contain block 4 are modified to do so. The blocks on bold are examined to find block 4. Since block 4 is located at level 3, caches 1 and 2 miss and caches 3 and 4 hit.
The bottom tree (b) shows the tree after this reference as been processed. The nodes in bold now contain the referenced block.

Then it searches for the referenced block beginning at the root of the tree. The path of the search is determined by the set of the reference in each cache. Any time a node is encountered that does not contain the reference, the node is updated to contain it. The processing of a reference stops when the

reference is found, or after a leaf node has been modified. If the reference is found at level i ,a counter distance (i) must be updated.

2.6 **Simulating Set-Associative Caches without Inclusion.**

Stack and forest simulation will simulate a series of caches with one pass through an address trace. Both methods are "efficient," because they take advantage of inclusion. Since inclusion does not hold for caches of all sizes and associativities, algorithms using inclusion must constrain the series of caches simulated. Hill(1987) describes an algorithm, which he calls all associativity simulation, that does not use inclusion, but can simulate set associative caches with the same block size, that do no prefetching, and use LRU replacement, with one pass over an address trace. With it, he can cover the design space in 3 simulations (one per block size) instead of 15 runs of stack simulation. The algorithm permits the set associative caches use of arbitrary set mapping functions. A literature search revealed that a version of all associativity simulation, where all set mapping functions use bit selection, was developed by researchers at IBM Mattson(1970), Trai (1971).

2.7**. Comparing Actual Simulation Times**

Hill (1987) compares the simulation times of implementations of stack, forest, and all-associativity simulation. While the exact quantitative results of this section do not necessarily apply to other implementations, there is no reason to believe that gross comparisons do not generalize. The advantage of this data over the run-time analysis is that these results apply to at least one set of implementations of these algorithms.

Hill (1987) has implemented stack, forest and all-associativity simulation in C under UNIX 4.3 BSD. Stack and forest simulation were added to a general cache simulator, called DineroIII Hill (1985). Dinero III originally contained 1250 C statements, as measured by the number of source lines containing a semicolon or closing brace. Adding stack simulation increased total code size by 150 statements, adding forest simulation, 220 statements. Stack simulation is implemented using linked lists and without using a hash table to detect previously unreferenced blocks. The forest simulation implementation restricts the set-mapping functions to be the block number modulo the cache size in blocks, a generalization of bit selection. He has implemented all-associativity simulation in a separate program, called Tycho, containing 800 C statements and having far fewer options than Dinero III. Tycho restricts the set-mapping functions to be bit selection. His implementations of these algorithms are available to interested researchers free of charge.

He estimates simulation time with the elapsed virtual time (user plus system) returned by the UNIX 4.3 BSD system call getrusage on an otherwise unloaded Sun-3/75 with 8M of memory and no local disk. Trace data is read from a file server via an Ethernet. He gives the results for four traces from four different architectures, described in Table 2-1, despite finding that results are

fairly insensitive to program traces. All caches simulated have 32-bytes blocks, do no prefetching, use LRU replacement, are mixed (data and instruction cached together), and use bit selection.

He begins by verifying that implementations of the three algorithms have similar run-times for simulating a single cache, using two methods. First, he ran each implementation using a trace of 1 million identical addresses so that all references, expect the first, hit at distance one. Results show that the elapsed virtual times of forest and stack simulation differ by 0.1 percent, while all-associativity simulation is 3 percent faster. All-associativity simulation is faster, because it is implemented in a different program, Tycho. It is not surprising that Tycho is slightly faster than Dinero III is a general cache simulator. Even through Dinero III's additional features are not used in these simulation runs, DineroIII uses some execution time to fall through the if statements that guard the additional features.

Second, he compares the algorithms simulating a 16K-byte direct-mapped cache with each of four traces; the results are similar to those above. In addition a stack and an all-associativity of a single 16K-byte four-way set-associative cache are also comparable.

Since his implementations of these algorithms have similar run-times for simulating single caches, and the time differences are not due to per reference overheads, thus these lead us to meaningful comparison of multiple cache simulations.

Chapter Three

Implementation

3.1 **Premise.**

Till now we have discussed the historical background with reference to trace driven simulations with special emphasis on trace processing techniques. From this discussion one thing becomes clear that not much research has been done as far cache simulations in multiprocessor environments are concerned. Whatever knowledge about multiprocessor environments has been given is in the form of slight clues that too only highlight the challenges involved and does not discuss the solution space.

In this chapter we start our discussion, from the implementation of already discussed theoretical aspects of uniprocessor based environments, in the form of algorithms and gradually extend the idea towards multiprocessor based environments. In the end of this chapter we present our algorithm that works in befitting manner in multiprocessor based environments.

3.2 **Trace-Driven Simulation**

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

In a classic paper, Mattson et al. (1970). showed that for certain replacement policies the miss ratios for all cache sizes could be calculated in a single pass over the reference trace. These policies are collectively known as stack algorithms. The technique depends on the inclusion property of these policies; the contents of any size cache includes (i.e. is a superset of) the contents of any smaller cache. Thus the cache at any time can be represented as a stack, with the upper k elements of the stack representing the blocks present in a cache of size k. The current stack level of any block is therefore the minimum cache size for which the block is resident. If a block is referenced while at level k, it is a "hit," and therefore resident, for all sizes k and larger. The level at which the block is found is referred to as its stack distance; see Figure 3.1 Using stack analysis, it is possible to compute the miss ratio of Equation ($MR_R(C) = m(C)/N$) for all sizes by recording the hits to each level.

$$\mathrm{MR_R}(C) = \left(N - \sum_{i=1}^{C} hits(i)\right) \Big/ N,$$

The miss ratio for a cache size C is where N is again the total number of references. Notice that, since hits (i) are never negative, this is a non increasing function of cache size. All stack algorithms possess this characteristic, whereas non

stack algorithms may show points at which performance declines with increased cache size.

The simplest example of a stack algorithm is the Least Recently Used (LRU) policy. The stack always contains the blocks in order of last reference, with the most recently referenced block on the top. For any cache size C, the LRU block for that cache size is the block at level C in the stack. When a block at level k is referenced, it is not in any cache smaller than k, and therefore it must be fetched. The block that must be removed from any cache of size j, j smaller than k, is the block at level j. The stack is updated by simply "pulling" the referenced block out of the stack and placing it on top. All blocks down to level k are effectively "pushed" down one level. Since the referenced block was in all caches k or larger, all blocks below level k remain unchanged. Figure 3.1 illustrates these operations for the case where the referenced block is at stack level 4 and the case in which the block is not currently in the stack.

More generally any stack algorithm possesses a "priority function" which imposes a total ordering on all blocks at any given time, independent of cache size. Notice that LRU imposes such an ordering based on the time of last reference.

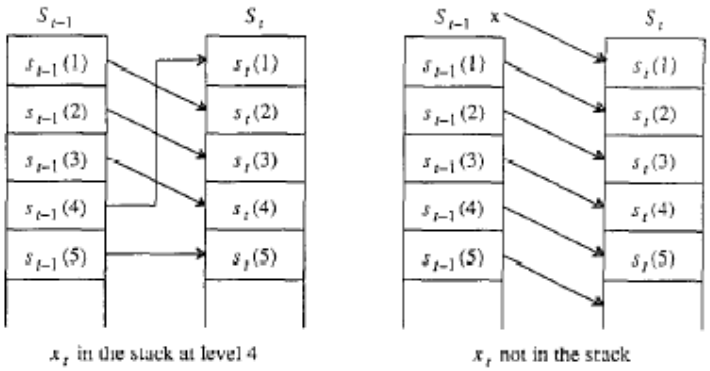


Figure3.1. Examples of stack maintenance using LRU replacement. The referenced block is always "pulled" to the top of the cache stack. All blocks with smaller stack distance are pushed down one level.

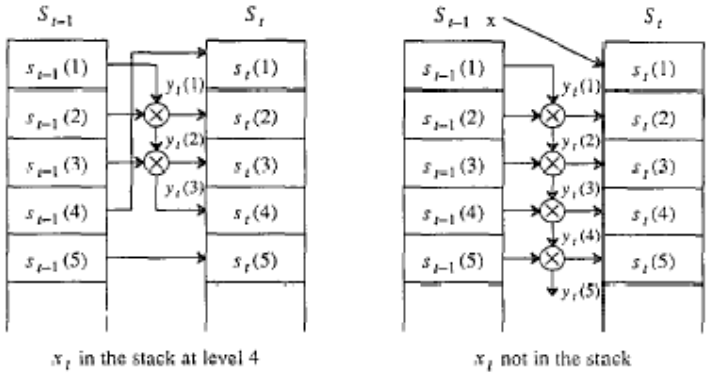


Figure3. 2. Examples of stack maintenance using a stack replacement algorithm. For each level C a single comparison (indicated by a circled cross) between the prior block at the level (st-1 (C)) and the block pushed from above (y_t (C - 1)) determines the new block at the level and the block pushed from the level. Update continues down to the current level of the referenced block or to the bottom of the stack if X_t is not in the stack.

However, in the more general case the relative priority of two blocks may change without either of them being referenced. (See, for example, Figure 3.4 where the relative positions of blocks A and B reverse between times 5 and 6. It is no longer the case that the block at level j is necessarily the one to be pushed from that size cache. This complicates the stack update procedure, but only slightly. The stack can still be updated in a single pass that is similar to one pass of a bubble sort. A single comparison at each level determines the new block at the level and the block pushed from the level.

First, the referenced block is still pulled to the top of the stack since it must become resident in all cache sizes. Using the terminology from Mattson et al. (1970), let y_t (C) be the block pushed ("yanked?") from a cache of size C. To make room for the referenced block, the top block in the stack,$s_{t-1}(1)$ must be pushed from a one-block cache, becoming y_t (l). Some block must also be pushed from a two-block cache-the one with the lowest priority. A single comparison between y_t (1)= $s_{t-1}(1)$ and $s_{t-1}(2)$ determines which becomes y_t (2). (Ties are broken by some arbitrary rule.)
Algorithm 1. General Stack Analysis Algorithm

Algorithm 1. General Stack Analysis Algorithm

1. FOR $1 \le t \le N$ DO — *For all events.*
2. IF $x_t \notin S_{t-1}$ THEN $\Delta = \infty$ — *If not referenced before.*
3. ELSE DO
4. FIND Δ SUCH THAT $s_{t-1}(\Delta) = x_t$ — *Find the stack distance.*
5. $rh(\Delta) = rh(\Delta) + 1$ — *Update the read hits.*
6. IF $\Delta \neq 1$ — *If stack needs updating.*
7. $y_t(1) = s_{t-1}(1)$ — *Calculate push from each level down to the referenced block. Push is the minimum of the block at level and push from above.*
   FOR $2 \le i \le \Delta$ DO
   $y_t(i) = pmin[y_t(i-1), s_{t-1}(i)]$
   FOR $i \ge \Delta$ DO $y_t(i) = \varnothing$ — *No pushes below reference.*
8. $s_t(1) = x_t$ — *Establish new stack.*
   FOR $i > 1$ DO $s_t(i) = s_{t-1}(i) + y_t(i-1) - y_t(i)$

Notes:

(1) In step 5, all counts that are not incremented are assumed to remain unchanged at the next time interval; thus the subscript t is not used.

(2) In step 7,pmin returns the block with the lowest priority, as defined by the replacement algorithm. Pmin is the comparison function in the circles of Figure 3.2.

(3) In step 8, plus and minus have the intuitive meaning of adding a member to a set or removing a member. In this context, adding a member that is already present, or subtracting a member that is not present, have no effect. The same is true of adding or subtracting ϕ. Thus, the block kept at level i, $s_t(i)$, is either $s_{t-1}(i)$ or the block pushed from above, $y_t(i-1)$, whichever is not pushed from level i.

Figure 3.3 General Stack Algorithm

Similarly, the block pushed from a three-block cache should be the lowest priority of the three blocks previously present. However, the lowest priority block can be determined in a single comparison of y_t (2) and s_{t-1} (3) since the third block, now st (2), has already "won" a comparison against yt (2), and thus cannot have the lowest priority. Similar logic applies for all levels down to level k, the original level of the referenced block; only the block currently at the level and the one pushed from above

need be compared to find the block to be pushed. The contents of all sizes larger than k are again unchanged.

The stack analysis algorithm is formally presented in Figure 3.4. This algorithm is used as the basis for the extensions. Let:

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

S_t = the cache stack just after reference to x_t, with $s_t(C)$ = the block at stack level C. $s_0(C) = \phi$ for all C.

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

$y_t(C)$ = the block pushed ("yanked") from cache of size C by reference x_t.

$rh(C)$ = a count of the number of hits to level C by time t.

Figure 3.4 The stack analysis algorithm

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

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

Figure 3.5. Cache contents using the Least Frequently Used policy. The number beside the block is the priority, that is, the number of references.

As an example, consider the application of a Least Frequently Used policy to the reference string {AAABBCCDB). Using this policy, the block pushed from any cache is the one that has been used the fewest total times. Figure 3.5 shows the contents of the stack after each reference, where the number beside each block is the priority (i.e. the number of uses of the block). Notice that a block may be pushed several levels because of a reference, as seen at time 8. Note too that blocks below the level where the referenced block is found are unchanged, even though they may have higher priorities, as seen after the last reference.

3.3 **Non stack Algorithms.**

The prohibition against a priority function that depends on cache size prevents some otherwise simple policies from being stack algorithms such as the First-In First-Out (FIFO) rule. Another common technique that is not a stack algorithm is the use of demand prefetch or prefetch-on-miss . Suppose that the prefetch policy is to fetch the following block along with any fetched block, but not to prefetch if the referenced block is already present. Assume an arbitrary stack algorithm for replacement. It is easy to construct counterexamples that violate inclusion because the priority of a

prefetched block depends on when it is fetched, which varies with cache size. For example, consider the examples of Figure 3.6, where the contents of a larger cache are clearly not a superset of a smaller cache after the final reference.

It is possible to construct prefetch policies that are stack algorithms. For example, the non demand policy that always prefetches the next block, regardless of whether the referenced block is resident, is a stack algorithm. This policy is a form of One Block Lookahead or OBL. From the point of view of the stack, this is equivalent to the insertion of a reference to the next block after each reference. Non demand prefetch is not practical if the cost of a fault is high, as it is in virtual memory systems, for example, because the penalty for faulting to prefetch a block that may not be needed is greater than the potential gain. Non demand prefetch is practical when it is possible to look for the next block in the cache and prefetch it if necessary without significantly slowing down processing the current reference. This is the case for many large processor caches and file system caches.

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

(a)

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

(b)

Figure3.6. Cache contents using one-block demand prefetch. Since this is not a stack algorithm, the contents of each cache size are listed separately. In both examples the next block is prefetched only if the referenced block is not present. In all cases the referenced block becomes the highest priority, followed by the prefetched block, if any. The inclusion property is violated after the last reference in both cases.

3.4 **Extensions to Stack Analysis.**

There have been several important extensions to the basic stack analysis technique. Mattson et al. (1970) showed how the hit ratio can be computed for an arbitrary number of levels, assuming a common block size and replacement policy. Gecsei (1974) showed how it could be generalized to multiple levels with different block sizes for LRU and certain related policies. Traiger and Slutz (1991) showed that it is possible to compute miss ratios for variable block sizes and variable associativity in a single pass.

Coffman and Randell (1971) investigated the "extension problem," that is, predicting the performance of cache sizes greater than C, given only the misses from cache size C instead of a full trace. For LRU, a trace of "pushes" and "pulls" was sufficient; for other stack algorithms, the priority ranking for the block pushed and all

blocks not in the cache of size C was also required. A trace of misses only was found to be sufficient for providing good approximations to the performance of larger caches.

A more recent extension by Silberman (1983) showed that stack analysis can be applied to a delayed-staging hierarchy in which the processor directly accesses several levels of the memory hierarchy. When a referenced block is not in a higher level cache, it is supplied to the processor (at the speed of the highest level cache to contain the block) and begins "migrating" into the higher caches. The time elapsed until it becomes "staged" (resident) in a higher cache is equal to the sum of the access times of the caches below it. Further, the displacement of a block in the higher level cache is also delayed, creating a situation where the stack level of a block may be a function of the size of several lower level caches and the time since the last reference to one or more other blocks. Silberman showed that stack analysis can be applied to this class of hierarchy by maintaining the time and cache depth of the last "migration" for each block. This information is used at the time of each reference to compute the stack distance of the block for different sizes of each level, by considering the delayed staging times. This idea of maintaining additional information about each block is seen again in our write-back algorithm presented in the next section.

3.5 **WRITE-BACK STACK ALGORITHM.**

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

3.5.1 **The Write-Back Problem.**

In write-back, a write access to secondary storage occurs whenever a dirty block is "pushed." The main problem with write-back is maintaining the "state" (clean or dirty) of each block in the stack. A single dirty bit is sufficient in the real cache, but not for the simulation stack. Consider a read to a dirty block at level k. For sizes h and larger the block is still dirty, since it has not been written; for sizes 1 to k - 1 it is clean. The inclusion property is violated since the contents of the larger cache are "different" in the sense that the block has different attributes in some larger sizes. A second problem is accounting for the "dirty pushes." Each miss from a memory of size C causes a push from each smaller memory; that pushed block may be dirty. On first inspection, this suggests that counts need to be maintained and updated for every memory size from which a dirty block is pushed. We show that a surprisingly simple technique solves both of these problems.

3.5.2 **A Non Stack Algorithm.**

We begin by assuming that write-back is not a stack algorithm and by imagining a general algorithm for computing write-back miss or transfer ratios. The algorithm is based on the stack analysis algorithm, but maintains a separate set of dirty blocks for each cache size in order to solve the problem of the no ninclusion of dirty bits. Let:

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

$D_t(C)$ = the set of dirty blocks in a memory of size C after the reference to x_t.

$p_t(C)$ = the dirty block "pushed" from a memory of size C by reference x_t

$$= \begin{cases} y_t(C) & \text{if } y_t(C) \in D_{t-1}(C) \\ \phi & \text{otherwise} \end{cases}$$

$dp(C)$ = the number of blocks written back from a memory of size C by time t.

Algorithm 2. General Nonstack Write-Back Algorithm

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

Figure 3.7

We define Algorithm 2 in Figure 3.7 by adding steps 7A and 8A to Algorithm 1 presented in section 3.2. When a block is written, it must be added to each dirty set (line 8A). A block is removed from a set if and only if a dirty block is pushed from memory (line 7A). Note that if write fetch is not used, then line 5 of Algorithm 2 must be conditioned on a read, that is, IF $w_t = \Phi$ THEN rh(Δ) = rh(Δ) + 1.

3.5.3 **Dirty Set Inclusion Property.**

The inclusion property of stack algorithms states that if a block is present in memory of size C, then it is present in size C + 1, and therefore in all larger sizes. This can be formally stated as

$$\text{IF } w_t = \phi \text{ THEN } rh(\Delta) = rh(\Delta) + 1.$$

for all t and C, where M_t (C) is the set of blocks present in a memory of size C after reference .x_t. We now show that a similar condition applies to dirty sets; that is, if a block is dirty in a memory of size C, then it is dirty in all larger sizes.

An intuitive argument of this fact is the following. In order to become dirty, a block must be written, which makes the block dirty in all sizes. A block becomes clean only when it is replaced (ignoring deletions for now). Because the replacement algorithm is a stack algorithm, the block is always pushed from a smaller cache before it is pushed from a larger one. The dirty level is therefore the maximum level to which the block has been pushed since it was written. A read may pull the block to the top of the stack, but will leave it dirty in an inclusive set of sizes. There is no way to make it dirty in some sizes without making it dirty in all sizes; therefore, inclusion holds. A more formal proof follows.

$$D_t(C) \subseteq D_t(C+1), \textit{ for all } t \textit{ and } C.$$

Proof. Choose an arbitrary C. The condition certainly applies at the start of the simulation when no blocks are in cache; therefore, the dirty sets are empty. Assume it to be true at time t - 1. Beginning with this induction hypothesis, the proof adds and subtracts blocks from each side, preserving the subset relation, and finally arrives at an expression for the dirty sets at time t.

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

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

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

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

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

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

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

Note that the right-hand side is exactly Dt(C + 1) as computed by line 8A of Algorithm 2, whereas the left-hand side differs from D_t(C) only by the term pt (C + 1). There are three possible values for pt (C + l), none of which affect the set on the left-hand side:

(1) If $y_t(C+1)$ is not dirty, then $p_t(C+1) = \phi$.

(2) If $y_t(C+1)$ is dirty, and $y_t(C+1) = y_t(C)$, then $p_t(C+1) = p_t(C)$. Subtracting this block twice cannot affect the contents of the left-hand set.

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

Removing this term gives

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

which is exactly equal to

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

as set by Algorithm 2, line 8A.

With these facts we can simplify the algorithm considerably.

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

Above relation implies that there is a minimum size at which a block is dirty (if it is dirty at all). Intuitively, this is the smallest memory from which the block has not been pushed since its last write reference, and therefore, the smallest memory size in which it is still dirty. This is also the largest stack distance the block has attained since it was last written. Therefore, the separate Dt(C) can be replaced by a single array. Let dl(x) be the dirty level of block x or infinity, if the block has never been written. A block at level k (i.e., s(k) = X) is dirty if and only if *dl*(x) ≤ I k. We can set the dirty level to 1 when a block is written and update it as the block is pushed.

3.5.4 **Writes Avoided.**

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

Ignoring deletes for now, a write is avoided only when a dirty block is overwritten, since both the previous and current modification can be written by the next write-back. Therefore, we can say that the previous write has been avoided for all sizes equal to or greater than the current dirty level. Notice that we now only care about the dirty level for the block being referenced, and therefore, we only need to adjust dl for the referenced block. If it is found at level Δ which is below its dirty level (i.e. Δ > dl(x,)), we can reason that the block has been pushed (while dirty) from all levels between dl (x_t) and Δ; therefore, the proper value for dl (x_t) is Δ; see line 6 of Algorithm 3 (Figure3.8).

We now define wa(C) to be the writes avoided at level C, that is, the number of writes for which the referenced block was still dirty in memory sizes C and larger. The write-back stack algorithm, Algorithm 3, is shown below. The differences between this algorithm and Algorithm 1 are line 6, which adjusts the dirty level as described above, and lines 10-13, which count the writes avoided and write references and reset the dirty level to one on a write.

For the special case of LRU, this algorithm is particularly simple. As in the standard stack analysis algorithm for LRU, updating the stack is a Matter of removing the referenced block and inserting it at the top of the stack. The fact

that only the referenced block affects the statistics is particularly useful for this case since no work needs to be done while searching for the referenced block.

3.5.5 **Dirty Push Computation.**

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

$$dp(C) = W_t - \sum_{i=1}^{C} wa(i) - |D_t(C)|,$$

Equation 3.1

where the count of write references by time t is

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

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

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

Algorithm 3. Write-Back Stack Algorithm

```
1.  FOR 1 ≤ t ≤ N DO                                          For all events.
2.    IF x_t ∉ S_{t-1} THEN Δ = ∞                             If not referenced before.
3.    ELSE
4.      FIND Δ SUCH THAT s_{t-1}(Δ) = x_t                     Find the stack distance.
5.      rh(Δ) = rh(Δ) + 1                                     Update the read hits.
6.      IF dl(x_t) < Δ THEN dl(x_t) = Δ                       Set the "real" dirty level.
7.    IF Δ ≠ 1                                                If stack needs updating.
8.      y_t(1) = s_{t-1}(1)                                   Calculate push set.
        FOR 2 ≤ i < Δ DO y_t(i) = pmin[y_t(i - 1), s_{t-1}(i)]
        FOR i ≥ Δ DO y_t(i) = ∅
9.      s_t(1) = x_t                                          Establish new stack.
        FOR i > 1 DO s_t(i) = s_{t-1}(i) + y_t(i - 1) - y_t(i)
10.   IF w_t ≠ ∅ THEN                                         If this is a write.
11.     IF dl(x_t) ≠ ∞ THEN
          wa(dl(x_t)) = wa(dl(x_t)) + 1                       Count writes avoided.
12.     dl(x_t) = 1                                           Block is dirty.
13.     W_t = W_{t-1} + 1                                     Count of write references.
```

Figure 3.8

The first two terms of (3.1) are obvious, but we should elaborate on the need for the third term. It should be clear that each block that is still dirty has avoided the most recent write for all sizes in which it is still dirty and should therefore be subtracted from the count of writes. This argument applies at any point during the trace and at the end of the simulation. Since the relevant metrics are those gathered during the trace period, regardless of any activity which occurs after the trace ends, we should consider each dirty block remaining at the end of simulation as having avoided a write. To simplify the computations, we can make a final scan of the memory stack and update wa(dl(x)) for each dirty block x. We can then eliminate the third term of (3.1). Of course, the effect of this should be small if the total number of trace events is large.

Using this expression for the number of dirty pushes leads to a simple recurrence for computing the transfer ratio.

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

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

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

Substituting (3.1) for dp(C) and assuming that the final scan has updated wa,
this simplifies to

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

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

Equation 3.2

or

$$T(0) = (N + W_t)/N$$
$$T(C) = T(C-1) - [(rh(C) + wa(C))/N].$$

Equation 3.3

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

3.5.6 **Warm Start.**

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

Warm start using the write-back algorithm can produce an anomaly in the transfer ratio. This is caused by the final scan of memory which considers

all dirty blocks as having avoided a write, which may have occurred before the warm start. Suppose, for example, that the write-back simulation is warm started, and suppose that W, and wa are zeroed. Then immediately after warm start, the value of dp(C) calculated using (3.1) may be negative for some values of C, as shown in Figure 3.9(c), where the number in parentheses is the dirty level of the block. Of course, a "negative push" is meaningless. We can keep the numbers positive by setting Wt to the number of dirty blocks in the cache at warm start, but then dp is immediately nonzero for some cache sizes. Another alternative would be to zero both wa and dl, but then it will be a long time before any dirty block could be pushed from large sizes-in conflict with the reason to warm start in the first place.

Since the third term of (3.1) increases with C, the second term of (3.1), the sum of wa must decrease for larger C if we want the computed value of dp to be zero immediately after warm start. This can only happen if some wa are negative. The solution we use is to zero wa at warm start, then decrement wa[dl(x)] for all dirty blocks X. With this solution dp(C) is zero immediately after warm start for all C, as it intuitively should be; see Figure 3.9(a). Now suppose that a reference to a previously unreferenced block causes all blocks to be pushed (Figure 3.9(b)). The result is that dp(C) is zero for all sizes except those from which a dirty block is pushed-which is exactly the result obtained from a simulation of a single cache size or a real cache.

Note, however, the unexpected result that the transfer ratio due to dirty pushes is no longer a monotone decreasing function of size. In fact, if the warm start of Figure 3.9(a) were followed by the unlikely event of five total misses,

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

(a)

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

(b)

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

(c)

Figure 3.9. Revised count of dirty pushes after warm start: (a) is immediately after warm start, and (b) is after all blocks are pushed one level. The count of dirty pushes from each size, dp(C) agrees with the results from a real cache.

the resulting transfer ratio would be increasing with cache size. It seems that the rate of dirty pushes may be exaggerated for larger cache sizes by the fact that there are more dirty blocks in the larger cache. (There may also be a higher probability that blocks pushed from larger caches are dirty). This "error" for large sizes is bounded by the number of dirty blocks in the stack divided by the number of references after warm start. It can therefore be made arbitrarily small by increasing the number of references after warm start (which also reduces the need for warm start). In most cases, locality causes the write-back traffic ratio to assume its normal decreasing form.

3.6 **Write-Through**

This policy is trivially included in the algorithm by setting $dl(x_t)$ to infinity instead of one after a write. In fact, since the total number of write requests is known, both the write-back and write-through transfer and traffic ratios are available simultaneously. It is also possible to simulate a combination of policies provided the choice of policy is not a function of memory size. For example, some blocks could be write-through and others write-back, a scheme used in some real caches, for example, the Intergraph CLIPPER processor [5,20] and the NEC disk cache[36].

An example of an algorithm for such a cache is given as Algorithm 4 (Figure 3.10). The only difference between this and Algorithm 3 (Figure 3.8) is that Line 12 ensures

Algorithm 4. Mixed Write-Back/Write-Through Stack Algorithm

```
1.  For 1 ≤ t ≤ N DO                                          For all events.
2.    IF x_t ∉ S_{t-1} THEN Δ = ∞                             If not referenced before.
3.    ELSE
4.      FIND Δ SUCH THAT s_{t-1}(Δ) = x_t                     Find the stack distance.
5.        rh(Δ) = rh(Δ) + 1                                   Update the read hits.
6.      IF dl(x_t) < Δ THEN dl(x_t) = Δ                       Set the "real" dirty level.
7.    IF Δ ≠ 1                                                If stack needs updating.
8.      y_t(1) = s_{t-1}(1)                                   Calculate push set.
        FOR 2 ≤ i < Δ DO y_t(i) = pmin[y_t(i − 1), s_{t-1}(i)]
        FOR i ≥ Δ DO y_t(i) = ∅
9.      s_t(1) = x_t                                          Establish new stack.
        FOR i ≥ 1 DO s_t(i) = s_{t-1}(i) + y_t(i − 1) − y_t(i)
10.   IF w_t ≠ ∅ THEN                                         If a write.
11.     IF dl(x_t) ≠ ∞ THEN                                   Update dirty pushes.
          wa(dl(x_t)) = wa(dl(x_t)) + 1                       Count writes avoided.
12.     IF BLOCK IS WRITE-BACK THEN                           Write to write-back block.
          dl(x_t) = 1                                         Block is dirty.
13.     W_t = W_{t-1} + 1
```

Figure 3.10

that only write-back blocks become dirty. Writes to both write-back and write-through blocks are counted in W, (Line 13), but only writes to write-back blocks are avoidable (Line 11) since write-through blocks are never dirty. As a simplification, both write-back and write-through are counted the same. In reality, a write-through may involve less data and, therefore, is less costly. This algorithm assumes that write allocate and write fetch are performed for write-through blocks; if this is not the case, then Line 5 and the stack update should be bypassed for a write-through miss .

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

Figure. 3.11. A gap "jumps" down the stack. The gap (γ) is unaffected by references B above it, but a reference below it causes it A to "jump" to the level of the referenced c block.

3.7 **Deletions.**

An important consideration in file system studies is the existence of deletions in the reference string. If a file is deleted, the blocks of that file should be removed from the cache without a write. With a write-back cache and short file lifetimes, it is likely that file blocks will be created and deleted without ever being written to the next level. Deletions also occur in processor caches when blocks are invalidated but generally not without writing the block first if it is dirty. Deletion of blocks from the cache was discussed by Mattson et al. (1970) in the context of a "call back" hierarchy, where cache blocks may be invalidated by a write directed to a lower level. The example used by Mattson(1970) is a virtual memory system in which all I/O occurs to blocks residing in an I/O Subsystem, not the CPU memory. If an I/O is addressed to a block which is in CPU memory, that block must be invalidated. Greenburg(1974) also discusses deletions and implements an algorithm to approximate the effect of deletion. Olken (1983) proposes an exact algorithm and discusses implementation using various data structures. None of these consider the effect of write-back. If a deleted block were simply deleted from the stack, the stack level for all lower blocks would be reduced. This would have the undesirable effect of calling these blocks back into a memory from which they had been pushed. Instead, what called a "marker" block is inserted in the stack replacing the deleted block.

We refer to the marker blocks as gaps in the stack, corresponding to a vacant block in all larger caches. The next push from above the gap replaces the gap with the pushed block since no block needs to be replaced in a cache containing a vacant block. Thus a gap stops the sequence of stack updates, just as finding the referenced block stops the pushes in the normal case. However, since the referenced block must still be pulled to the top and blocks below the referenced block do not change stack level, the referenced block must be replaced in the stack by another gap. Thus, a reference to a block below the first gap seems to make the gap "jump" down the stack. As an example, consider the sequence of Figure 3.12. After block D is deleted, a gap is left at level 4. A reference to block B above level 4 does not affect the gap. However, the reference to block F below level 4 "jumps" the gap to the stack level of F. From the point of view of the "real" cache, the gap represents the same vacant block, which was in all memory sizes 4 or larger. Since block F is already resident in memories of size 6

Algorithm 5. Write-Back Stack Algorithm with Deletes

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

Figure3.12

or larger,the reference to F has not fetched any block to fill the gap. Therefore, the gap still exists in these sizes. The effect of deletions on the transfer ratio is to introduce another way in which a write can be avoided, particularly evident in large cache sizes. If a block is written, then deleted before it is pushed, the write-back is avoided for all sizes greater than the current dirty level. It is therefore a simple matter to increment the appropriate wa[dx(x_t)] on deletion. In addition, the count of read hits must exclude deletes, since a deleted block is never fetched. This is seen in lines 6 and 7 of Algorithm 5. The complete, though somewhat complicated, algorithm for write-back with deletions is given as Algorithm 5 (Figure 3.12). Let:

γ = a gap marker in the stack.
Γ = the level of the first gap in the stack.
Δ' = min(Δ, Γ), the level at which pushes stop.

There are actually only a few changes between Algorithm 3 and Algorithm 5. First, line 6 handles a deletion by updating the count of writes avoided and replacing the block by a gap in the stack. Line 8 computes Γ, the level of the topmost gap, while line 9 determines whether the referenced block or Γ stops the sequence of updates. Line 10 uses this value instead of Δ. A subtle change in line 13 inserts x_t on top of the stack even if Δ' = 1; this handles the case where there is a gap at the top of the stack. Finally, line 14 replaces the referenced block with a gap if it was below the first gap by implementing the "jump" of a gap described above.

3.8 **Periodic Write-Back.**

With large caches, there may be a very long delay before a block is removed by replacement. In practice, reliability considerations may dictate that a dirty block be written before this time. Suppose that all dirty blocks are written every n seconds instead. An example of this is the UNIX™ file system policy of writing all dirty file system buffers to disk every 30 seconds. Alternatively, suppose only certain blocks are written, for example, by a policy to write a block after it has been unreferenced for n seconds. These policies are all stack algorithms, provided that the write happens for all memory sizes where the block is dirty, in order to maintain inclusion in the dirty set.
A forced write-back is implemented in the algorithm by setting dl (x) to infinity for each written block. It has no effect on writes avoided, except that the write which made the block dirty cannot subsequently be avoided. The effect of this is to increase the calculated number of dirty pushes. Consider the third term in (3.1) for any C where the block is dirty: The block was dirty and included in $D_t(C)$; it is now clean and not in the term; the net increase to dp(C) is 1.

3.9 **Trace-Driven Simulation for Write-Back Caches.**

3.9.1 **one-pass Trace-Driven Simulation Algorithm for Write-Back Caches.**

In this section we present a one-pass algorithm which allows capturing the number of write-backs, in addition to hit ratios, for set-associative write-back caches. We first present briefly the original method for stack simulation and the dirty level concept introduced by Thompson and Smith (1989). To explain the stack algorithm in some detail, consider the example given by Mattson et al. (1970). A reference to a block at time t, denoted x_t, is compared with elements in the LRU stack, denoted $S_{(t-1)}$, that holds the reference history. The number of right match bits is recorded. These right match bits are used to calculate the stack distance i.e. the number of elements that are ahead of the currently referenced element plus 1. An element with i right match bits is in the same set as x_t for caches with 2^t or smaller number of sets. For example, the first element in the stack has 2 right match bits. This implies that this element is in the same set as x_t for fully associative caches, caches with 2 sets and caches with 4 sets. That is, this element is "ahead" of x_t for caches with 4 or smaller number of sets. For the ease of keeping track of the stack distance, an array of right match counters, denoted u(r), is used to record the number of times that exactly bits are right matched. This recording process concludes when x_t is found or the end of the stack is reached (which gives an infinite stack distance). In our example, x_t is found at the seventh position in the stack and the rightmatch- bits frequency counts, u(r), are recorded (see the top right chart of Figure 13.13). From u(r) the stack distances for caches with a different number of sets can be easily calculated as follows. For caches with 2^t sets $0 <= t <=$ maxset, where maxset = 6 in the example, the stack distance of x_t $\sum_{k=1}^{k=6} u(k)$.

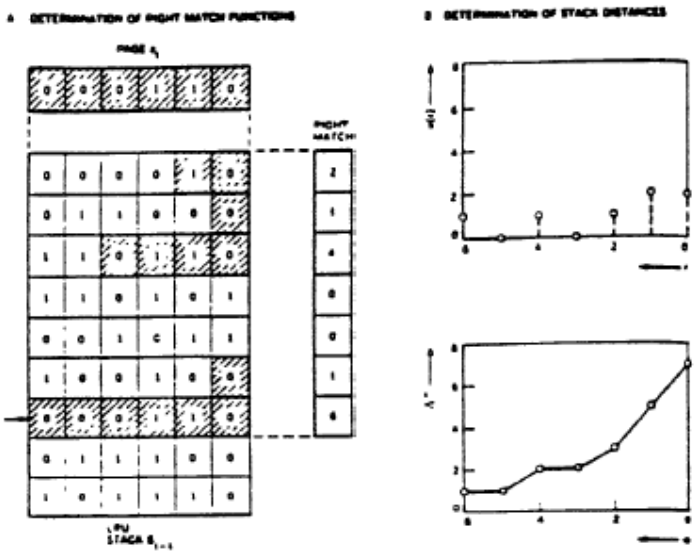


Figure 3.13 an example of Mattson (1970) algorithm

For example, the stack distance for a cache of 16 (2^4) sets is u(6) + u(5) + u(4) = 2. This stems from the fact that if an element is ahead of x_t in the same set for caches with a given number of sets, it will also be ahead of x_t in the same set for caches with a smaller number of sets.

This method permits the computation of hit ratios but it must be extended if other metrics of interest, e.g., number of writebacks, are to be determined. Thompson and Smith (1989) solved this problem in the context of fully associative caches. Their method is to attach a dirty level (*dl*) to each block in the stack. A block is dirty for caches larger than or equal to dl blocks and is clean for those smaller than dl blocks. In other words, dl is the maximum distance A where a dirty block was pushed between two write references. For a given dirty block, dl is updated only when it is referenced. Thus, instead of directly simulating the replacements and recording the number of dirty blocks being displaced, another measure is introduced, namely wa(C), the writes avoided at level C, that is, the number of writes for which the referenced block was still dirty in memory sizes c and larger. The wa(C) count is incremented when a write is performed on a block with a C dirty level. Another counter, wc is used to record the total number of write requests. At the end of the simulation, the number of dirty block replacements for a cache with size C can be obtained by subtracting the number of write requests (wc) by the number of writes being avoided at levels 1 through C.

However, a single dirty level is not sufficient to capture the complete dirty information in simulating set-associative caches. For example, Figure 3.14 shows a block (block 4) which has a dirty level of 3 in a cache with only one

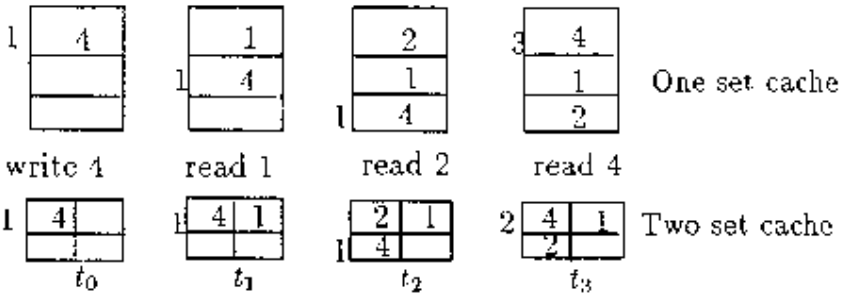


Figure3.14. A block can have different dirty levels in caches with different numbers of sets; in this example, block 4 has dirty level 3 for a one-set cache, but a dirty level of 2 for a two-set cache.

set (i. e., fully associative cache), but has the dirty level of 2 in a cache with 2 sets (one for the even blocks and the other for the odd blocks). In order to simulate caches with a different number of sets using a single stack as in Mattson's (1970) algorithm, we need to attach a vector of dirty levels to each block in which each element of the vector corresponds to the dirty level of the block for caches with a specific number of sets. For example, block 4 in Figure 3.14 will have a (3, 2) dirty-level vector.

We can now present the outline of the all-associativity write-back simulation algorithm in Figure 3.15. The notations used are listed in Table V. In most practical situations we will not simulate all possible set-associativi - ties. Generally, set-associativities will range from one-way (direct-mapped) to a maximum of 16-way. Consequently, this limits the number of sets from maxset = cache size/block size to minset = cache size /(block size x maximum set-associativity). Hence, we will divide the LRU stack into minset substacks to optimize the search. Let S_t^a: denote a substack at the end of time t. With each block in the stack we attach a vector of dirty levels denoted dl[set, block]. Furthermore, write-avoid counts, denoted wa[set, level], are used to record the number of writes that are avoided. Finally, RDistancecnt[set, l:maxassoc] and WDistancecnt[set, 1 :maxassoc] are used to record the histogram of stack distances for read and write accesses respectively. For example, a RDistancecnt[4set] of [10,5,0,0] (i.e., RDistancecnt[4set, 1] = 10, RDistancecnt[4set, 2] = 5, RDistancecnt[4set, 3] = O, and RDistancecnt[4set, 4] = O) records that 10 read accesses have been satisfied with a stack distance of 1 and 5 with a stack distance of 2, That is, if the total number of references were 18 (say three references were not found in the stack) when this state is reached, then the read hit ratio for a 4-set direct-mapped cache is 10/18 – 56 percent and for a 4-set two-way cache, (10 + 5)/18 = 84 percent.

Our algorithm works as follows. For each reference x_t, find the substack a to which it is mapped. A search is done through the substack and the number of right match bits (b) is determined. This number, b, if not larger than log(maxset), is then added to the right match histogram U[b] for producing later the appropriate stack distances. If b is at least as large as log(mcmset), then u[log(maxset)] gets incremented instead.

```
For 1≤ t ≤ N DO
    If write then wc++;
    a = Mod(x_t, minset);
    " Clear u[j] for all j "
    For (i=0; x_t != S^a_{t-1}(i) or (not end of stack); i++)
        b = NumberOfRightBitsMatched(S^a_{t-1}(i), x_t)
        u[max(b, log(maxset))] + +;
    If (end of stack)
        "set all distances infinity"
        "if a write set all dirty levels 1"
        "if a read set all dirty levels infinity"
    Else
        Δ = 0;
        For (k = log(maxset); k ≤ log(minset); k - -)
            Δ = u[k] + Δ
            If dl[k, x_t] < Δ then dl[k, x_t] = Δ
            Case Read:
                    RDistancecnt[k, Δ]++;
            Case Write:
                    WDistancecnt[k, Δ] + +;
                    if dl[k, x_t]!= ∞
                        wa[k, dl[k, x_t]]++;
                    dl[k, x_t]= 1;
    "Establish new stack"
"Gather statistics"
```

Figure 3.15 an outline of an algorithm for simulating write-back set-associative caches.

Table V. Notations for the A1l-Associativity Write-Back Cache Simulation Algorithm

minset	minimum number of sets under study
maxset	maximum number of sets under study
maxassoc	maximum number of associativities under study
S_t^a	substack a at the end of time t, $0 \leq a \leq minset - 1$
Δ	stack distance
wc	write counts
u[*log*(*minset*) : *log*(*maxset*)]	right match counters
RDistancecnt[set,1:maxassoc]	data read distance count
WDistancecnt[set,1:maxassoc]	data write distance count
dl[set,x_t]	dirty level for x_t
wa[set,1:maxassoc]	write avoid counts

If the requested block cannot be found in the stack, all stack distances are set to infinity and the block is brought in. The dirty levels of the block are set to 1 if the request is a write, otherwise they are set to infinity.

If the block is in the stack, the stack distances are determined by accumulating the right match histogram starting from log(maxset) and down through each smaller number of sets, until log(minset) is reached. For each number of sets, the dirty level is set to the current stack distances unless it is already larger than the stack distance in which case it is not modified. If the

reference is a read, the read distance counts are updated. If the reference is a write, the write distance counts are updated and the write avoid counts of the current dirty level are incremented. Finally, the stack is rearranged by moving the referenced block to the top and shifting those top elements down. At the end of simulation, the stack distance counts and the write avoid counts are used to calculate the hit ratios and the number of dirty blocks being replaced.

Figure 3.16 gives a snapshot example to show how a read request is processed under this algorithm. For simplicity, let us assume that we want to simulate caches with 4 sets and 8 sets only, with a maximum set-associativity of 4 (i.e., we want to obtain performance figures for 6 points in the design space, namely 4-set direct-mapped, 4-set 2-way, 4-set 4-way, 8-set direct-mapped, 8-set 2-way, and 8-set 4-way). That is, the minset is 4 and the maxset is 8 for this example. Let us further assume that the contents of the stack are the same as in Figure 13.13. Then we can partition the stack into 4 (i.e., minset) substacks as in Figure 13.16, assuming that the rightmost bits of a block number determine the set number. With relevant states at the end of time t – 1 as given in Figure 13.16a, we can now process the read request on block 6 (000110) as follows. The requested block (block 6) is first tested to determine which substack it might be in and a search is done in that substack only (the leftmost in the figure, the others will remain unchanged). The first block in the stack, block 2, has 2 right bits in common with block 6, the requested block. So block 2 will be in the same set as block 6 for caches with 4 or a smaller number of sets (u[2] is incremented by 1). The next block in the stack, block 54, has 4 right bits in common with block 6, so it will be in the same set as block 6 for caches with up to 16 sets i.e. in the example, for both 4-set and 8-set caches (u[3] = u[log(maxset)] is incremented by 1). Finally the block is found when the third block in the stack is reached (u[3] is incremented by 1). Thus, for an 8-set cache the distance of block 6 is 2 (i.e. u[3]= 2; that is, only block 54 is ahead of block 6) and for 4-set, the distance is 3 (i.e., u[3] + u[2] = 2 + 1 = 3; that is, 2 and 54 are ahead of block 6 in the stack for a 4-set cache).

Assume that at the end of time t – 1 the read distance counts, RDistancecnt, for 4-set caches are [10,5,2, 0]. Then the RDistancecnt will be [10,5,3, 0] at time t,reflecting the fact that block 6 is found at a distance of 3. SimiIarly, for 8-set caches the RDistancecnt is updated to [12,5, 1,0] from [12,4, 1, O]. Moreover, since the stack distances of block 6 are larger than its dirty-levels, the dirty-levels of block 6 far 4-set and 8-set caches are updated to 3 and 2 respectively. Lastly, the substack is rearranged by moving block 6 to the top of the substack. As a final remark, if

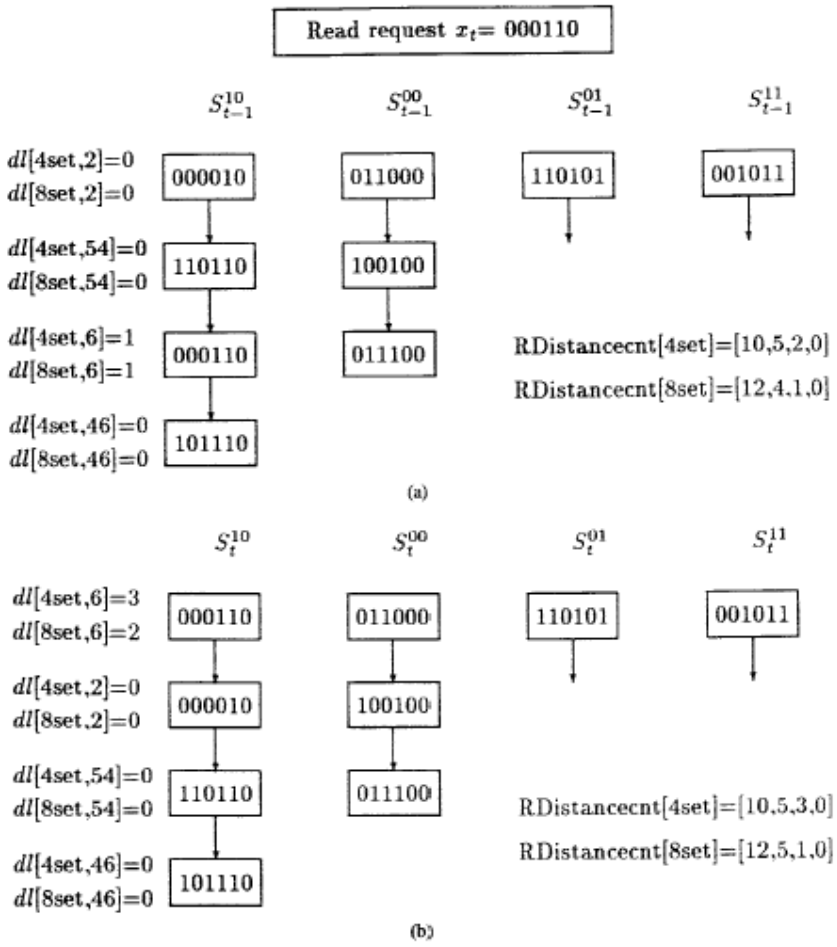


Figure 3.16 a snapshot example for the one-passwrite-back algorithm. (a) before and (b) after the read request being serviced.

the request is a write instead of a read, we need to update the write distance counts and the dirty levels in a similar manner. Furthermore, we also need to increment the write-avoid counts *wa*[4set, 3] and *wa*[8set, 2] to reflect the fact that for a 4-set cache with a set-associativity of at least 3 or for a 8-set cache with a set-associativity of at least 2, the write is to a dirty block, and therefore a write-back to memory is saved.

We used this algorithm to simulate 20 caches in a single pass. All caches have a 4-byte block size and their sizes range from 8-K-byte to 128-K-byte with set-associativities from 1 to 4. The run times for the same traces used in

Table VI Speed-Up Using One-Pass Approach

Table VI Speed-Up Using One-Pass Approach

traces	Pero	Thor	Pops
naive sim. time (orig.)	208.7 sec	246.6 sec	268.3 sec
one-pass sim. time (orig.)	329.6 sec	401.8 sec	393.9 sec
one-pass/naive	1.58	1.63	1.46
one-pass sim. time (redu.)	19.4 sec	36.5 sec	50.1 sec
naive sim. time (orig. 20 caches)	2169 sec	2590 sec	2761 sec
speed-up (one-pass on redu.)	111.8	70.9	55.1

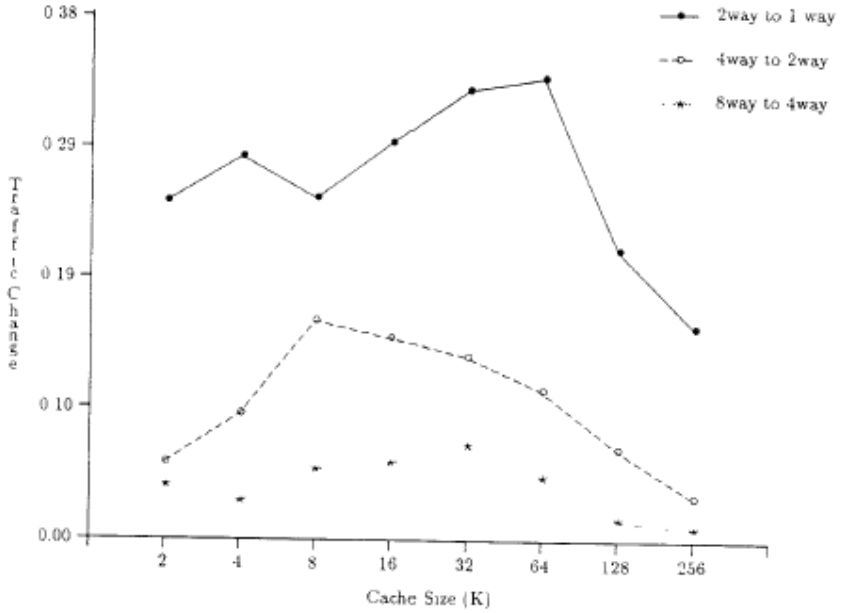


Figure 3.17 Relative traffic change vs. cache size (8-byte block size)

the previous section are given in Table VI. Table VI (line 3) shows that this one-pass simulation is about 1.5 times slower than the one cache per simulation approach (line 1), but can generate 20 results at a time. From Table VI we also observe that if the reduced traces are used together with this one-pass algorithm, the performance of 20 caches can be obtained in less than a minute of VAX 8550 CPU time (line 4). On the other hand, the naive one-result-per-simulation approach (line 5), if used without taking advantage of trace reduction, would need between 50 and 100 times longer to produce the same results. Thus, roughly speaking, trace reduction could bring us a ten-fold disk space saving (see Table II) and, together with the one-pass

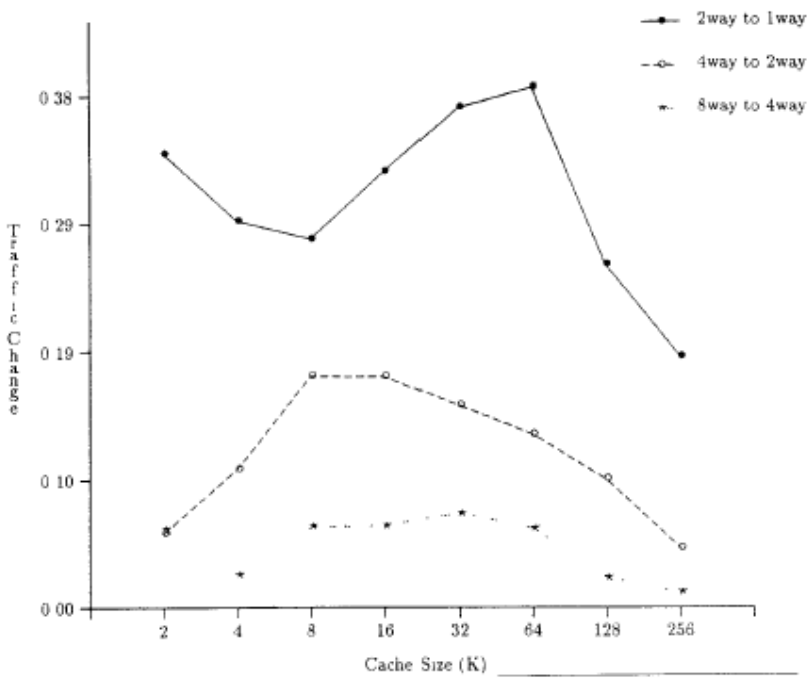


Figure 3.18.Relative traffic change vs. cache size (16-byte block size).

simulation algorithm, could reduce the simulation time by nearly two orders of magnitude over the naive one-result-per-simulation approach.

3.9.2 **An Application Example.**

we now give an example to show how the above techniques can be used to produce useful results on cache performance that have not been covered in the vast cache literature. More specifically, we want to explore the impact of set-associativity on cache-bus traffic. This problem is interesting because it is the cache-bus traffic that will limit the number of processing elements (i.e., CPU + cache) that can be put on a single sharedbus. For the following discussion we define the cache-bus traffic as the number of misses plus the number of write-backs.

The techniques given in this section allow us to explore the design space of write-back caches quickly. Using these techniques and seven Atum-2 traces(see Table I for summary of the characteristics of these traces) , we report in Figures 3.17, 3.18 and 3.19 the impact of set-associativit y on the relative cache-bus traffic change over a wide range of cache sizes, from 2 K-byte to 256 K-byte. In these figures, the X axis represents cache sizes and the Y axis

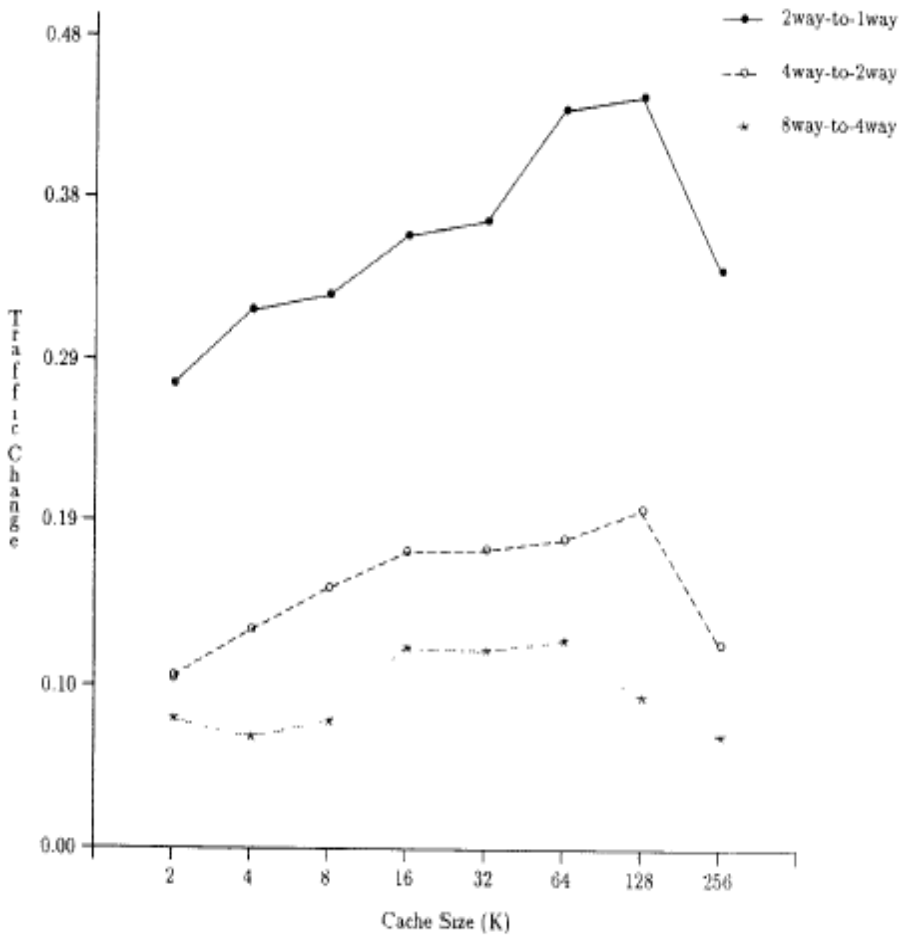


Figure 3.19 Relative traffic change vs. cache size (32-byte block size)

represent the relative traffic increases when we move from a larger set-associativity to a smaller set-associativity. (For the reader's reference, Figure 3.20 shows the write-back traffic percentage for a 16-byte-block size.)

From these figures, we see a big increase (on average 31 percent) in traffic when we move from a 2-way set-associativity to a direct-mapped organization. The average relative traffic changes from 4-way to 2-way and from 8-way to 4-way are 12 percent and 6 percent respectively. We also notice that the relative traffic change increases as the block size increases, especially for small set-associativities. This is due to the fact that the larger the block size the lesser the number of sets, and therefore the conflicts due to set collisions will be more frequent. Similar observations on the relative miss ratio change were reported by Hill and Smith (1989).

To complete this section, we present in Figure 3.20 the percentage contribution of write-back traffic to the total cache-bus traffic. We observe from Figure 3.20 that write-back traffic accounts for 15 to 22 percent of total

bus-cache traffic. Furthermore, the larger the cache, the higher the write-back traffic contribution. This is because a block in a larger cache tends to stay in

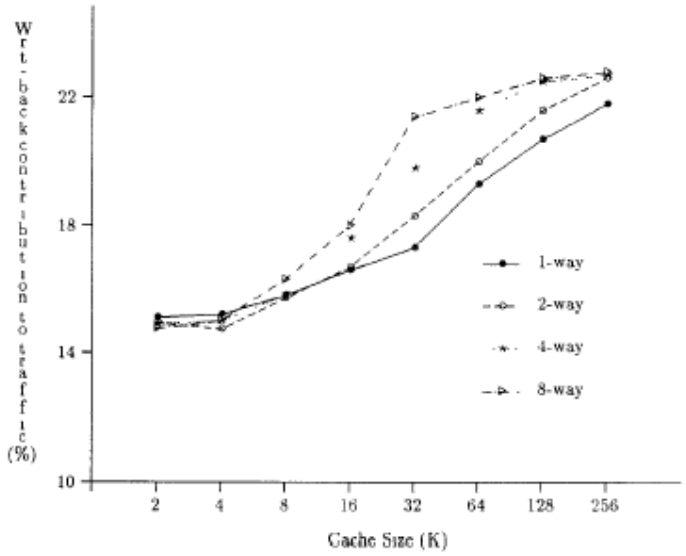


Figure 3.20 Percentage contribution of write-back to total traffic.

the cache longer and therefore has a higher probability of being modified. As a result, upon a miss, the likelihood of having to write back the replaced block is higher for a larger cache. For the same reason, the increase in the set-associativity enlarges the contribution of write-back traffic to the overall cache-bus traffic.

3.10 **Other Stack Simulation Implementation.**

Bennett and Kruskal (1975) examine the paging behaviour of a large data base. They find mean stack distances of 1 to 328 entries for varying page sizes. Bennett and Kruskal propose an algorithm for stack simulation using an m-ary tree and a hash table where the run time per reference is approximately logarithmic in the number of block since the last reference to the current block. In contrast, the time per reference for linked list stack simulation is linear in the number of distinct blocks since the last reference of the current block. Bennett and Kruskal conclude that their algorithm is of order ten time faster than linked list stack simulation for mean stack distances of 150 entries. The storage requirements of the algorithm are large, but this is not important since the memory required is small relative to modern main memory sizes. The tree size is linear in the length of the address trace, N, and the hash table must be larger than the number of distinct blocks (Nδ∞). A simulation of 10 million references with 200,000 unique blocks requires only 3M bytes of storage if it uses two bits per reference and two words per unique block. Olken (1981) changes Bennett and Kruskal's algorithm by replacing their m-ary tree with an AVL tree.

Bennett and Kruskal's algorithm and Olken's algorithm use a hash table to learn about a block's history. A hash table can also be used in linked list stack

simulation to see if a block has ever been referenced. This reduces the time to process a previously unreferenced block from *Kmax* to a constant, reducing simulation time to:

$$\sum_{k=1}^{kmax} k\,\delta_k + kmax^* \sum_{k=kmax+1}^{\infty} \delta_k + O(\delta_\infty) + O(1).$$

This change will significantly improve performance only if both kmax and are large, that is both the degree of associatively and the fraction of the references to previously un-referenced blocks are large.

Thompson et al. (1986) examine each of these algorithms, and conclude that linked list stack simulation performs best for most CPU cache simulations. Consequently, we will compare the performance of forest and all associativity simulation with linked list stack simulation only, and use stack simulation to linked list stack simulation.

3.10.1 **Inclusion in Set Associative Caches .**

Hill(1987) proves several theorems about inclusion for set associative caches using (possibly) differing set mapping functions. Recall that Mattson et al, (1970) discuss inclusion only in caches that use the same set mapping function, and hence have the same number of sets (e.g. all are fully associative). In this section, as in the rest of this chapter, he assume that all caches have the same block size, do no pre-fetching, and use LRU replacement. He wants to use inclusion to rapidly simulate alternative single level cache designs. Consequently when he discusses a large and small cache, he is considering using one or the other in a memory system, not using both as components in cache hierarchy.

Consider two caches, C_1(A=n_1, F=f_2) and C (A=n_2, f=f_2), with blocks, associativities of n_i and set –mapping functions f_i for i=1,2 . An important condition necessary for cache C_2 to include (the blocks of) cache C_1 is that all blocks mapping to the same set in C_2 map to the same set in C_1. That is, for all blocks x and y:

Hill (1987) calls this condition set hierarchy, because it means that f_2 induces a finer partition on all blocks than does f_1. Assume also that each set mapping function maps a large number of blocks(>=2* max(n_1 , n_2))to each set. Set mapping functions used in real caches, including bit selection, trivially meet this restriction.

For cache C_2 to include cache C_1, C_2 must be at least as large as C_1 otherwise inclusion will be violated as soon as C_1, is full. For cache C_2 to include a different cache C_1, C_2 must be strictly larger than C_1. Hill(1987) considers two caches to be equivalent if they always contain the same blocks, i.e. ,are identical up to placement of sets. Suppose cache C_1 and C_2 are of the same size. For cache C_2 to include cache C_1 . It must always contain a superset of cache C_1blocks. Since cache C_2 contains the same number of blocks as C_1. It

must always contain exactly the same blocks, and therefore is not a different cache. For this reason he sometimes refer to cache C as the larger cache.

Theorem 3.1

Cache C_2 ($A=n_2$ $F=f_2$) includes cache C_1 ($A=n_1$ $F=f_1$) if and only if $f_2(x) = f_2(y)$ implies $f_1(x) = f_1(y)$ (set hierarchy) and $n_2 >= n_1$, (non-decreasing associativity).

Proof

➔Suppose cache C_2 includes cache C_1 and $f_2(x_1) = f_2(x_2) = = \ldots = f_2(x_{2n1})$ for some $2n_2$ blocks $x_1 \ldots x_{2n2}$.The x_j , exist, because Hill(1987) assume each set-mapping function maps a large number of block to each set. To demonstrate that both set hierarchy and non decreasing associativity are necessary for inclusion, He show that one of the x_j 's must be in cache C_1 but not in larger cache C_2 if either (1) set hierarchy is false or (2) set hierarchy holds, but the larger cache has the smaller associatively.

(1) With set hierarchy false, let the $2n_2$ x_j s be chosen so that at least one block, y, maps to a different set in cache C_1, than does x_1, (i,e, f_1, (y)!=f_1 (x_1). Either (a) less than n_2 of the x_j 's map to $f_1(x_1)$ or (b) n_2 or more of the x_j map to $f_1(x_1)$. For (a), reference $x_1 >= n_2$ and the blocks that do not map to f (x) . Inclusion is now violated since x_1 is in cache C_1 but not in larger cache C_2. it is in cache C_1 since all other blocks referenced map to other sets; It is replaced in n_2- way set associative cache C_2 since at least n_2 other blocks mapping to its set are more recently referenced. For (b), references y and the $>= n_2$ blocks that do map to $f_2(x_1)$ Inclusion is now violated since y is in cache C_1 but not in the larger cache C_2.

(2) Since set hierarchy holds and $f_2(x_1) = f_2(x_2) = \ldots = f(x_{n2+1})$, Hill(1987) knows that $f_1(x_1) = f_1(x_2) = \ldots = f_1(x_{n2+1})$. Reference x_1 through x_{n2+1} in succession. Inclusion is now violated since x_1 is in n_1- way set associative cache C_1 ($n_1 > n_2$ implies $n_1 >= n_{2+1)}$ but not in n_2-way set associative cache C_2.

<=. Suppose set hierarchy and $n_2 > n_1$. Initially both caches are empty and inclusion holds, because everything (nothing) in cache C_1 is also in cache C_2. Consider the first time inclusion is violated, i.e some block is in cache C_1 that is not in cache C_2. This can only occur when some block y is replaced from cache C_2. but not from cache C_1. A block y can only be replaced from cache C_2 if n_2 blocks, $x_1$through x_{n2} all mapping to $f_2(y)$, are referenced after it. By set hierarchy, $f(y)=f(x_1)=\ldots=f(x_2)$. Since $n_2 >= n_1$, y must also be replaced in cache C_1.

QED.

Theorem 3.1 states that inclusion holds between two set associative caches only if the two caches obey set hierarchy and not decreasing associativity. In Section 3.9.3 Hill (1987) shows that set hierarchy and non decreasing assoriativity are too restrictive to permit inclusion to hold between many pairs of set associative caches, and then he describes an algorithm for simulating numerous set associative caches does not try to take advantage of inclusion.

Hill(1987) next shows that the includes relation is a partial ordering of the set of set associative caches (will the same block size, that do no perfecting, and use LRU replacement). A partial ordering differs from total ordering (e.g "≤" on the set of real numbers), because some elements may not be comparable (i.e neither C_2 includes C_1 nor C_1 includes C_2). While establishing includes as a partial ordering is mostly of theoretical interest, it does enable transitivity to be used in the proof of Theorem 3.3.

Theorem3. 2

The i*ncludes* relation is a partial ordering of the set of caches.

Proof

Hill(1987) must show that includes is reflexive (C_1includes C_1) antisymmetric (C_2includes C_1 and C_1 includes C_2) implies C=C) and transitive ((C_3 includes C_2 and C_2 includes C_1)) implies C_3 includes C_1).

Reflexive, A cache includes another if it contains a superset of the blocks of the other. Clearly C_1 includes C_1. Since two identical caches always contain the same blocks.

Antisymmetric, Suppose C_2 includes C_1 and C_1 includes C_2. Therefore cache C_2 must always contain a superset of the blocks in cache C_1 and cache C_1 must always contain a superset of the blocks in cache C_2. Since superset is antisymmetric, both caches must always contain the same blocks, and therefore are equivalent.

Transitive. Suppose C_3 ($A=n_3$, $F=f_3$) includes C_2 ($A=n_2$, $F=f_2$) includes C_2 ($A=n_2$,$F=f_2$) By Theorem 1, $n_3 >= n_{2,n2} > n_{1,} f_3(x) = f_3(y)$ implies $f_2(x) = f_2(y)$ and $f_2(x) = f_2(y)$ implies $f_1(x) = f_1(y)$, for all block x and y. Since both relations ">=" and implies are transitive $n_3 >= n_1$ $f_3(x) = f_3(y)$ implies $f_1(x) = f_1(y)$. By Theorem 1 C_3 includes C_1.

QED

Next Hill(1987) considers caches using set mapping functions of the form "h (x) rem s," where h (x) is a hash function whose image if the set of all block numbers, "rem" is the remainder operator, and s is the number of sets in a cache, I show that set hierarchy holds between two such caches if and only if the number of sets in the larger cache is a multiple of the number of sets in the smaller cache.

Theorem 3.3

Set hierarchy holds, that is $f_2(x) = f_2(y)$ implies $f_1(x) = f_1(y)$, for set mapping functions of the form (x) rem s_i if and only if s_1 divides s_2.

Proof

=> Suppose set hierarchy holds, that is, h(x) rem s_2 (y) rem s implies h (x) rem s_1h(y) rem s_1. Suppose s_1 does not divide s_2 then s_2 rem s_1 =k where k!=0. Let h(x) = s1, s_2 and h (y) =s_2 (s_1+1). Hill(1987) knows that there exist some block numbers x and y for which the above is true, because he requires the image of hash function h be the set of all block numbers. For these values of h (x) and h (y), h (x) rem s_2 =h (y) rem s_2 = 0 but h(x) rem s_1 = 0 and h (y) rem s_1 = s_2 rem s_1 = k where k != 0. Thus, h(x) rem s $_2$= h (y) rem s_2 is true while h(x) rem s_1 = h (y) rem s_1 is not. A contradiction, There fore, s_1 must divide s_2 for set hierarchy to hold.

<= Suppose s_1 divides s_2. By definition of divides, $s_2 = ns_1$ for some integer n. If h(x) rem s_2 = h(y) rem s_2 , then h(x) =x' s_2 + k and h (y) =y's_2+K for some integers x' ,y' and k. Substitution yields h(x) = x'ns_1 + k and h(y) = y'ns_1 + k. By definition of remainder,h(x) rem s_1 = h(y) rem s_1 = k. Thus h(x) rem s_2 = h (y) rem s_1 implies h (x) rem s_1 = h (y) rem s_1 or set hierarchy holds.

QED.

Theorem 3.3 allows us to prove that inclusion holds for many practical direct mapped caches C_i, including those using bit selection, Consider a series of direct mapped caches c_i , where each cache uses set mapping function f_i (x) = h (x) rem c_i and each c_{i+1} divides C. By theorem 3.3 set hierarchy holds between each pair of caches. Since set hierarchy holds and all associativities are equal (to one). Inclusion holds between each pair of caches by Theorem I. Since inclusion is a partial ordering (Theorem 3.2), inclusion holds between all caches in the series, The above applies to series of direct mapped cache that use bit selection, because for such caches h (x)=x and each c_i divides c_{i+1}, because both are powers of two. Consequently inclusion holds between direct mapped caches that use bit selection.

Since inclusion holds for many direct mapped caches and inclusion can be used to make simulations run more rapidly, Hill (1987) develops an algorithm for simulating direct mapped caches that obey inclusion, which is presented in the next section.

3.10.2 **Simulating Direct Mapped Caches with Inclusion.**

The section introduces forest simulation for evaluating direct mapped caches that have the same block size and obey inclusion. Like stack simulation, forest simulation takes advantage of inclusion by searching for a block from the smallest to largest cache. When a block is found, a hit can be implicitly recorded in all larger caches. Forest simulation is so named because it uses a forest (a set of disjoint trees) rather than a stack to store cache blocks.

Let the direct mapped caches be named C_1 C_2… C_L. Assume that each cache C has c block frames and uses set mapping function rem c. While forest simulation works for arbitrary set mapping functions of the form rem c. Let 1 $\leq$ c < c < …<c and c for I=1,L-1. By the argument presented after Theorem 3, inclusion holds for these caches.

The key data structure in forest simulation is a forest of L levels. The number of trees in the forest is equal to the number of blocks in the smallest cache, c. The c nodes of level I represent the blocks in cache c_1. The branching factor between two levels is equal to the cache size of the larger level, divided by the cache size of the smaller level c_{i+1} / c_i . The leaves represent the blocks in the largest cache, c. This forest can be implemented as a heap containing twice as many nodes as there are blocks in the largest cache, since c_{i+1} / c_i <= 2 for all I implies $\sum_{i=1}^{L} c_i$ is less than 2* c_L . For example, the heap location of block x

a cache of c blocks using set mapping function f can be calculated with *f (x) +* c. Figure 3.20 shows an example forest simulation forest.

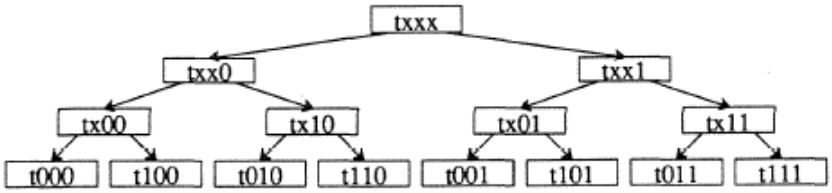


Figure 3.21 Forest Simulations. This figure displays the forest for caches of size 1, 2, 4 and 8 blocks. This forest contains only one tree, because the smallest cache contains only one block. This tree is a binary tree, because each cache in this example is twice as large as the next smaller cache. In this example we assume blocks are mapped to block frames with bit selection. Each node holds the information for one block frame in a direct mapped cache. The block at the root of the tree has no block number bits constrained, because a one block direct mapped cache can hold any block. This illustrated with a t representing arbitrary high order bits of the block number and three x's representing don't cares for the three low order bits. The tags txx0 and txx1 in the nodes of level two indicate that the blocks that can reside in these nodes are constrained to have even and odd block numbers, respectively,. Similar rules with more bits constrained apply to the rest of the levels.

Forest simulation works as follows and as is illustrated in Figure 3.21. On each reference, the algorithm selects the tree corresponding to the set of the reference in the smallest cache. Then it searches.

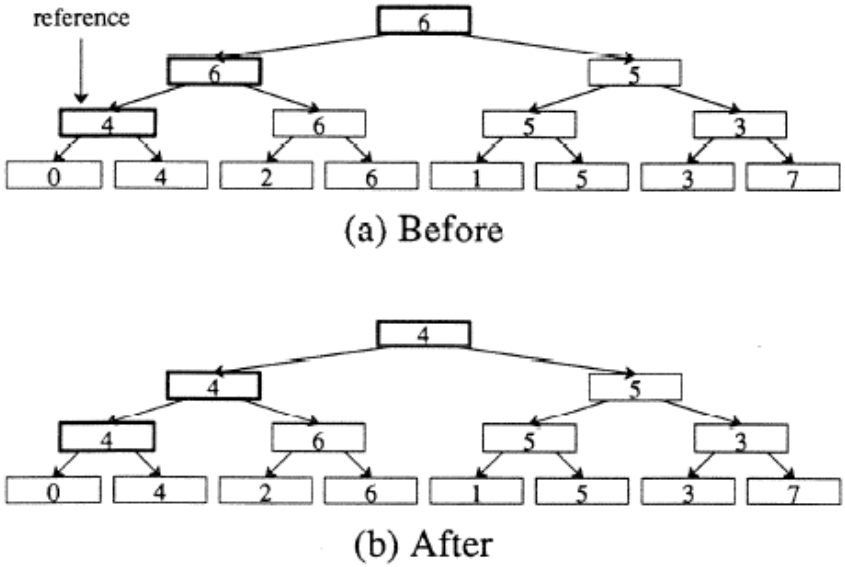


Figure 3.22 Forest Simulation Example .The top tree (a) depicts the forest of Figure 3.20 after a series of references. Information in the tree tells us that block 6 is in a cache of size one block: blocks 6 and 5 are in a direct mapped cache of size two: blocks 4.6,5, and 3 are in a direct mapped cache of size four: and blocks 0 through 7 are in a direct mapped cache of size eight.
Let the next reference be to block 4. A path from the root to a leaf is determined using the set mapping function for each cache (here bit selection is assumed). A search begins at the root and stops when block 4 is found. All nodes encountered in the search that do not contain block 4 are modified to do so. The blocks on bold are examined to find block 4. Since block 4 is located at level 3, caches 1 and 2 miss and caches 3 and 4 hit.
The bottom tree (b) shows the tree after this reference has been processed. The nodes in bold now contain the referenced block.

```
integer L -- number of direct-mapped caches

-- set-mapping functions that obey set hierarchy.
-- i.e., f_{i+1}(x)==f_{i+1}(y) ==> f_i(x)==f_i(y) for i=1, L-1.

function f_1(x), ..., f_L(x)

integer c_1, ..., c_L -- cache sizes (in blocks)

integer N -- number of references

-- distance counts so that m(C(F=f_i, A=1) = 1 - sum_{j=1}^{i} distance[i]/N
integer distance[1:L]

integer forest[1:2*c_L] -- the forest
define block(x, i) = ( f_i(x) + c_i ) -- for accessing the forest
```

Figure 3.23 Forest Simulation Storage

```
For each reference x {
        read(var x)
        N++

        -- FIND
        found – FALSE
        for i=1 to L or found {
                y = forest[block(x, i)]

                if (x==y)
                        found = TRUE
                        -- METRIC
                        distance[i]++
                else
                        -- UPDATE
                        forest[block(x, i)] = x
        }
}
```

Figure 3.24 Forest Simulation Storage

For the referenced block beginning at the root of the tree. The path of the search is determined by the set of the reference in each cache. Any time a node is encountered that does not contain the reference, the node is updated to contain it. The processing of a reference stops when the reference is found, or after a leaf node has been modified. If the reference is found at level i, a counter distance [i] must be updated.

Figure 3.23 and 3.24 show the pseudo-code for forest simulation. Forest simulation is efficient because it uses inclusion and direct mapping. It uses inclusion in the same way as stack simulation, i.e., by ending the processing of a reference when it is found in a cache, regardless of how many larger caches are being simulated. Direct mapping implies that a block can reside in only one block frame in a cache, Forest simulation benefits from direct mapping by examining only that one block frame per cache. In contrast, a simulation of set associative caches must often search more than one block frame per cache size of interest.

As with stack simulation, the exact storage required for a forest simulation of CPU caches is small relative to main memory sizes, The storage required is dominated by the size of the forest, which can be implemented in a heap of 2c nodes, where c is the number of blocks in the largest cache simulated (see figure 3.22). The storage required for simulating direct mapped caches with 32 byte blocks of sizes 128 K, 256K, 512 K and 1M byte, for example, is approximately 500 K bytes, given node sizes of four to eight bytes.

Next Hill (1987) shows the time used to process one reference in a forest simulation of direct mapped caches, c_i, is;

$$1 + \sum_{i=1}^{L-1} m_i + O(1)$$

Where m is the miss ratio of cache c and each iteration requires unit time. The power of this analysis is limited, however, because several constant factors are difficult to calibrate.

The time to simulate each reference is determined by how many times the loop is executed for each reference, plus a constant amount of overhead for reading trace addresses. Forest simulation executes one iteration per cache (level in the forest) up to a maximum of L levels. If one iteration requires unit time, the execution time per reference is:

$$1*(1-m_1) + 2*(m_1-m_2) + \cdots + i*(m_{i-1}-m_i) + \cdots + L*(m_{L-1}-m_L) + L*m_L + O(1).$$

Rearranging terms yields

$$1 - m_1 + \sum_{i=2}^{L} i\,(m_{i-1} - m_i) + L m_L + O(1)$$

$$= 1 - m_1 + \sum_{i=2}^{L} i\,(m_{i-1}) - \sum_{i=2}^{L} i\,(m_i) + L m_L + O(1).$$

This equation can be simplified by manipulating the third term $^{L}\sum_{i=2}$ $i(m_{i-1})$, so that the m_{i-1}'s are changed into m_i's . This manipulation changes the index variable from to replaces j.s with I +1s, simplifiles,and changes summation bounds to yield:-

$$\sum_{i=2}^{L} i\,(m_{i-1}) = \sum_{j=2}^{L} j\,(m_{j-1}) = \sum_{i+1=2}^{L} (i+1)(m_{i+1-1}) = \sum_{i=1}^{L-1} (i+1)(m_i)$$

$$= \sum_{i=2}^{L} (i+1)(m_i) + 2m_1 - (L+1)m_L .$$

Substituting this result back the time per reference equation produces:-

$$1 - m_1 + (\sum_{i=2}^{L} (i+1)m_i + 2m_1 - (L+1)m_L) - \sum_{i=2}^{L} i m_i + L m_L + O(1),$$

which reduces to:

$$1 + m_1 + \sum_{i=2}^{L} 1 * m_i - m_L + O(1).$$

Equation 3.4

Readjusting summation limits yields a run time per reference for forest simulation of:

$$1 + \sum_{i=1}^{L-1} m_i + O(1).$$

The miss ratios for ***L*** direct mapped caches can also be computed with ***L*** separate or concurrent stack simulations of individual caches. In separate simulation, cache C_1 is simulated with all references, then cache C_2 is simulated with all references, and so forth, until cache C_L is simulated with all references. In concurrent simulation all caches are simulated at the same time with each reference processed by all the caches before the next reference is processed. Concurrent simulation is faster than separate simulation, but requires more storage. It is faster, because each trace address is read once rather than times. It uses more storage, since blocks for all caches must be simultaneously resident. Since we care about run time and not about storage, we consider only concurrent simulation further.

In concurrent simulation each address is read once, and unit processing is required for each level. The run time per reference is therefore:

$$L + O(1)$$

This time is greater than the time for forest simulation,

$$1 + \sum_{i=1}^{L-1} m_i + O(1)'$$

For practical (not equal to one) miss ratios.

3.10.3 **Simulating Set-Associative Caches without Inclusion.**

Stack and forest simulation will simulate a series of caches with one pass through an address trace. Both methods are "efficient," because they take advantage of inclusion. Since inclusion does not hold for caches of all sizes and associativities (see Theorem 3.1), algorithms using inclusion must constrain the series of caches simulated (see figure 3.24). Hill (1987) describes an algorithm, which he calls all associativity simulation that does not use inclusion, but can simulate set associative caches with the same block size, that do no prefetching, and use LRU replacement, with one pass over an address trace. With it, he can cover the design space of figure 3.25 in 3 simulations (one per block size) instead of 15 runs of stack simulation .The algorithm described here permits the set associative caches use of arbitrary set mapping functions. A literature search revealed that a version of all associativity simulation, where all set mapping functions use bit selection, was developed by researchers at IBM (Mattson(1970) , Trai(1971).

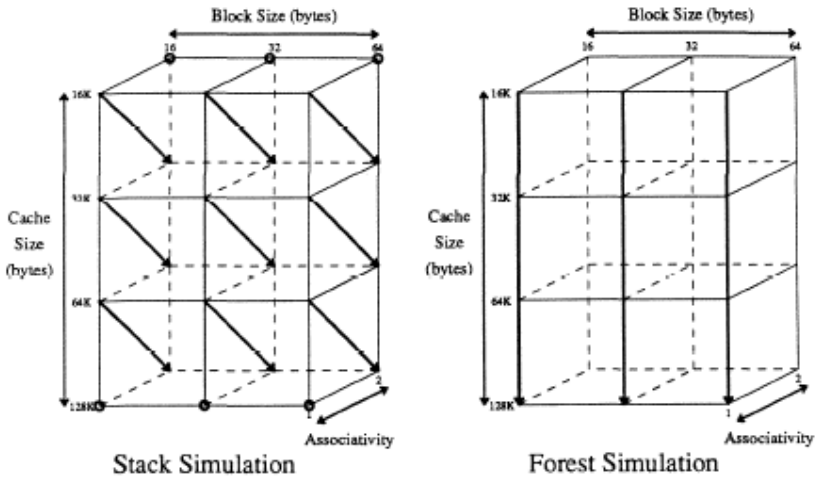


Figure 3.25 Cache design Space. This figure displays a portion of the cache design space: cache size, block size and associativity. The solid lines on the left connects cache designs that can be simulated with a single stack simulation. Circles indicate simulations of single cache designs. The solid lines on the right connect cache designs that can be simulated with single forest simulation.

Covering portions of the cache design space can require many simulations even though stack and forest simulation simulate several caches at a time. In this example,

15 stack simulations are needed, and 3 forest simulation cover half the space but can't simulate the rest of it. Alternatively, three all associativity simulations (one per block size) cover the same space.

All associativity simulation does not take advantage of inclusion, because inclusion does not hold for many groups of set associative caches. For example: (a) direct mapped and two way set associative caches of any size do not include any four way set associative caches, because, the former have smaller associativities; (b) a four way set associative cache of c blocks does not include a direct mapped cache of c/2 blocks even if both use bit selection, because x rem c/4 = y rem c/4 does not imply x rem c/2=y rem c/2 (e.g.=0 and y=c/4); and (c) it is not possible for a cache c_2 to include a different cache c_1 of the same size, because C_2 can never contain any block not in $C_1$and still contain all the blocks of cache C_1.

Hill(1987) now develops all associativity simulation from stack simulation through successive refinements. An all associative simulation run can simulate caches that use different set mapping functions, f_i(x) and have different capacities. The same caches can be simulated with concurrent stack simulations. This approach requires a stack simulation for each different set mapping function. For instance, if the

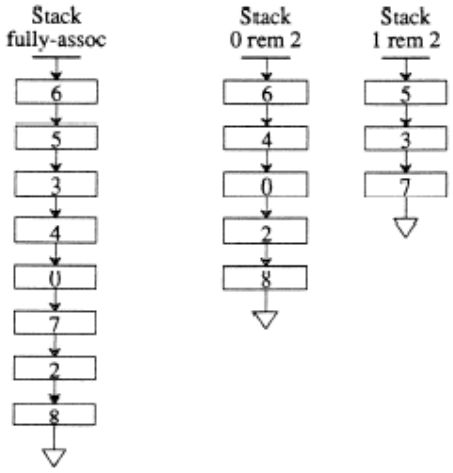


Figure 3.26 Stacks for caches with one or two sets using bit selection. This figure displays how the stacks for caches with one (fully associative) or two sets using bit selection (f (x)=0 and f (x)=x rem 2) could look during a simulation. The stack for one set contains a list of all the block numbers recently referenced, listed from most recently referenced to least recently referenced. we call this stack a fully associative stack, because it models fully associative caches. The stacks for two sets contain similar lists for the even and odd block numbers. 2^L stacks are required to simulate

with bit selection for 2^L sets. A block resides in a cache of C blocks with one set if and only if the block is in the fully associative stack at a distance of less than or equal to c. A block resides in a cache of c blocks with two sets if and only if it is in the appropriate stack at a distance of less than or equal to c/2. A block resides in a cache of c blocks with 2^L sets if and only if it is in the appropriate stack at a distance of less than or equal to $c/2^L$.

---

caches to be simulated use bit selection with 0, 1 and 2 bits, the following stack simulations are sufficient: a stack simulation with one stack for caches with $f_1(x) = 0$, a stack simulation with two stacks for caches with $f_2(x) = x \text{ rem } 2$ and a stack simulation with four stacks for caches with $f_3(x) = x \text{ rem } 4$. Figure 3.26 illustrates how the stacks for one and two sets with bit selection could look during concurrent stack simulation. In this example both sets of stacks contain the same nodes. In practice when stack sizes are bounded by the largest cache size of interest, the sets of stacks will (usually) contain slightly different nodes. For example node 8 would be missing from stacks for two sets if caches of interest are restricted to eight total blocks (four per set). Nevertheless, many blocks will be shared by both caches, since similar caches have similar hit ratios.

Storage can be reduced in this simulation by allocating a single node per block and including in the node a next pointer field for each group of stacks being simulated. Figure 3.27 illustrates how the nodes of the single fully associative stack can be linked with a second set of next pointers to form the stacks for caches with two sets.

While reducing storage is not important, this node sharing holds the key to reducing time. Observe that all the pointers in the stack point down. This is always the case for LRU replacement, because the order of two blocks in any stack is a function of references to those two blocks, independent of all other references. For example, stack 0 rem 2 in Figure 3.26 indicates that block 6 was referenced more recently than block 4. Block 6 must also be above block 4 in the fully associative stack, because intermediate references to blocks 3 and 5 do not effect whether block 6 was referenced more recently than block 4. Since all pointers point down, the fully associative stack contains all the information

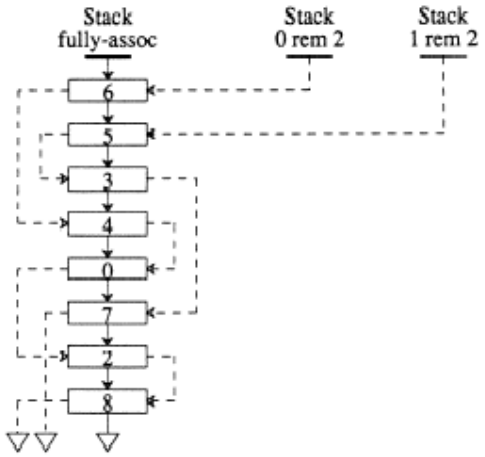


Figure 3.27 Concurrent Stack Simulation with Shared Storage. This figure illustrates a single set of nodes can be used to represent the stacks for caches using bit selection with one and two sets. A second next pointer field must be added to each node so that it can be linked into a second stack.. The stack for stack simulations with L different set mapping functions can share one group of nodes if each node contains storage for L different next pointers. This reduces storage requirements with respect to using separate stacks, but does not reduce simulation time

.

necessary to determine the order of nodes in all other groups of stacks. Thus, the stack for reference x with set-mapping function f (x) can be constructed by finding all blocks y in the fully-associative stack where f (y) =f (x) and listing these blocks in the same order as they are encountered in the fully-associative stack.

The goal of this research is to find stack distances, and hence miss ratios, however, not construct all the groups of stacks. The following algorithm computes stack distances for set-associative caches of different capacities and set-mapping functions f_i through f_L directly from the fully-associative stack.. When simulation completes, each counter distance (i,k) holds the number of references to stack distance k with set-mapping function f_i. For each reference to block number x {

Zero the L total_above counters.
Look at nodes y in the fully-associative stack until x is found or the stack exhausted. If y=x {
Increment the L total_above counters, move x to the top of the stack, and increment distance (i, total_above[i]) for i =1 to L.
} else {
For i=1 to L, increment total_above [i] if f_i (x)=f_i (y).
}

If the stack is exhausted without finding x, push x on the top of the stack.

}.

Figure 3.28 illustrates the algorithm operating on one reference. Figures 3.29 and 3.30 give pseudo-code for this algorithm.

All-associativity simulation can be improved further if I restrict the f_i's so that the set hierarchy condition holds. Recall that this condition is:

$$f_{i+1}(x)=f_{i+1}(y) \text{ implies } f_i(x)=f_i(y).$$

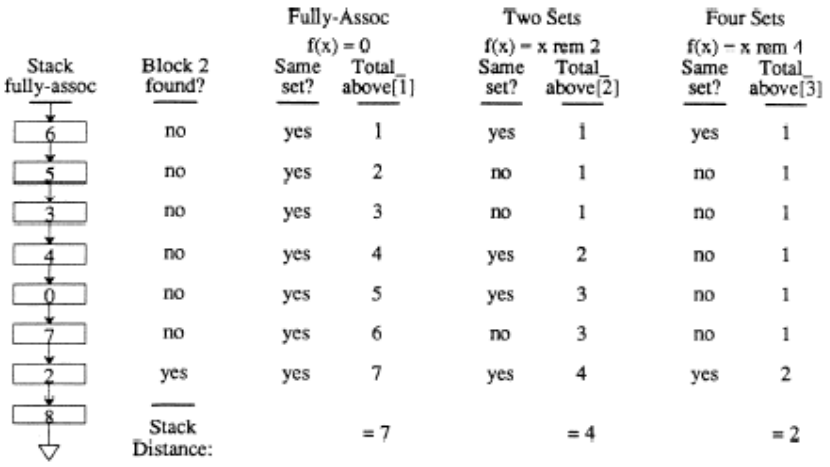


Figure 3.28 All-Associativity Simulation Example. This figure illustrates how all-associativity simulation processes a reference to block 2 for caches with set-mapping functions $f_1(x)=0$, $f_2(x)=x$ rem 2, and f3(x)=x rem 4. Counter total_above (i) always contains the number of blocks encountered so far stack fi(2), since block 2 is referenced. Each row of the figure shows that total_above (i) is incremented in response to block y in the stack if and only if i(y) =fi(2).

Processing stops when the reference is found (block 2). The stack distance of block 2 in a cache with set-mapping function fi is total_above[i]. The stack distances found for block 2 are 7,4, and 2, respectively.

```
integer L -- number of set-mapping functions
function f_1(x), ..., f_L(x) -- arbitrary set-mapping functions
integer N -- number of references
integer max_assoc -- maximum associativity for metrics
-- distance counts so that m(C(F=i, A=k)) = 1 - Σ_{j=1}^{k} distance[i,k]/N
integer distance[1:L, 1:max_assoc]

define stacknode_type {
        integer block_number
        stacknode_type *next
}
stacknode_type *stack -- top of stack pointer
stacknode_type stacknodes[1:O(N*δ_∞)] -- pool of dynamically linked stacknodes
```

Figure3.29. All-Associativity Simulation Storage.

In all-associativity simulation with arbitrary fi ‘s, it is necessary to know which of any two blocks are more recently referenced. Consequently, a total ordering of the previously referenced blocks must be

```
For each reference x {
        integer total_above[1:L]  -- distance counters for x

        for i=1 to L { total_above[i] = 0 }
        read(var x)
        N++

        -- FIND
        found = FALSE
        previous_node_pointer = NULL
        node_pointer = stack
        while ((NOT found) AND (node_pointer==NULL)) {

                y = node_pointer->block_number

                if (x==y) {
                        found = TRUE
                        for i=1 to L { total_above[i]++ }
                }
                else {
                        for i=1 to L {
                                if (f_i(x)==f_i(y))  total_above[i]++
                        }
                        previous_node_pointer = node_pointer
                        node_pointer = node_pointer->next
                }
        }

        -- METRIC
        if (found) {
                for i=1 to L {
                        -- References to distances beyond max_assoc are implicit misses.
                        if (total_above[i] ≤ max_assoc)  distance[i, total_above[i]]++
                }
        }

        -- If was found, move the stack node of x to the top of the stack.
        -- Otherwise, store x in a new stacknode and move it to the top of the stack.

        UPDATE(x, found, previous_node_pointer, node_pointer)
}
```

Figure 3.30 All Associativity Simulation

maintained with a fully-associative LRU stack. Since the set hierarchy condition also implies:

$$f_1(x) != f_1(y) \text{ implies } f_i(x) != f_i(y) \text{ for } i=1,L,$$

two blocks in different f_1-stacks will never be compared. This means all-associativity simulation with set hierarchy need only maintain the LRU stacks for each element in the image of f_1. Simulating with multiple stacks is faster than simulating with one, because the average number of active blocks one must look through to find a block is smaller, since active blocks are spread across many stacks. This reduction is significant since, the number of stacks for practical CPU cache simulations is often greater than 100. The number of stacks used in a simulation of the VAX-11/780's cache, for example, is 512.

Another benefit of set-hierarchy is that a simulation need not examine f_i (x)= f_i(y) for i=L down to 1, since f_{i+1} (x)= f_{i+1} (y) implies f_i (x)= f_i (y). Instead of iterating through all L set-mappings, one can begin with f_L and stop as soon as f_i (x)= f_i (y). For instance, if x and y are in the same set in the largest cache simulated, i.e., f_L (x)= f_L (y) ,the number of iterations is reduced from L to one. Additional time can be saved if one increments above(i) only for the largest i for which x and y map to the same set, rather than incrementing total_above[i] for each i where f_i (x)= f_i (y). When x is found or the stack exhausted.

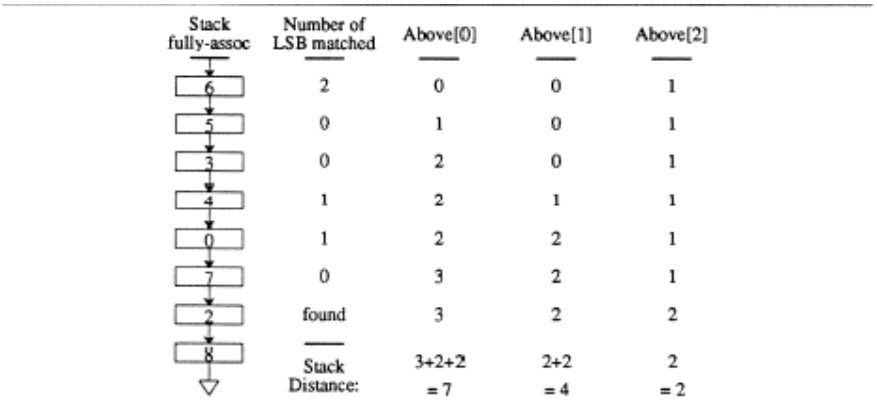


Figure 3.31. All-Associativity Simulation with Set Hierarchy Example. This figure illustrates how all-associativity simulation with set hierarchy processes a reference to block 2 by scanning the stack until block 2 is found (or the stack is exhausted). For each block before the reference is found: (a) The algorithm calculates the largest set-mapping function, f_i, for which the reference and the stack node are in the same set. For bit selection, the calculation reduces to computing the number of least-significant bits that match between the block numbers of the reference and the stack node. (b) it increments above(i). Once the reference is found, above (L) is incremented, the reference's stack distance with set-mapping function fi is $\sum_{k=1}^{L}$ *above [k].*

```
integer L  --  number of set-mapping functions

--  set-mapping functions that obey set hierarchy,
--  i.e., f_{i+1}(x)==f_{i+1}(y) ==> f_i(x)==f_i(y) for i=1, L-1.

function f_1(x), ..., f_L(x)

integer number_of_stacks  --  number of sets induced by f_1(x)

integer N  --  number of references

integer max_assoc  --  maximum associativity for metrics

--  distance counts so that m(C(F=i, A=k))  =  1 - sum_{j=1}^{k} distance[i,k]/N

integer distance[1:L, 1:max_assoc]

define stacknode_type {
        integer block_number
        stacknode_type *next
}

stacknode_type *stack[0:number of stacks-1]  --  top of stack pointers

stacknode_type stacknodes[1:O(N*δ_∞)]  --  pool of dynamically linked stacknodes
```

Figure 3.32. All Associativity Storage w/ Set Hierarchy

```
For each reference x {
        integer above[1:L]  -- distance counters for x

        for i=1 to L { above[i] = 0 }
        read(var x)
        N++
        stack_number = f1(x)

        -- FIND
        found = FALSE
        previous_node_pointer = NULL
        node_pointer = stack[stack_number]

        while ((NOT found) AND (node_pointer==NULL)) {

                y = node_pointer->block_number

                if (x==y) {
                        found = TRUE
                        above[L]++
                }
                else {
                        match = FALSE
                        for i=L down to 1 or match {
                                if (fi(x)==fi(y)) {
                                        match = TRUE
                                        above[i]++
                                }
                        }
                        previous_node_pointer = node_pointer
                        node_pointer = node_pointer->next
                }
        }

        -- METRIC
        if (found) {
                total_above = 0
                for i=L down to 1 {
                        total_above = total_above + above[i]
                        -- References to distances beyond max_assoc are implicit misses.
                        if (total_above ≤ max_assoc)  distance[i, total_above]++
                }
        }

           If was found, move the stack node of x to the top of its stack.
        -- Otherwise, store x in a new stacknode and move it to the top of the stack.

        UPDATE(x, stack_number, found, previous_node_pointer, node_pointer)
}
```

Figure 3.33 All Associativity Simulation w/ Set Hierarchy

$$total\_above\,[i\,] \;=\; \sum_{k=i}^{L} above\,[k\,].$$

Figure 3.31 gives an example of all associativity simulation using set hierarch. Figures 3.32 and 3.33 give the pseudo-code for this improved algorithm.

All-associativity simulation can be made to run even faster in practice if the f_i's are all bit selection. Bit selection set-mapping functions make it easy to compute the largest i for which x and y map to the same set. The computation reduces to finding the minimum of L and the number of least significant bits that match between x and y.

Hill (1987) has defined all-associativity simulation for set-associative caches that use LRU replacement. He now shows that it does not work with two other commonly-implemented replacement algorithms, FIFO and RANDOM. All-associativity simulation does not work with FIFO replacement, because all associativity simulation is based on stack simulation, and FIFO is not a stack algorithm.

Figure 3.34 shows by example that all-associativity simulation does not work with RANDOM replacement even though RANDOM is a stack algorithm. The example illustrates the following general problem. Any replacement algorithm may reorder blocks in the set of a reference between the top of stack and the original position of the reference, so long as no blocks other than the reference move up. In all-associativity simulation, multiple set-mapping functions are concurrently simulated. Therefore, some blocks can be in the set of a reference with one set-mapping function and not in the set of a reference with another set-mapping function. Incorrect behaviour occurs any time blocks not in the set of the reference are reordered. LRU prevents such blocks from being reordered by never changing the order of unreferenced blocks.

While Hill (1987) has shown that all-associativity simulation fails with FIFO and RANDOM, he has not shown that all-associativity simulation fails with all replacement algorithms other than LRU. One way to show this is to prove the following. Consider an all-associativity simulation with two set-mapping functions, $f_1 \; ! = f_2$. (Recall that all-associativity simulation reduces to stack simulation if only one set-mapping function is used). The stacks in all-associativity simulation are updated incorrectly in response to a reference x if blocks not in f_1 (x) or not in f_2 (x) are reordered. While he has not done it, one can demonstrate that LRU replacement is necessary for all-associativity simulation by showing that any replacement algorithm which obeys the above constraint never reorders any unreferenced blocks, and is therefore equivalent to LRU.

The storage for all-associativity simulation is dominated by storage for the stack nodes (see Figure 3.31). Like unbounded stack simulation, the storage required is proportional to the number of unique blocks in a trace, $N\delta_\infty$. Even for a long trace, however, the storage required is small relative to modern main memory sizes. A simulation of 10 million references with 200 thousand unique blocks requires only 1.6M bytes of storage if it uses two words per block. While stacks in stack simulation can be bounded by the largest associativity of interest, stacks in all-associativity simulation cannot be bounded, because these

stacks are used to construct stacks for other set-mapping functions. Consider a fully-associative stack and stacks for even and odd blocks. The fully-associative stack cannot be bounded, because the first odd block can reside at an arbitrary large distance in the fully-associative stack.

The run-time (per reference) of all-associativity simulation with set hierarchy centres around how many times the "while" loop is executed (see Figure 3.32). Let δ_k be the probability that a reference is found at stack depth k, and let δ_∞ be the probability that a reference is not found. References at stack distance k are found in k iterations. References at stack distance ∞ are found by looking through the entire stack. The size of the stack is equal to the number of distinct blocks previously referenced, which is $O(N\delta_\infty)$, where N is the number of blocks in the address trace. On each iteration in all-associativity simulation, a stack node must be compared to the reference to see if they are the same. If not, additional work is required to find the maximum i for which the reference and the stack node are in the same set. Let the average amount of this extra work be called match_compute. Whenever a reference is found at distance k, unit work must be done on k iterations and match_compute work on all but the last iteration. In addition, each reference must be read from a trace file, L above counters initialized and summed to form the stack distances. We gather the per-reference overhead in 0(1). Thus, time to process a reference is of order:

$$\sum_{k=1}^{\infty} \delta_k * [k + (k-1)*match\_compute] +$$
$$\delta_\infty * O(N\delta_\infty) * [1+match\_compute] + O(L) + O(1).$$

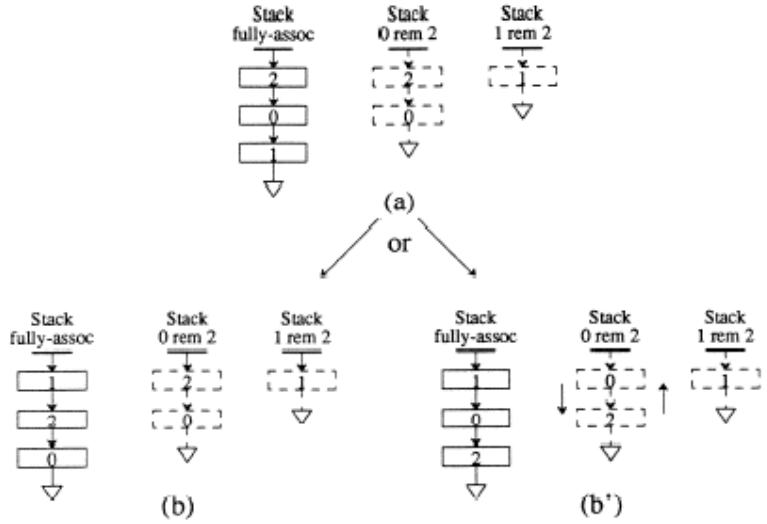


Figure 3.34. Random Replacement Does Not Work. This figure shows that all-associativity simulation does not work with RANDOM replacement. Part(a) illustrates a fully-associative stack after a series of references (left), and the pair of stacks for

even and odd blocks implied by the fully-associative stack (right). Among other things, the stacks imply that a two-block fully-associative cache contains blocks 0 and 2, and a two-block direct-mapped cache contains blocks 1 and 2.

Let block 1 be referenced. RANDOM replacement in the two-block fully associative cache requires that block 0 or block 2 be replaced with equal probability. Part (b) shows block 0 replaced, while part (b') shows block 2 replaced. The stacks in part (b) are consistent, since RANDOM replacement coincidentally replaces the least recently used block..

The state of the fully-associative stack in part (b'), however, implies that block 0 is in the two-block direct-mapped cache. The state of the stacks is inconsistent, since block 0 was not originally in the two-block direct-mapped cache, block 0 was not referenced, and prefetching is not allowed. Therefore the fully-associative stack under RANDOM replacement cannot be used to infer the positions of blocks in the even and odd stacks, which demonstrates that all-associativity simulation does not work with RANDOM replacement.

$$K=1\sum^{\infty}[k +(k-1)* \text{match\_computer}]+$$

$$\delta\infty*(N\infty)*[1+ \text{match\_computer}] + o (L) + o (1)$$

The first term is the time to process previously referenced blocks; the second is for previously unreferenced blocks; the third and final terms are for manipulating counters and reading the reference, respectively.

To see how this run-time compares with stack simulation, let us assume the all set-mapping functions are bit-selection. For this to be possible with a 32-bit address, L must be less than 32. If the low-order bits of block numbers for recently-referenced blocks are independent and equally likely to be zero or one, then the expected number of least-significant bits that match is less than 1 (1/2+1/4+1/8+…). Since the loop computing match iterates until a mismatch is found, the expected number of iterations is two . Substituting two for match_compute yields:

$$\sum_{k=1}^{\infty}\delta_k * [3k - 2] + 3*O(N\delta_\infty)*\delta_\infty + O(32) + O(1),$$

Which should not be more than three times greater than the time for stack simulation.

$$\sum_{k=1}^{\infty} k\delta_k + O(N\delta_\infty)*\delta_\infty + O(1).$$

In practice the relative difference in run times should be smaller, because it is expected that the $O(32)$ term to be small compared to other terms, δ_i to be near one (the direct-mapped hit ratio near one) often saving any match_compute overhead, and the per-reference overhead $O(1)$ to be relatively large.

3.11 **Comparing Actual Simulation Times**

Hill (1987) compares the simulation times of implementations of stack, forest, and all-associativity simulation. While the exact quantitative results of this section do not necessarily apply to other implementations, there is no reason to believe that gross comparisons do not generalize. The advantage of this data over the run-time analysis presented earlier is that these results apply to at least one set of implementations of these algorithms.

He implemented stack, forest and all-associativity simulation in C under UNIX 4.3 BSD. Stack and forest simulation were added to a general cache simulator, called DineroIII (Hill(1987)85). DineroIII originally contained 1250 C statements, as measured by the number of source lines containing a semicolon or closing brace. Adding stack simulation increased total code size by 150 statements, adding forest simulation, 220 statements. Stack simulation is implemented using linked lists and without using a hash table to detect previously unreferenced blocks. The forest simulation implementation restricts the set-mapping functions to be the block number modulo the cache size in blocks, a generalization of bit selection. Hill (1987) implemented all-associativity simulation in a separate program, called Tycho, containing 800 C statements and having far fewer options than DineroIII. Tycho restricts the set-mapping functions to be bit selection.

He estimates simulation time with the elapsed virtual time (user plus system) returned by the UNIX 4.3 BSD system call getrusage on an otherwise unloaded Sun-3/75 with 8M of memory and no local disk. Trace data is read from a file server via an Ethernet. He gives results for four traces from four different architectures, despite finding that results are fairly insensitive to program traces. All caches simulated have 32-bytes blocks, do no prefetching, use LRU replacement, are mixed (data and instruction cached together), and use bit selection.

He begins by verifying that implementations of the three algorithms have similar run-times for simulating a single cache, using two methods. First, he ran each implementation using a trace of 1 million identical addresses so that all references, expect the first, hit at distance one. Results show that the elapsed virtual times of forest and stack simulation differ by 0.1 percent, while all-associativity simulation is 3 percent faster. All-associativity simulation is faster, because it is implemented in a different program, Tycho. It is not surprising that Tycho is slightly faster than DineroIII which is a general cache simulator. Even through DineroIII's additional features are not used in these simulation runs, DineroIII uses some execution time to fall through the if statements that guard the additional features.

Second, Hill (1987) compares the algorithms simulating a 16K-byte direct-mapped cache with each of four traces. A stack and an all associativity of a single 16K-byte four-way set-associative cache are also comparable.

Since Hill (1987) implementations of these algorithms have similar run-times for simulating single caches, the comparisons of multiple cache simulations that follows are meaningful, because we know that simulation time differences are not due to per reference overheads. Therefore we just take the advantage of Hill's research and corresponding results and implement them in own devised algorithms for multiprocessor environments.

3.12 **One-Pass Simulation Technique for Multiprocessor Set-Associative Caches.**

The techniques presented in the previous sections could be further improved if one were to extend the uniprocessor one-pass algorithm in Section 3.8.1 to multiprocessors. In this section we discuss the difficulty of extending this algorithm to multiprocessors with invalidation protocols and describe the feasibility for distributed-write protocols.

For multiprocessors and an invalidation-based coherence protocol, the algorithm in Figure 3.33 requires modifications to handle shared reads and shared writes. A shared read will cause a reset of the requested block's dirty level since the block will be clean after being read by another cache. A shared write will cause a deletion and will leave a "hole" in the stack. This hole cannot be deleted since that will change the stack level of all lower blocks. We call the deleted block a marker. For fully associative cache simulations, a marker will remain at the same position in the stack until another block below the marker is accessed. In this case the marker propa - gates to the position of the newly accessed block while the block is moved to the top of the stack.

For a set-associative cache simulation algorithm , a deletion also leaves a marker in the stack. However, the propagation of a marker is a complicated Matter. the caches with several different numbers of sets, say 2, 4 and 8, are represented by a single stack. Now, when a block below the marker is accessed, it can be, for example, in the same set with the deleted block for caches with 2 and 4 sets but not for 8 sets. Therefore, from the viewpoint of 2 and 4-set caches, the marker needs to be propagated, but for 8-set caches the marker should stay. Thus, a marker can become multiple markers, and keeping track of their propagations complicates the one-pass algorithm.

This problem is not present for multiprocessors using distributed- write protocols. This is because a shared write will update instead of invalidate other copies of the requested block. That is, the effect of a shared write by other processors is to reset the dirty level of the block, as a shared read would do. Thus, the one-pass algorithm can be straightforwardly extended for multiprocessors using distributed-write protocols.

3.13 **Cache Coherence Protocol for Multiprocessor Set-Associative Caches.**

In contemporary multiprocessor systems, it is customary to have one or two level of cache associated with each processor. This organisation is essential to achieve reasonable performance. It does however; create a problem known as the cache

coherence problem. The essence of the problem is this: Multiple copies of the same data can exist in different caches simultaneously, and if processors are allowed to update their own copies freely an inconsistent view of memory can result. The are two common write policies:-

- Write back: Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.
- Write Through: All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

It is clear that a write back policy can result in inconsistency. If two caches contain same line, and the same line is updated in one cache, the other cache will unknowingly have an invalid value; subsequently reads to that invalid line produce invalid results. Even with the write through policy, inconsistency can occur unless other caches monitor the memory traffic or receive some direct notification of the update.

In this section we will briefly survey various approaches to cache coherence problem and then focus on the approach that is most widely used: the MESI protocol.

For any cache coherence protocol, the objective is to let recently used local variables get into the appropriate cache and stay there through numerous reads and writes, while using the protocol to maintain consistency of shared variables that might be in multiple caches at the same time. Cache coherence protocols have generally been divided into software and hardware approaches. Some implementations adopt a strategy that involves both software and hardware elements. Nevertheless, the classification into software and hardware approaches is still instructive and commonly used in surveying cache coherence strategies.

Software Solutions. Software cache coherence schemes attempt to avoid the need for additional hardware circuitry and logic by relying on the compiler and operating system to deal with the problem. Software approaches are attractive the overhead of detecting potential problems is transferred from run time to compile time, and the design complexity is transferred from hardware to software.

- Hardware Solutions. Hardware based solutions are generally referred to as cache coherence protocols. These solutions provide dynamic recognition at run time of potential inconsistency conditions. Because the problem is only dealt with when it actually arises, there is more effective use of caches, leading tom improved performance over a software approach. In addition these approaches are transparent to the programmer and the compiler reducing the software development burden.

  Hardware schemes differ in number of particulars, including where the state information about data lines is held, how that information is organised , where coherence is enforced, and the enforcement

mechanisms. In general hardware schemes can be divided into two categories: directory protocol and snoopy protocol.

- Directory Protocol. Directory protocols collect and maintain information about where copies of lines reside. Typically there is a centralized controller that is the part of the main memory controller, and a directory that is stored in main memory.
- Snoopy Protocol. Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor.

3.13.1 **The MESI Protocol.**

The data cache includes two status bits per tag, so that cache line can be in one of four states:

- *Modified*: The line in the cache has been modified (different from main memory) and is available only in this cache.
- *Exclusive*: The line in the cache is same as that in the main memory and is not present in any other cache.
- *Shared*: The line in the cache is the same as that in main memory and may be present in another cache.
- *Invalid*: The line in the cache does not contain valid data.

Now we discuss different situations of read and write.

- Read Miss. When a read miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. The processor inserts a signal on the bus that alerts all other processor/ cache units to snoop the transaction. There are number of possible outcomes:
  - If one other cache has a clean (unmodified since read from memory) copy of the line in the exclusive state, it returns a signal indicating that it shares this line. The responding processor then transitions the state of its copy from exclusive to shared, and the initiating processor reads the line from main memory and transitions the line in its cache from invalid to shared.
  - If one or more cache have a clean copy of the line in the shared state, each of them signals that it shares the line. The initiating processor reads the line in its cache from invalid to shared.
  - If one other cache has a modified copy of the line, then that cache blocks the memory read and provides the line to the requesting cache

over the shared bus. The responding cache then changes its line from modified to shared.

- If no other cache has a copy of the line (clean or modified), then no signals are returned. The initiating processor reads the line and transitions the line in its cache from invalid to exclusive.

- Read Hit. When a read hit occurs on a line currently in the local cache, the processor simply reads the required item. There is no state change : the state remains modified, shared or exclusive.
- Write Miss. When a write miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. For this purpose the processor issues the signal on the bus that means *read-with-intent-to-modify* (RWITM). When the line is loaded it is immediately marked modified. With respect to other caches, two possible scenarios precede the loading of the line of data.

  First some other cache may have modified copy of this line (state = modified). In this case the alerted processor signals the initiating processor that another processor has a modified copy of the line. The initiating processor surrenders the bus and waits. The other processor gains access to the bus, writes the modified cache line back to main memory, and transitions the state of cache line to invalid (because the initiating processor is going to modify this line). Subsequently the initiating processor will again issue a signal to the bus of RWITM and then reads the line from main memory, modify the line in the cache and mark the line in the modified state.

  The second scenario is that the other cache has a modified copy of the requested line. In this case, no signal is returned, and the initiating processor proceeds to read in the line and modified. Meanwhile, if one or more caches have a clean copy of the line in the shared state, each cache invalidates its copy of the line and if one cache has a clean copy of the line in the exclusive state, it invalidates its copy of the line.
- Write Hit. When a write hit occurs on a line currently in the local cache the effect depends on the current state of that line in the local cache:

- *Shared*: Before performing the update, the processor must gain exclusive ownership of the line. The processor signals its intent on the bus. Each processor that has a shared copy of the line in its cache transitions the sector from shared to invalid. The initiating processor then performs the update and transitions its copy of the line from shared to modified.
- *Exclusive*: The processor already has exclusive control of this line so it simply performs the update and transitions its copy of the line from exclusive to modified.
- *Modified*: The processor already has exclusive control of this line and has the line marked as modified and so it simply performs the update.

All aspects of cache coherence have been incorporated in the algorithm that is discussed in the section 3.15. To implement these aspects a special structure has been introduced in the algorithm that simulates the cache controller and maintains the coherence. Here instead of generating signals, the simulated processors simply set the relevant bits in the common structure.

3.14 **Deletion Issues in Multiprocessor Set-Associative Caches.**

As mentioned in section 3.12 for multiprocessors and an invalidation-based coherence protocol, when a shared write causes a deletion, there raise certain issues regarding propagation of markers. Let us discuss these issues one by one and try to find solution for these problems. Since the hole or the marker can not be straightaway deleted for the reasons that it is to be catered for certain caches and not for others. A deliberate research on the issue leads us to following theorems.

Theorem 3.4. *For a deleted block 'D' and a referenced block 'R' such that 'R' is a miss in the cache and 'n' least significant bits of both 'D' and 'R' match then for all further references 'D' will not be considered for all* $2^{i\ set}$ *caches where i = 1 to n, but it will be accounted for all* 2^{j} ***set*** *caches where j>n.*

Example. We verify the above theorem by considering an example.

Figures 3.35 (a), (b) and (c) show a 2 set and 4 set (real caches), and the simulating stack respectively, after a sequence of references. Assume that block '8' has been invalidated and thus it should be deleted for some caches while it should not for others. Let the new reference is made for block '14' which is a miss and has to be brought in the cache from main memory. Now in 2 set cache '14' comes on the top of the stack and subsequent pushes fill up the gap thus the 2 set (real) cache holds a

sequence of references as shown in figure 3.36(a). Similarly the state of 4 set (real) cache and the simulating stack are also shown in figures 3.36 (b) and (c) respectively.

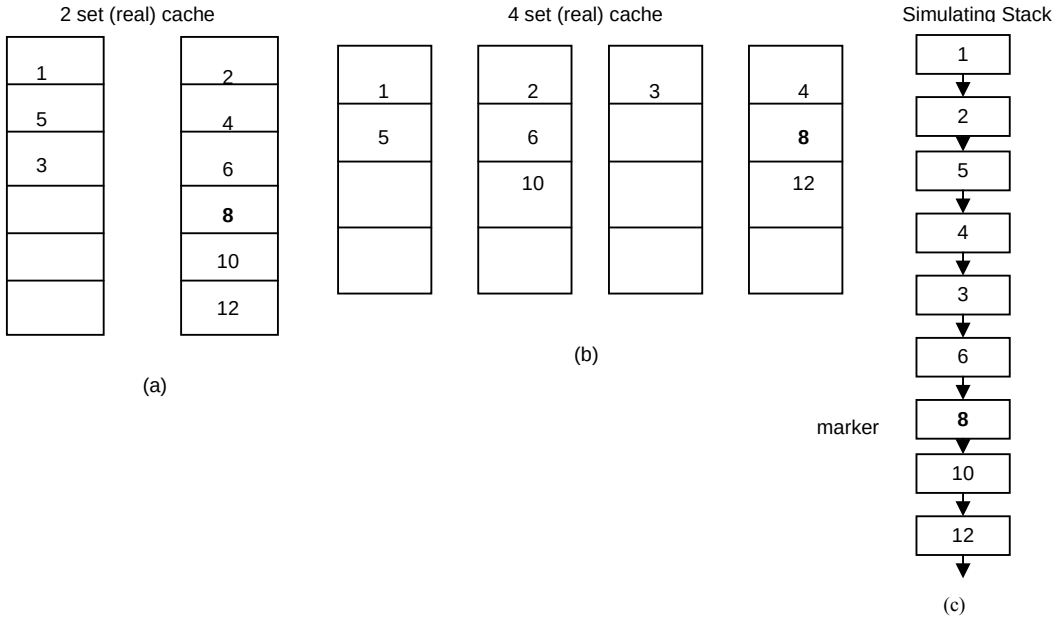


Figure 3.35 Deletion Issue (In case of Miss : Initial States)

Let new reference is made for block '12' which is a hit and LRU just demands that it should come at top of the stack. But we are also keeping a record of all those caches for which it is a hit and for which it is not by maintaining the distance counters. A keen observation of the real and simulated caches makes it clear that '12' is a hit at stack distance 6 in 2 set (real) cache and at stack distance 3 in 4 set (real) cache but in the simulating stack it is a hit at stack distance 7 for 2 set cache and at stack distance 3 for 4 set cache. It means that the hole should not be catered for a 2 set cache but it should be catered for a 4 set cache.

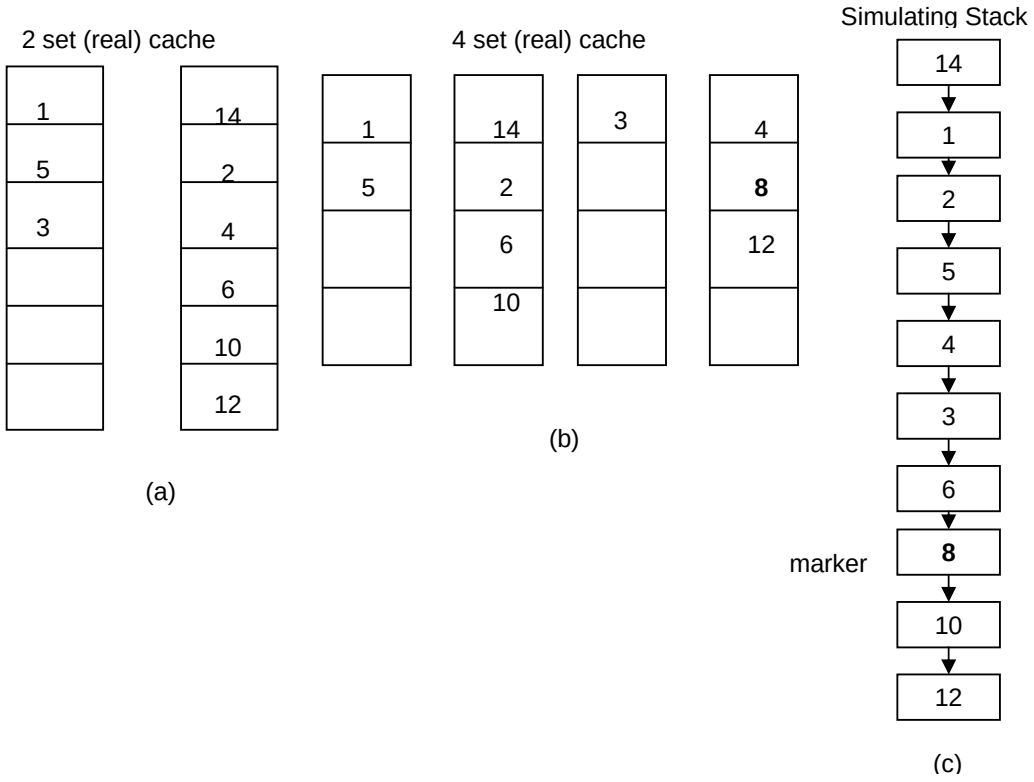


Figure 3.36 Deletion Issue (In case of Miss : States after Hole Propagation)

If we observe the bit patterns for digits '8' and '14' (i.e. 01000 and 01110), we'll find that only the first bit matches, it means these two references can fall in the same set only in the case of 2 set cache and hence '14' can fill the hole created by invalidation of '8' in 2 set cache only. Consequently if any further references are made then the hole should not be catered for any more in the 2 set cache but it should be catered for all other set-associative caches.

On the other hand, if the new reference is made for '28' instead of '14' then the hole created by '8' has to be filled in 2 set as well as 4 set cache. If we match the least significant bits (i.e. 00000 and 11100) then we find last two bits similar (i.e. n = 2) and we are not considering the hole for 2 set (2^1 set) and 4 set (2^2 set) caches which is again in accordance with our theorem.

Theorem 3.5. *For a deleted block 'D' and a referenced block 'R' such that 'R' is a hit in the cache and 'n' least significant bits of both 'D' and 'R' match then for all further references*

(i) 'D' will not be considered for all $\boldsymbol{2^i}$ ***set*** *caches where i = 1 to n and it will be accounted for all* $\boldsymbol{2^j}$ ***set*** *caches where j>n.*

(ii) Moreover a duplicate of 'D' will be accounted for all $\boldsymbol{2^i}$ ***set*** *caches where i= 1 to n and will not be considered for all* $\boldsymbol{2^j}$ ***set*** *caches where*

j>n, provided that the duplicate occurs at the same place where 'R' was previously residing.

Example. We verify the above theorem by considering an example

Figures 3.37 (a), (b) and (c) show a 2 set and 4 set (real caches), and the simulating stack respectively, after a sequence of references. Assume that block '8' has been invalidated and thus it should be deleted for some caches while it should not for others. Let the new reference is made for block '14' which is a hit and has to be brought in the cache from main memory. Now in 2 set cache '14' comes on the top of the stack and subsequent pushes fill up the gap thus the gap is pushed to a location where block '14' was residing previously. 2 set (real) cache, after the propagation of the marker is shown in figure 3.37 (a). Similarly the state of 4 set (real) cache and the simulating stack are also shown in figures 3.37 (b) and (c) respectively.

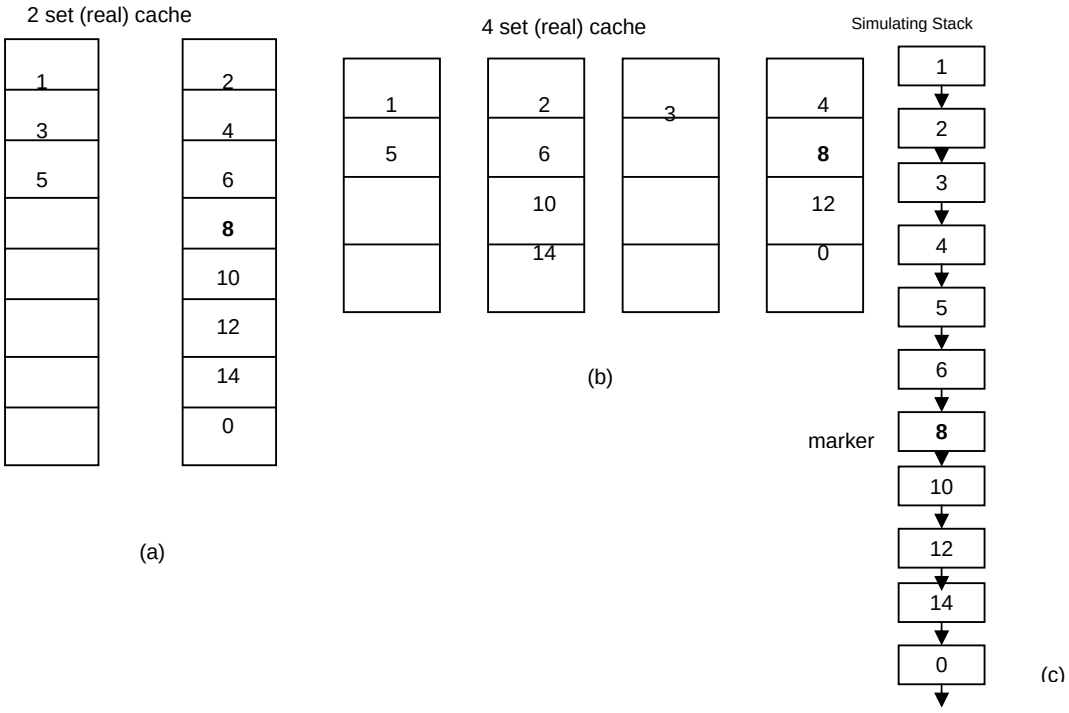


Figure 3.37 Deletion Issue (In case of Hit : Initial States)

Let new reference is made for block '0' which is again a hit and LRU just demands that it should come at top of the stack. But we are also keeping a record of all those caches for which it is a hit and for which it is not by maintaining the distance counters. A keen observation of the real and simulated caches makes it clear that '0' is a hit at stack distance 8 in 2 set (real) cache (because the marker has moved to stack distance 7) and at stack distance 4 in 4 set (real) cache. Moreover in the simulating stack it is a hit at stack distance 8 for 2 set cache and at stack distance 4 for 4 set cache which is in agreement to real caches. Still there is a problem because in 2 set (real) cache the marker is at stack distance 7 and in 4 set (real) cache it is at stack distance 2

for the fourth set. The current state of simulating cache (figure 3.38(c)) show that the simulating stack is in agreement with 4 set (real) cache but not with 2 set (real) cache. If in simulating stack we move the marker to stack distance 7 to bring it in agreement with 2 set (real) cache then it no more remains in agreement with the 4 set (real) cache.

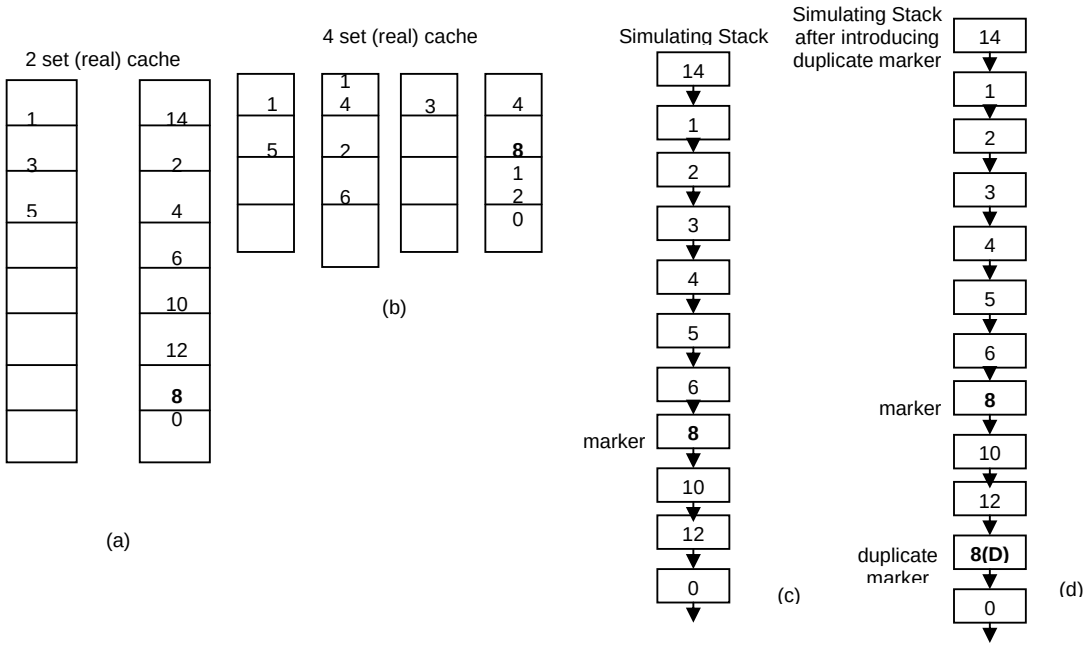


Figure 3.38 Deletion Issue (In case of Hit : States after Hole Propagation)

Here, basic issue is the decision for new location of the marker (hole) in the simulating stack. This issue can be resolved if we duplicate the marker and move this duplicate to the same location where we found the hit (as it actual happens in real caches). By introducing two markers in the simulating stack we can bring our simulating stack in accordance with the real caches (figure 3.38 (d)). The only remaining problem is the decision as when to consider the original hole and when to consider the duplicate.

.

If we observe the bit patterns for digits '8' and '14' (i.e. 01000 and 01110), we'll find that only the first bit matches, this means that these two references can fall in the same set only in the case of 2 set cache. Since we introduced the duplicate to bring our simulating stack in agreement with the 2 set (real) cache it means that the duplicate should be considered while simulating 2 set cache. Moreover, for the reasons described while verifying Theorem 3.4, it is obvious that original marker must not be considered while simulating a 2 set cache. Same principle should be followed for any further references.

On the other hand the original marker should be considered while simulating 4 set and 8 set caches but the duplicate must not be considered for these. Same concepts

can be verified further by applying different sequences of references to *M* set caches, where *M* can be any number.

Above described theorems sufficiently cover all the cases of delete and give the solution for placement of markers. In actual implementation of these concepts (which will be covered in detail in next section) we have introduced an integer and we set its bits to show whether to consider a marker or not. For instance, for the case of figure 3.36 (c) a bit pattern of [00000000000000000000000000000001] shows that the marker should not be considered for a 2 way cache but it should be considered for rest all caches. Similarly for the case described in 3.37 (c) a bit pattern [11111111111111111111111111111110] reflects that the duplicate should be considered for a 2 set cache and should be neglected for the rest of cache. These concepts will be covered again in next section.

The last vital point is the understanding that either of these theorems will always apply on a sequence of references while dealing with shred writes in case of multiprocessor based environments. If the next reference in the queue is a miss in cache Theorem 1 will govern the changes in simulating stack, whereas if it is a hit Theorem 2 will come in power.

3.15 **Implementation of One-Pass Simulation Technique for Multiprocessor Set-Associative Caches.**

After considering all the options available and after discussing different issues of the trace driven simulations in uniprocessor based environments, now we are at a stage where we can extend all those ideas to multiprocessor based environments. Here we make certain assumptions and develop our algorithm, and consequently, our software for the defined parameters only, however these algorithms are flexible enough to incorporate further changes and modifications.

We develop our algorithm for shared writes and we assume that already 'treated' traces are available for processing. We define 'Treated' traces as those which are pre-collected and pre-reduced in the best possible manner. Secondly we assume that there are 32 processors working parallel and are arranged in an array. We also assume that each processor has its own private cache at level-1, whose minimum size is 4 K byte. These parameters are not rigid and suitable modifications can be done for any required changes. We also assume that minimum and maximum limits for line size are dictated by the user. Same is the case with the degree of associativity which can vary between 1 way (direct mapped) to 8 way (fully associative).

We implement our algorithms in "C" language to simulate multiprocessor set-associative write back caches. The algorithms are in "C" like syntax for ease of understanding and comprehension. An approach which is best used by experienced programmers is to declare variable names which are self explanatory is also used here. Sections 3.12, 3.13 and 3.14 are referred since these provide the basic understanding for these algorithms.

The structure of the caches needed to implement the simulation is given in the following algorithm

```
structure Cache
{
```

```
structure Read--Hit--Structure
{
 Integer  HitArray[Array Of][ MaxAssociativity + 1]
                                        ::: Comments:::;//0,1,2,3....8 deg of
                                           associativity

 Read--Hit--Structure[Array Of] [ MaxPossibleSetBits +1]
                                         :::Comments:::;//from 0 to
                                           MaxPossibleSetBits 0  represents pow(2,0)
                                           only one set
                                        :::Comments::: this structure stores the no
                                           of read hits in the Cache having pow
                                           (2, MaxPossibleSetBits + 1) sets and deg
                                           of       associativity  of(1,2,3,4,5,6,7,8) we
                                           calculate only(1,2,4 and 8)
```

```
   structure Write--Hit--Structure
         {
Integer  HitArray[Array Of][ MaxAssociativity + 1]          ::: Comments:::;//0,1,2,3....8
                                                               deg of associativity
         }
Write-Hit-Structure [Array Of] [MaxPossibleSetBits + 1]  :::Comments:::;//from 0 to
                                                               MaxPossibleSetBits
                                                            :::Comments::: this structure
                                                               stores the number of write
                                                               hits in the Cache having
                                                               pow(2,
                                                               MaxPossibleSetBits + 1)
                                                               sets and deg of
                                                               associativity
                                                               of(1,2,3,4,5,6,7,8) we
                                                               calculate only (1,2, 4 and
                                                               8)
```

```
   structure Write--Avoidance--Structure
         {
Integer  HitArray[Array Of][ MaxAssociativity + 1]          :::Comments:::;//0,1,2,3....8
                                                               deg of associativity
         }
Write-Avoidance-Structure[Array Of]
                              [ MaxPossibleSetBits + 1]     :::Comments:::;//from 0 to
                                                              MaxPossibleSetBits
```

:::Comments::: this structure stores the no of writes being avoided in the (write back)Cache having pow(2, MaxPossibleSetBits + 1) sets and deg of associativity of(1,2,3,4,5,6,7,8) we calculate only(1,2,4 and 8)

Stack(Address Of) Top :::Comments:::;//top of the stack

Stack(Address Of) Previous :::Comments:::;//previous element to the current stack element

Stack(Address Of) Current :::Comments:::;//current element of the stack being considered

Integer TotalNoOfRef :::Comments:::;//total no of references read from the file

Integer TotalNoOfReads :::Comments:::;//total no of read references in the file

Integer TotalNoOfWrites :::Comments:::;//total no of write references in the file

Integer TotalNoOfDeletes :::Comments:::;//not used so far because delete request can be from other Cache

Integer ArrayOfDeletes :::Comments:::;//represents how many elements will be added
:::Comments::: for a particular reference

Stack(Address Of) ArrayOfDel[Array Of]
[MaxPossibleSetBits + 1] :::Comments:::;//this is the array
:::Comments::: of Integers for the deletes that will be added in the stack for
:::Comments::: particular sets

Integer Above[Array Of][MaxPossibleSetBits + 1]
:::Comments:::;//from 0 to MaxPossibleSetBits
:::Comments::: 0 represents pow(2,0) only one set

```
                                        :::Comments:::the value in the
                                          Above[Array Of][] array is the no. of
                                          references coming
                                        :::Comments::: before a given reference
                                          for pow(2,i) no of sets

  char  AccessType[Array Of][10]        :::Comments:::;//access type is read write
                                          or delete.
                                        :::Comments:::This information is stored
                                          in this string per reference

  Integer  ResetDel                     :::Comments:::;//when finding which
                                          block is deleted for which set
                                        :::Comments:::we should be conscious not
                                          to include two deleted blocks for one
                                        :::Comments:::particular set. this Integer
                                          indicates which set has been catered
                                        :::Comments:::for and which not the rest
                                          explanation will be given
                                        :::Comments:::while using this variable

  Integer  Address                      :::Comments:::;//will contain the address
                                          read from the file

  Integer  Tag                          :::Comments:::;//results in removing
                                          LineBits from  Address

  Integer  Found                        :::Comments:::;//hit in the Cache or not
                                            0=miss,1=hit,2=hit but delete
}
```

Simulation of main memory is implemented by following algorithm

```
Structure  SharedMemory                 :::Comments:::represents the contents in
                                            the struct rep shared data
{
  Integer  Block;                       :::Comments:::represents the shared
                                          block
  Integer Flag;                         ::: Comments:::flag represent which
                                          Cache processes
                                        :::Comments:::the block and its first bit
                                            represents whether the block
                                        :::Comments:::in the shared memory is
                                          modified  or not(1 = modified)
                                        :::Comments:::(0 = notmodified)
                                        :::Comments:::the rest of the bits in flag
                                          represents which Cache has the
```

```
                                        :::Comments:::block(i-e)bit 2 represents
                                          Cache1 bit 3 represents Cache2
                                        :::Comments:::and so on
    SharedMemory(Address Of)  Next      :::Comments:::;//next element in the
shared memory
};
```

Structure of main memory

```
structure   MainMemory
{

 SharedMemory(Address Of)  Top;         :::Comments:::top of the memory link
                                          list stack
 SharedMemory(Address Of)  Previous;    :::Comments:::previous to the current
                                          element of the stack being examined
 SharedMemory(Address Of)  Current;     :::Comments:::represents the current
                                          stack being examined
}
```

In find

If we have found the address as in previous algorithms we will not increment any read or writes.

We will just access it and then delete it not from the link list but make it invisible for some caches while keeping it visible for other caches the reason is explained in sections 3.12, 3.14.

```
Find(Cache(Address of) cache)           :::Comments:::;//find in the stack
{
                                         :::Comments:::;//cout<<"entered
                                           find"<<endl;
                                        :::Comments:::;//getch();
   AdjustmentsBeforeFind(cache);

   cache(contains)Found=0;              :::Comments:::;// 0 means not found 1
                                           means found 2 means found but found in
                                           delete

   if the case in the brackets is true
   (strcmp(cache(contains)AccessType,Delete)!=0)
                                        :::Comments:::;//if the case in the brackets
                                           is true access type is not delete
        {
        if the case in the brackets is true
         (cache(contains)Top == NULL)   :::Comments:::;//if the case in the brackets
                                           is true very first address
        {}
```

```
    else do this                                    :::Comments:::;//if the case in the brackets
                                                       is true not the very first element
        {
                                                    :::Comments:::;//cout<<"not very first
                                                       address";
                                                   :::Comments:::;//getch();
 cache(contains)Current = cache(contains)Top;

while(cache(contains)Current != NULL && (cache(contains)
Above[Array Of][MaxMatchBits]<MaxDataAssociativity))
                                                    :::Comments:::;//you don't need to continue
                                                       searching if the case in the brackets is
                                                       true the address has gone down greater
                                                    :::Comments:::;//than the max deg of
                                                       associativity of the largest
                                                    :::Comments:::;//cache
                          {
if the case in the brackets is true
(cache(contains)Tag == cache(contains)Current(contains)Block)
                          {
                           if the case in the brackets is true
  (cache(contains)Current(contains)RWD == 2)
                                                      :::Comments:::;//if the case in the
                                                       brackets is true the block is delete
{ cache(contains)Found = 2;}                          :::Comments:::;//found but delete

 if the case in the brackets is true
 (cache(contains)Current(contains)RWD != 2)
        {
cache(contains)Found=1;                               :::Comments:::;// 1 means found
        }
                                  }
if the case in the brackets is true
(cache(contains)Tag == cache(contains)Current(contains)Block &&
cache(contains)Found == 1) break;
                                                    ::: Comments:::;//if the case in the brackets
                                                       is true found and the block is not
                                                       deleted
else do this
                                                    :::Comments:::;//if the case in the brackets
                                                       is true it is not the current block the
                                                       current block
                          {
                                FindRightMatchBits(cache);
                                                    :::Comments:::;//find least significant the
                                                       case in the brackets is true icant bits
                                                       matching between the block of the
                                                       current cache and the tag
```

```
                                                    :::Comments:::;//to see how far it has
                                                       gone  down the stack
                                                    :::Comments:::;//in finding the req
                                                       address wetger found or not
                                                    :::Comments:::;// cout<<endl;
                                for(Integer k=0;k<=MaxPossibleSetBits;k++)
                                      {
                                                    :::Comments:::;//cout<<cache(contains)
                                                       Above[Array Of][k];
                                      }

                                cache(contains)Previous= cache(contains)Current;
                                cache(contains)Current =
cache(contains)Current(contains)Next;
                               }
                              }
                    }
        ArrangeAfterFind(cache);


          }


    if the case in the brackets is true
     (strcmp(cache(contains)AccessType,Delete) ==0)      :::Comments:::;//if the case in
                                                            the brackets is true the new
                                                            reference is delete
{                                                       :::Comments:::;//cout<<"Enter
                                                            ing Delete" <<endl;
        if the case in the brackets is true
(cache(contains)Top != NULL)                            :::Comments:::;//it will always
                                                            be null but the condition is
                                                            just precautionary
                    {
                    cache(contains)Current = cache(contains)Top;
                          while(cache(contains)Current != NULL)
                             {
                             if the case in the brackets is true
                              (cache(contains)Tag == cache(contains)Current(contains)Block)
                                      {
                                           cache(contains)Current(contains)RWD = 2;
                                           cache(contains)Current(contains)Delse do thist =
                                           ( (Integer)pow(2,MaxMatchBits+1) -1);
                                                        :::Comments:::;//this causes
                                                            every bit of the
                                                        :::Comments:::;//delse do thist
                                                            to be one indicating that the
```

```
                                                                    block is deleted for every
                                                                    set and
                                                                :::Comments:::;//should be
                                                                    catered for rightmatchbits
                                                                    because
                                                                :::Comments:::;//it will
increment cache(contains)above[Array Of][] to indicate
                                                                :::Comments:::;//that this
                                                                    block is actually present in
                                                                    the
                                                               :::Comments:::;//actual cache
                                                                    which this program is
                                                                    simulating
                                    }
                                    if the case in the brackets is true
          (cache(contains)Tag == cache(contains)Current(contains)Block) break;
                              cache(contains)Current = cache(contains)Current(contains)Next;
                                    }
          ArrangeAfterFind(cache);
                    }
          }

}
```

Now next time we do the right match bits we find whether block is deleted or not and if it is delete for which sets as explained earlier (section 3.14).

```
FindRightMatchBits(Cache(Address of) cache)
{
                                                          :::Comments:::;//cout<<"entering
                                                            right match"<<endl;

   if the case in the brackets is true
    (cache(contains)Current(contains)RWD NotEqualTo 2)
                                                        :::Comments:::;//if the case in the
                                                            brackets is true the block is not
                                                            deleted
          {
                    cache(contains)Above[Array Of][0]++;
                                                        :::Comments:::;//it will be above in
                                                            one set any way
      Integer    XOR = cache(contains)Tag ^ cache(contains)Current(contains)Block;
      Integer AND;

      for(Integer i=0; i<MaxMatchBits; i++)
                    {
                              AND = XOR Anding (Integer)pow(2, i);
```

```
        if the case in the brackets is true
( AND EqualTo (Integer)pow(2,i) )                    :::Comments:::;//if the case in the
                                                          brackets is true i bits don't match
                                                         break;
                                                    :::Comments:::;//it is not above for
                                                       any other set

              else do this
                        {
                                                    :::Comments:::;//cout<<"bit"<<(i+1)
                                                      <<cache(contains)Current(contains)
                                                      Block<<"is matching"<<"  ";

                    cache(contains)Above[Array Of][i+1]++;
                                                    :::Comments:::;//no of ref above for
                                                       pow(2,i) no of sets
                        }
                  }
        }


   if the case in the brackets is true
   (cache(contains)Current(contains)RWD EqualTo 2)
        {
                                                  :::Comments:::;//cout<<"Access type is
                                                     delete";
     Integer Delse do thistForNew = 0;            :::Comments:::;//this basically provides
                                                     Delse do thist
                                                  :::Comments:::;//for the new element
                                                     that will be added as a delete in the
                                                  :::Comments:::;//link-list for some sets
                                                     and not for other sets
                                                   :::Comments:::;// ResetDel =
                                                     ((Integer)pow(2,MaxMatchBits) -1);

      if the case in the brackets is true
((cache(contains)Current(contains)Delse do thist Anding (Integer)pow(2,0))
NotEqualTo 0)
            {
                                                 :::Comments:::;//cout<<"bit one is 1";
                                                 :::Comments:::;//this means check bit 1 if
                                                   the case in the brackets is true it is "1"
                                                   then this delete
                                                ::Comments:::;//block is present in the
                                                    actual "one set" cache at this level
              cache(contains)Above[Array Of][0]++;
```

```
                                                    :::Comments:::;//this if the case in the
                                                      brackets is true for one set a special
                                                      case
        }
    if the case in the brackets is true
( (cache(contains)ResetDel Anding
(Integer)pow(2,0))NotEqualTo )                      :::Comments:::;//if the case in the
                                                      brackets  is true pow(2,0) bit of
                                                      ResetDel is 0 then it means that set no
                                                      pow(2,i+1) is catered for
        {
                if the case in the brackets is true
((cache(contains)Current(contains)Delse do thist Anding (Integer)pow(2,0))
NotEqualTo 0)
        {
                                                    :::Comments:::;//cout<<"entering if the
                                                      case in the brackets is true "<<endl;
          cache(contains)ResetDel = ( cache(contains)ResetDel
Anding( ~(Integer)pow(2,0)) );
                    Delse do thistForNew = ( Delse do thistForNew Oring
(Integer)pow(2,0) );
                                                  :::Comments:::;//the new set added will not
                                                      be catered for these sets represented by
                                                      bits
                  cache(contains)Current(contains)Delse do thist =
(cache(contains)Current(contains)Delse do thist Anding( ~(Integer)pow(2,0)) );
                                                    :::Comments:::;//cout<<cache(contains)
                                                      Current(contains)Delse do thist<<endl;
                                                    :::Comments:::;//cout<<"Resetdel"<<ca
                                                      che(contains)ResetDel<<endl;
                }
        }

    Integer     XOR = cache(contains)Tag ^ cache(contains)Current(contains)Block;
                                                    :::Comments:::;//same "0" opposite "0ne"
                                                        Integer AND;

    for(Integer i=0; i<MaxMatchBits; i++)
                                                  :::Comments:::;//i=0 means bit 1
        {
                AND = XOR Anding (Integer)pow(2, i);
                                                  :::Comments:::;//checking bit pow(2,i)

          if the case in the brackets is true
( AND EqualTo (Integer)pow(2,i) )
                                                  :::Comments:::;//if the case in the brackets
                                                    is true i bits don't match  break;
```

```
        else do this
                                                    :::Comments:::;//means i bits do match for
                                                       set pow(2,i+1) and onwards i-e 2 and
                                                       onwards
                {
                                                   :::Comments:::;//cout<<"bit"<<(i+1)<<"of"
                                                       <<cache(contains)Current(contains)Block
                                                       <<"is matching"<<"  ";

                    if the case in the brackets is true
((cache(contains)Current(contains)Delse do thist Anding (Integer)pow(2,i+1))
NotEqualTo 0)
                    cache(contains)Above[Array Of][i+1]++;

                    if the case in the brackets is true
( (cache(contains)ResetDel Anding (Integer)pow(2,i+1)) NotEqualTo )
                                                    :::Comments:::;//if the case in the brackets is
                                                       true pow(2,i+1) bit of ResetDel is 0 then
                                                       it means that set no pow(2,i+1) is catered
                                                       for
                    {
                        if the case in the brackets is true
((cache(contains)Current(contains)Delse do thist Anding (Integer)pow(2,i+1))
NotEqualTo 0)
                        {
                            cache(contains)ResetDel =
( cache(contains)ResetDel Anding( ~(Integer)pow(2,i+1)) );
                            cache(contains)Current(contains)Delse do thist =
(cache(contains)Current(contains)Delse do thist Anding( ~(Integer)pow(2,i+1)) );
                            Delse do thistForNew = ( Delse do thistForNew Oring
(Integer)pow(2,i+1) );
                                                     :::Comments:::;//the new set added
                                                        will not be catered for these sets
                                                        represented by bits
                        }
                    }
                                                    :::Comments:::;//cout<<"resetdel"<<ca
                                                        che(contains)ResetDel<<endl;
                                                    :::Comments:::;//Above[Array
                                                       Of][i+1]++;
                                                    :::Comments:::;//no of ref above for
                                                       pow(2,i) no of sets
                }
        }
                                                   ::Comments:::;//end of for
    if the case in the brackets is true
(Delse do thistForNew NotEqualTo 0)
        {
```

```
                cache(contains)ArrayOfDel[Array of][cache(contains)ArrayOfDeletes]
= new Stack;

cache(contains)ArrayOfDel[Array of][cache(contains)ArrayOfDeletes](contains)RWD
= 2;
        cache(contains)ArrayOfDel[Array
                                        of][cache(contains)ArrayOfDeletes](contains)
                                        Block =
                                        cache(contains)Current(contains)Block;
        cache(contains)ArrayOfDel[Array
                                        of][cache(contains)ArrayOfDeletes](contains)
                                        Next = cache(contains)ArrayOfDel[Array
                                        of][cache(contains)ArrayOfDeletes+1];
        cache(contains)ArrayOfDel[Array
                                        of][cache(contains)ArrayOfDeletes](contains)
                                        Delse do thist = Delse do thistForNew;
        cache(contains)ArrayOfDeletes++;
           }

         }


}
```

Cache Coherence Protocol

Following is the algorithm that performs cache coherence protocol as discussed above (section 3.13).

```
            if the address is in the shared region then enter the following function
SharedRegion(MainMemory(Address of)
mainmemory,CCacheSimulationView::Cache(Address of) cache)
{
                                        :::Comments:::;//cout<<"entered shared
                                           region"<<endl;
                                        :::Comments:::;//getch();
   Integer Found=0;
                                        :::Comments:::;//found in shared memory
                                           or not "0" means not found
   if the case in the brackets is true
    (mainmemory(contains)Top EqualTo NULL)
                                        :::Comments:::;//if the case in the brackets
                                          is true very first shared memory
                                        :::Comments:::;//access
{
                                        :::Comments:::;//cout<<"main memory is
                                          null"<<endl;
      mainmemory(contains)Top = new SharedMemory;
      mainmemory(contains)Top(contains)Block = cache(contains)Tag;
```

```
      mainmemory(contains)Top(contains)Next  = NULL;
      mainmemory(contains)Top(contains)Flag    = 0;
                                                  :::Comments:::;//initialise the flag
      if the case in the brackets is true
 (strcmp(cache(contains)AccessType,Read) EqualTo 0)
                                                  :::Comments:::;//if the case in the brackets
                                                    is true the cache is reading the block
                  {
                              mainmemory(contains)Top(contains)Flag =
( mainmemory(contains)Top(contains)Flag Anding (~(Integer)pow(2,0)));
                                                 :::Comments:::;// first bit=0 means that
                                                   cache has just asked for reading
                                                 :::Comments:::;//it has not modif the case in
                                                    the brackets is true it
               mainmemory(contains)Top(contains)Flag =
( mainmemory(contains)Top(contains)Flag Oring (Integer)pow(2,CacheNo+1) );
                                                 :::Comments:::;//(for cache no 0 it should be
                                                    1)it means that this cache
                                                  :::Comments:::;//has the copy of the block
                                                    and has no rite to modif the case in the
                                                    brackets is true it
                                                ::Comments:::;//without notify the case in
                                                    the brackets is true ication
                  }
      else do this
                  {
                              if the case in the brackets is true
(strcmp(cache(contains)AccessType,Write) EqualTo 0)
               mainmemory(contains)Top(contains)Flag =
(mainmemory(contains)Top(contains)Flag Oring (Integer)pow(2,0) );
                  :::Comments:::;//first bit=1 means that cache has  asked for modif the
case in the brackets
                  :::Comments:::;// or has  modif the case in the brackets is true
mainmemory(contains)Top(contains)Flag =
( mainmemory(contains)Top(contains)Flag Oring (Integer)pow(2,CacheNo+1) );
                  :::Comments:::;//(for cache no 0 it should be 1)it means that this cache
                  :::Comments:::;//has the (private)copy of the block and has a rite to
                  :::Comments:::;//modif the case in the brackets is true                  }
      mainmemory(contains)Current      = mainmemory(contains)Top;
      mainmemory(contains)Previous     = mainmemory(contains)Top;


         }
   else do this     :::Comments:::;//if the case in the brackets is true  not very first shared
memory reference
         {
                     :::Comments:::;//cout<<"main memory is not null"<<endl;
```

```
mainmemory(contains)Current = mainmemory(contains)Top;

while(mainmemory(contains)Current NotEqualTo NULL)
        {
                if the case in the brackets is true (cache(contains)Tag EqualTo
mainmemory(contains)Current(contains)Block)
                {
                        Found = 1;     :::Comments:::;//found(1)
                if the case in the brackets is true
(strcmp(cache(contains)AccessType,Read) EqualTo 0)
                        {
                                :::Comments:::;//check the first bit of the flag
if the case in the brackets is true
                        :::Comments:::;//is 1 then some
                        :::Comments:::;//other cache has this block in dirty state
                      Integer Temp =
( mainmemory(contains)Current(contains)Flag Anding ((Integer)pow(2,0)) );

                    if true the case in the brackets is true(Temp EqualTo
(Integer)pow(2,0))     :::Comments:::;//it means that some other
                        :::Comments:::;//cache has made this block private to it
self
                        :::Comments:::;//(remember it will only be one cache)
                                {
                                        :::Comments:::;//cout<<"some other
cache has made the block dirty"<<endl;

                            RequestBlockBack(mainmemory,cache);
                               :::Comments:::;//bring it back(make it read only
in that
                               :::Comments:::;//cache)

                               }
                      mainmemory(contains)Current(contains)Flag =
( mainmemory(contains)Current(contains)Flag Oring (Integer)pow(2,CacheNo+1) );
                        :::Comments:::;//(for cache no 0 it should be 1)it means
that
                        :::Comments:::;//this cache has the copy of the block
and has
                        :::Comments:::;//no rite to modif the case in the brackets
is true

                      }
                else do this
                                :::Comments:::;//if the case in the brackets is
                                   true access type is write or delete
                        {
```

```
                                        :::Comments:::;//remove the delete in the
future(or should you)
                                if the case in the brackets is
true(strcmp(cache(contains)AccessType,Write) EqualTo 0 )
                                        {
                                                :::Comments:::;//check the first bit of
the flag if the case in the brackets is true then
                                        :::Comments:::;//some other cache has this block
in dirty state
                                Integer Temp =
( mainmemory(contains)Current(contains)Flag Anding ((Integer)pow(2,0)) );

                                if the case in the brackets is true (Temp EqualTo
(Integer)pow(2,0))     :::Comments:::;//it means that some
                                        :::Comments:::;//other cache has made this block
private to it
                                        :::Comments:::;//self (remember it wail only be
one cache
                                                {
                                                        :::Comments:::;//cout<<"cache
has made the block dirty"<<endl;

RequestBlockBackInvalidate(mainmemory(contains)Current(contains)Flag,cache);
                                                 :::Comments:::;//in case of write only one
can have the total
                                                 :::Comments:::;//access
                                                }
                                else do this
                                        :::Comments:::;//all the other caches has the
block in read state

InvalidateAllOtherCaches(mainmemory(contains)Current(contains)Flag,cache);
                                        :::Comments:::;//delete this shared block in
all other caches
                                        :::Comments:::;//containing it because the
current cache
                                        :::Comments:::;//requires this block to be
private

                                mainmemory(contains)Current(contains)Flag
= 0;     :::Comments:::;//no other cache should have the copy of it now
                                mainmemory(contains)Current(contains)Flag
= (mainmemory(contains)Current(contains)Flag Oring (Integer)pow(2,0) );
                                         :::Comments:::;//first bit=1 means that
cache has  asked for
                                         :::Comments:::;//modif the case in the
brackets is true
```

```
                                                mainmemory(contains)Current(contains)Flag
= (mainmemory(contains)Current(contains)Flag Oring (Integer)pow(2,CacheNo+1) );
                                                ::Comments:::;//(for cache no 0 it should
be 1)it means that
                                                ::Comments:::;//this cache has the copy of
the block and has
                                                ::Comments:::;// rite to modif the case in
the brackets is true
                                        }
                                    if the case in the brackets is true
(strcmp(cache(contains)AccessType,Delete) EqualTo 0 )
                                        {
                                                Integer
Change=1;     ::Comments:::;//should i change the "0"th bit of the flag or not
                                            ::Comments:::;//cout<<"accesss type is delete";
                                        mainmemory(contains)Current(contains)Flag =
(mainmemory(contains)Current(contains)Flag Anding(~
(Integer)pow(2,CacheNo+1) ));
                                        for(Integer i=1; i<=NoOfProcessors; i++)
                                                {
                                                        if the case in the brackets is true
( (i-1) NotEqualTo CacheNo )
                                                        {
                                                                Integer AND =
mainmemory(contains)Current(contains)Flag Anding (Integer)pow(2, i);
    if the case in the brackets is true
( AND EqualTo (Integer)pow(2,i) )
                                                ::Comments:::;//(bit 0 is there)if the case
                                                   in the brackets is true i bit
                                                ::Comments:::;//match(means i'th
                                                   processor has the block
        {
        Change = 0;
                                                 ::Comments:::;//cout<<"cache no"<<(i-
                                                   1)<<"has it"<<endl;
         }
            }
                }
 if the case in the brackets is true
 (Change EqualTo 1)
  {
  mainmemory(contains)Current(contains)Flag = 0;
  }
        }
                }
                }
if the case in the brackets is true
 (cache(contains)Tag EqualTo mainmemory(contains)Current(contains)Block )break;
```

```
                                                    :::Comments:::;//if the case in the brackets
                                                       is true found and the block is not
                                                       deleted
            else do this
                                                    :::Comments:::;//if the case in the brackets
                                                       is true it is not the current block the
                                                       current block
                        {
                                mainmemory(contains)Previous=
mainmemory(contains)Current;
                        mainmemory(contains)Current =
mainmemory(contains)Current(contains)Next;
                        }
                }

     if the case in the brackets is true (Found EqualTo 0)
                                                :::Comments:::;//i-e not found
                {
                        ArrangeSharedRegion(mainmemory,cache);
                }
  }
  mainmemory(contains)Current = mainmemory(contains)Top;
  while(mainmemory(contains)Current NotEqualTo NULL)
        {
                                                :::Comments:::;//cout<<(mainmemory(con
                                                   tains)Current(contains)Block)<<"
                                                   "<<(mainmemory(contains)Current(cont
                                                   ains)Flag)<<"     ";
     mainmemory(contains)Current = mainmemory(contains)Current(contains)Next;
        }
                                                :::Comments:::;//cout<<endl<<endl;
}

InvalidateAllOtherCaches(Integer Flag,Cache(Address of) cache)
{
                                                :::Comments:::;//it should not be called if
                                                   the case in the brackets is true the
                                                   shared block is in the same cache
                                                :::Comments:::;// asking for it}
                                                :::Comments:::;//cout<<"entered
                                                   Invalidating all other other caches "<<"
                                                   ";
                                                :::Comments:::;//cout<<Flag<<"  "<<Flag;
  for(Integer i=1; i<=NoOfProcessors; i++)
        {
                if the case in the brackets is true
( (i-1) NotEqualTo CacheNo )
                {
```

```
                        Integer AND = Flag Anding (Integer)pow(2, i);
        if the case in the brackets is true
( AND EqualTo (Integer)pow(2,i) )
                                                ::Comments:::;//(bit 0 is there)if the case
                                                  in the brackets is true i bit  match(means
                                                  i'th
                                                ::Comments:::;//processor has the block in
                                                  a read state
        {
                                                ::Comments:::;//cout<<"cache no"<<(i-
                                                  1)<<"has it"<<endl;
                Stack(Address of) Temp  = FindRequest(cache,AndingCache1[Array
Of][i-1]);
                                                ::Comments:::;//cout<<Temp(contains)Bloc
                                                  k<<"   ";
                                                ::Comments:::;//cout<<Temp(contains)RW
                                                  D<<endl;

                        }
                }
        }

}


RequestBlockBackInvalidate(Integer Flag,Cache(Address of) cache)
{
                                                ::Comments:::;//it should not be called if the
                                                  case in the brackets is true the shared block
                                                  is in the same cache
                                                ::Comments:::;// asking for it
                                                ::Comments:::;//cout<<"entering
                                                  requestblockbackinvalidate";
                                                ::Comments:::;//cout<<Flag<<"  "<<Flag;
  for(Integer i=1; i<=NoOfProcessors; i++)
        {
                if the case in the brackets is true
( (i-1) NotEqualTo CacheNo )
                {
                        Integer AND = Flag Anding (Integer)pow(2, i);
        if the case in the brackets is true
( AND EqualTo (Integer)pow(2,i) )
                                                ::Comments:::;//(bit 0 is there)
                                                ::Comments:::;//if the case in the brackets is
                                                  true i bit  match(means i'th
                                                ::Comments:::;//processor has the block in
                                                  dirty state
                        {
```

```
                                        :::Comments:::;//cout<<"cache no"<<(i-
1)<<"has it"<<endl;
                Stack(Address of) Temp  = FindRequest(cache,AndingCache1[Array
Of][i-1]);
                                        :::Comments:::;//cout<<Temp(contains)Block<<
                                            "   ";
                                        :::Comments:::;//cout<<Temp(contains)RWD<<
                                            endl;
                        }
                }
        }

}



RequestBlockBack(MainMemory(Address of)
mainmemory,CCacheSimulationView::Cache(Address of) cache)
{
                                                :::Comments:::;//it should not be called
                                                    if the case in the brackets is true the
                                                    shared block is in the same cache
                                               :::Comments:::;// asking for it

                                                :::Comments:::;//now check which bit of
                                                    the flag is 1 other than bit 0 because
                                                    onlyone cache can have a particular
                                                    block in dirty state
                                                :::Comments:::;//cout<<"entered request
                                                    block back "<<"   ";
                                                :::Comments:::;//cout<<"mainmemory(c
                                                    ontains)Current(contains)Flag"<<"
                                                    "<<mainmemory(contains)Current(co
                                                    ntains)Flag;
   for(Integer i=1; i<=NoOfProcessors; i++)
        {
        if the case in the brackets is true
( (i-1) NotEqualTo CacheNo )
                {
                        Integer AND = mainmemory(contains)Current(contains)Flag
Anding (Integer)pow(2, i);
        if the case in the brackets is true
( AND EqualTo (Integer)pow(2,i) )               :::Comments:::;//(bit 0 is there)if the
                                                    case in the brackets is true i bit
                                               :::Comments:::;//match(means i'th
                                                    processor has the block in dirty state
                        {
```

```
                                                    :::Comments:::;//cout<<"cache no"<<(i-
1)<<"has it"<<endl;
                    Stack(Address of) Temp  = FindRequest(cache,AndingCache1[Array
Of][i-1]);
                                                    :::Comments:::;//cout<<Temp(contains)
                                                       Block<<"   ";
                                                     :::Comments:::;//cout<<Temp(contains)
                                                      RWD<<endl;
                    mainmemory(contains)Current(contains)Flag =
( mainmemory(contains)Current(contains)Flag Anding (~(Integer)pow(2,0)));
                                                    :::Comments:::;// first bit=0 means that
                                                       cache has just asked for
                                                    :::Comments:::;//reading it and has not
                                                       modif the case in the brackets is
                                                       trueied it
                              }
                    }
          }

}
```

Chapter Four

The Acumen

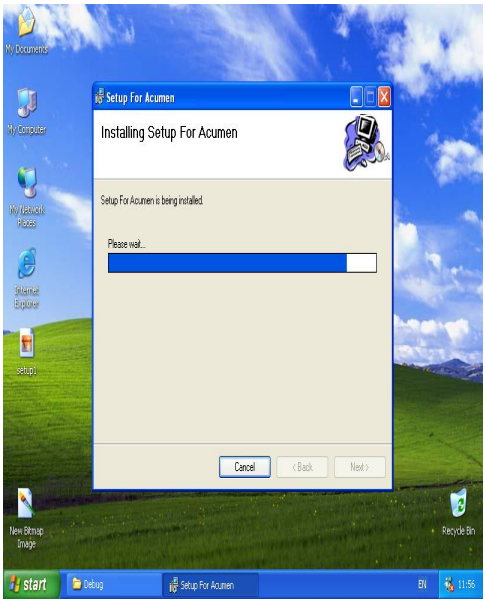
4.1 **Introduction to Acumen.**

After the discussion of first three chapters the software which we developed is named *Acumen* because of it's ability to efficiently propose appropriate cache designs basing on user's choices.

4.2 **How to Use the Software.**

The software is packaged for best deployment and is portable. When we explore the compact disc containing the software we find the familiar set up icon. Following the easy steps as per instructions displayed the software can be deployed on any windows platform containing windows installer (issued in service pack).

4.2.1 **Screen shot of Installation Process.**



To find best cache design user will have to give following inputs:-

- An address trace.
- Number of Processors in the system for which cache design is required. Up to 32 processors are permitted. (Simulation can work for uni-processor based environment also).
- User has the choice of combined and split caches.
- For split caches user can further choose between instruction and data cache.
- Minimum and maximum cache size. i.e limits are defined by the user
- Minimum and maximum line size.

- Degree of associativity i.e. minimum and maximum associativity
- Whether the cache required will be write through or write back.
- After necessary processing results will be available for the user in various combinations of line and bar graphs. Main combinations are :-
    - Cache size vs. read hits / write hits / write avoidance.
    - Line size vs. read hits / write hits / write avoidance
    - Degree of associativity vs. read hits / write hits / write avoidance.

4.3 **Graphical User Interface.**

4.3.1 **Initial Screen.**

The user starts the simulation either by an icon on the toolbar or by pressing init simulation from a drop down menu.

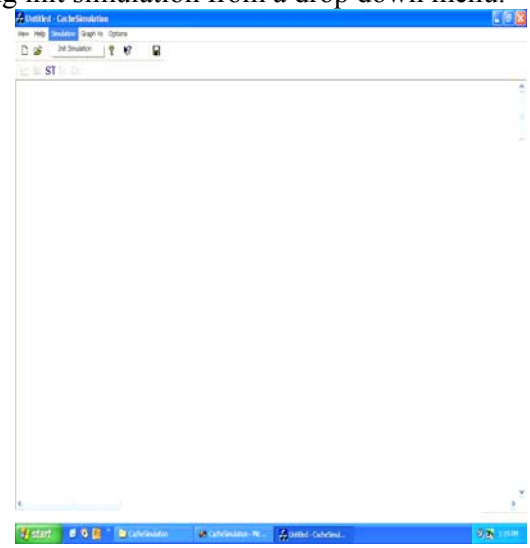


Figure 4.1 Initial Screen.

4.3.2 **User Input Choices**

The user gives the input choices through radio and sliding buttons

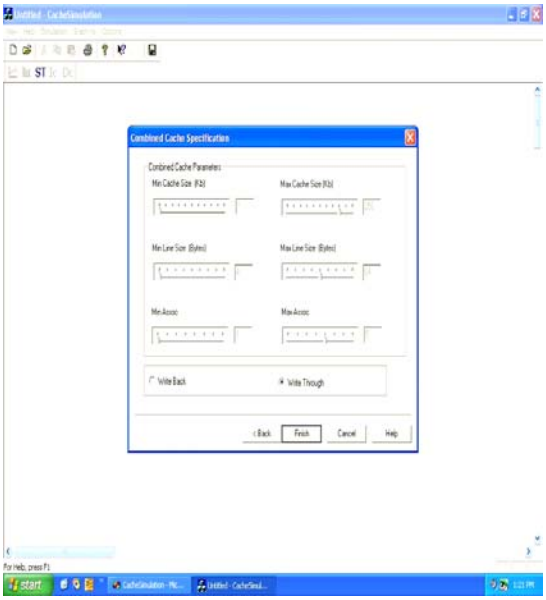


Figure 4.2 User Input .

4.3.3 **User Choices for Outputs**

The user can obtain out put in different forms and has choice of best caches depending on various parameters.

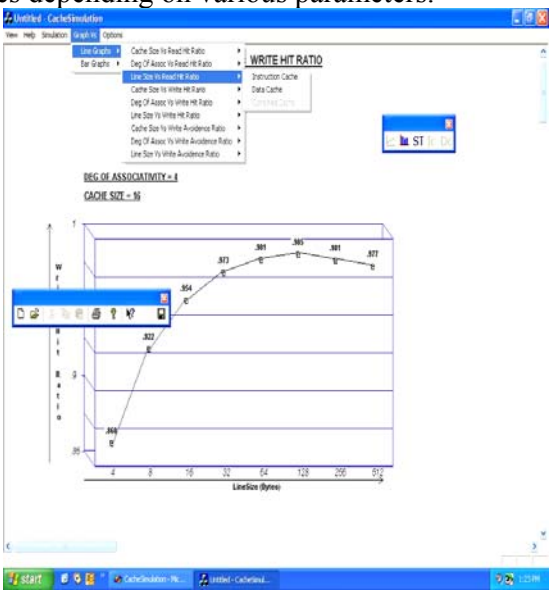


Figure 4.3 Output Choices.

4.4 **The Out Put.**

4.4.1 **Line Graphs**

The out put with line graphs

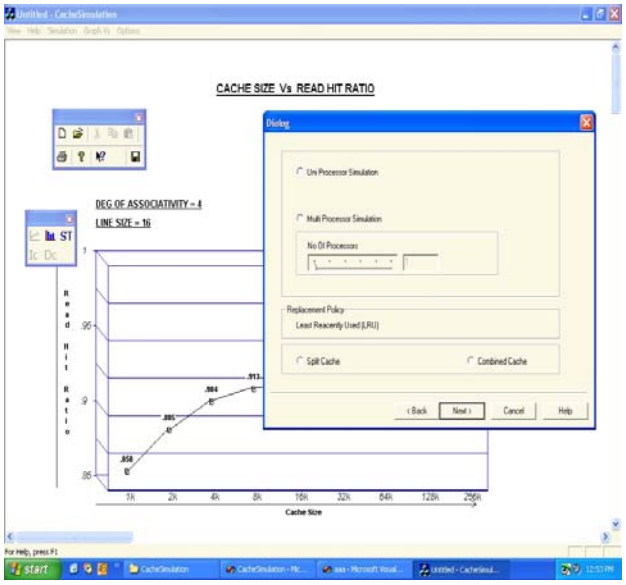


Figure 4.4 Line Graph.

4.4.2 **Bar Graphs**

The out put with bar graphs

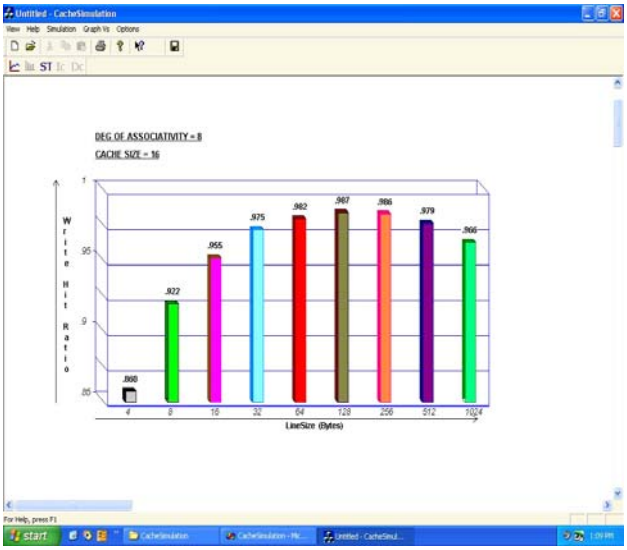


Figure 4.5 Bar Graph

Chapter Five

Conclusion and Future Prospects

5.1 **Conclusion.**

All associative simulations can be made faster by taking advantage of set hierarchy, a necessary but not sufficient condition for inclusion. Since we find the set hierarchy usually holds between set associative caches (for example those that use bit selections) with set hierarchy , the time to run most of the set associative simulations is within 30% of the time of one stack simulation. This facilitates the rapid simulation of direct mapped and set-associative caches.

The principal impact of this project is that all associative simulations with set hierarchy allows a similar or wide cache design space to be examined in comparable or less simulation time than required with stack simulation.

With all associative simulations, requiring comparable time for practical CPU caches (normally CPU caches are less than or equal to 32- way set- associative and use bit selection to map references to sets), one can evaluate mixed , instruction only and data only of two block size and numerous associativities and sizes in particular the use of all associativity simulation facilitated the evaluation of large number of CPU cache designs.

We further amortized the cost of reading reduced traces by devising a one-pass simulation algorithm that can simulate many write-back caches during a single simulation run, yielding speed-ups of two orders of magnitude over a naive method. In addition, we extended the trace reduction and the efficient simulation techniques to parallel multiprocessor cache simulations.

We have shown how stack analysis can be extended to important new areas. The ability to collect transfer ratios, considering both reads and writes, for all memory sizes in a single pass reduces simulation time by as much as 90 percent compared to running 8-10 individual simulations, making this metric much more reasonable to collect. The transfer ratio is increasingly important in the study of shared-memory systems, including multiprocessor caches and network file systems. Equally important, the ability to easily simulate set associative caches, including write backs and write throughs.

5.2 **Future Prospects.**

Software and hardware projects can not be declared as perfect or last words but just milestones in their respective fields. Trends of refinements and enhancements are the basic attributes in the further development of the projects. Like other projects our cache design simulation for multiprocessor based environment can be further explored in many ways.

The performance evaluation of any system can be done through simulations economically and efficiently. Trace driven simulations are of many types and each can be further explored. Trace collection of executing programmes further divides into three main types of hardware probes, processor and programme simulations including modifications in instruction sets to obtain traces via special buffers and CPU stalling , i.e. software approach and lastly the compiler approach of programme profiling and debugging.

Trace reduction is another important task. Processors generate millions of address- traces in a split of time. Sampling these traces and reducing them to evaluate correct performance of system is what the need of time is.

Brigham Young University has specially established a Trace Distribution Centre, Performance Evaluation Laboratory and National Trace Collection Centre, their site can be visited at http://tds.cs.byu.edu. Establishment of such evaluation centres can be a national level project for future development of computer sciences.

Finally the trace processing for uni-processor based systems and multiprocessor based systems can be undertaken. One prospect can be of enhancing level of sharing among different number of processors, at different cache levels and in different architectures of multiprocessors for example multiprocessor based system with ringed architecture.

Appendix A

ABC'S OF CACHE

Introduction

The purpose of caching is to improve the average access time to items in memory by keeping the most frequently used items in a small, fast, cache memory and by leaving the remainder in a larger, slower memory. The contents of cache are checked on each reference; if the referenced item is present in cache, then the item is available at the speed of the cache. If not, then the item is read into cache from memory, replacing something already cached. The speed of the combined memory system is a function of the two memory speeds and the probability that the referenced item is in cache.
There are a large number of design parameters to any cache, most of which must be considered in any analysis of that design. We briefly present definitions of a number of these.

Blocking. The cache may be divided into fixed-size blocks or variable-size segments. Blocks are also referred to as pages in the context of virtual memory and lines or sectors in the context of a processor cache. The cache block or line size may be equal to the amount of data retrievable in one memory cycle, or it may require several memory cycles to fetch a block. A larger block size reduces per-block overhead and provides a form of pre fetch, discussed below.

Replacement Policy. The replacement policy determines which block to remove when the cache is full and a new block must be fetched. Commonly suggested policies include the Least Recently Used (LRU) policy, First-In First-Out (FIFO), Least Frequently Used (LFU), and Random (RAND). An optimal policy, MIN, exists, but is unrealizable in practice because it requires knowledge of the future. The MIN policy does not consider writes or deletes and is known to be non optimal if writes are considered.

Write Policy. The write policy determines when a modification is presented to secondary storage. Writes may always go directly to secondary storage using the write-through or store-through policy. Alternatively, the write may go to the cache to be written at some later time, usually when the block is about to be replaced, using the write-back or copy-back policy. Write-back is motivated by the expectation that the block will be modified several times before it has to be written. Clearly, write-back can never cause more accesses than write-through and usually far fewer. On the other hand, since it deals in blocks rather than words, write- back may increase the number of bytes written. In addition, dirty blocks may remain in the cache for a long time, leading to reliability issues in large volatile caches such as file system caches in main memory. The decrease in memory traffic from write-back makes it very valuable in systems with limited memory bandwidth such as shared-bus multiprocessor systems. Write-back is also desirable in file system caches because many files are temporary and may never have to be written.

Write Allocate. When a written block is not present in a write-through cache, the block may be inserted in the cache (write allocate) or the cache may be bypassed altogether. Write allocate is again motivated by locality-the expectation that the written block will soon be referenced again. A write-back cache always allocates a cache block to the written block.

Write-Fetch. If write allocate is used by a cache where partial-block modification is allowed, and the block to be written is not in the cache (a write miss), then it is usually necessary to fetch the block prior to modifying it. This write-fetch is needed, for example, if one word of a multiword block is being written. The alternative is to keep track of the portion(s) of the cache block that are "valid," which becomes costly when several disjoint portions of a large block are written. However, there are situations in which write-fetch can be avoided such as when the entire block are being overwritten, or when the contents of the rest of the block are predictable (e.g., when the block is a "new" block in a file system).

Prefetch. Because of spatial locality, a reference to a block often implies that the next block will soon be referenced. It is possible to take advantage of this anticipated reference and to prefetch the next block in advance. This reduces the delay when the next block is actually referenced. Prefetch is advantageous when it can be overlapped with processing of other references or when two or more blocks can be fetched in much less time than all of them individually, as is the case with disk secondary storage. Although it reduces the delay, prefetch increases memory traffic unless all pre fetched blocks are referenced before they are replaced. It may also result in memory pollution in which a soon-to-be-referenced block is displaced to make room to prefetch an unnecessary block. If a prefetch is only permitted in conjunction with a fetch, then the policy is a demand prefetch policy. Demand prefetch is desirable when the overhead of a fault is large; demand prefetch amortizes this over two (or more) blocks. With modern memory systems and file system caches, it is simple and inexpensive to initiate a prefetch even if the referenced block is present.

Metrics The performance of a memory system can be measured in several ways. Perhaps the most widely used is the miss ratio, which is the fraction of references that were not satisfied by the cache. Conversely, the hit ratio is the fractions that were satisfied by the cache. The miss ratio is latency metric since it determines the apparent access time of the memory system. The effective access time for any multilevel hierarchy is given by 2 tihi, where ti is the access time to the ith level, and hi is the fraction of references satisfied by the ith level cache. Sometimes overlooked is the fact that the access time to each level should include any queuing delays. These are usually negligible in a single-processor system, but may become important when several processors compete for access to a single secondary store.

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

$$MR_R C) = m(C)/N, \qquad (1.1)$$

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

$$MR_{WT} = (m_T(C)) + W/N \quad (1.2)$$

where m_T (C) is the number of reads that "miss," and W is the number of write references.

When write-back is used, a write could result in two accesses to secondary storage, one to fetch the block and another later to write it. The miss ratio is now given by

$$MR_{WB}(C) = (m_T(C) + m_W(C) + dp(c)/N \quad (1.3)$$

where m (C) is the number of write misses (i.e., write fetches), and dp(C) is the number of dirty blocks "pushed" from a cache of size C. This becomes

$$MR_{WB}(C) = (m(C) + dp(C))/N \quad (1.4)$$

by using the fact that a write-fetch is actually just a read reference and occurs if the block reference "misses."

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

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

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

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

The general form of the traffic ratio computation is

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

where m_w (C) is again the number of write misses; *dp*(C) is again the number of write-backs; and *f*(C) is the number of prefetched blocks. This expression assumes that write-fetch is used. Notice that the traffic ratio is identical to the miss ratio when there is no prefetching, no write buffering, and the cache block size is the same as BP.

A third metric is the transfer ratio, which is the ratio of secondary storage accesses with and without cache . This metric has also been called the transaction ratio G. Gibson, personal communication 19861, the I/O ratio , and the swapping ratio . The

transfer ratio is similar to the traffic ratio but is more appropriate when performance is dominated by the cost of a memory access, relatively independent of the number of bytes transferred. Thus it is appropriate for disk caches and often for networks using small (1K or less) messages. For example, the transfer ratio decreases if two blocks are read from disk in a single I/O, whereas the traffic ratio is the same regardless of the number of I/Os used to transfer the data.

The transfer ratio also has an indirect effect on the access time if there are enough transfers to create contention, particularly in multiple processor systems with shared memory. Assuming that prefetches occur only when the referenced block is not in cache (demand prefetch), then they do not affect the transfer ratio. A general expression for the transfer ratio is

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

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

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a memory hierarchy. This takes advantage of locality and cost/performance of memory technologies.

Principle of Locality. This says that most programs do not access all code or data uniformly . This principle, plus the guideline that smaller hardware is faster, led to the hierarchy based on memories of different speeds and sizes. Since fast memory is expensive, a memory hierarchy is organized into several levels each smaller, faster and more expensive per byte than the next level. The goal is to provide a memory system with cost almost as low as the cheapest level of memory and speed almost as fast as the fastest level. The levels of the hierarchy usually subset one another: all data in one level is also found in the level below, and all data in that lower level is found in the one below it, and so on until we reach the bottom of the hierarchy. Note that each level maps addresses from a larger memory to a smaller but faster memory higher in the hierarchy. As part of address mapping, where miss rate is the fraction of accesses that are not in the cache and miss penalty is the additional clock cycles to service the miss. Recall that a block is the minimum unit of information that can be present in the cache (hit in the cache) or not (miss in the cache).

The ABC's. Cache is the name generally given to the first level of the memory hierarchy encountered once the address leaves the CPU. We start our description of caches by answering the four common questions for the first level of the memory hierarchy. The memory hierarchy is given the responsibility of address checking: hence protection schemes for scrutinizing addresses are also part of the memory hierarchy.

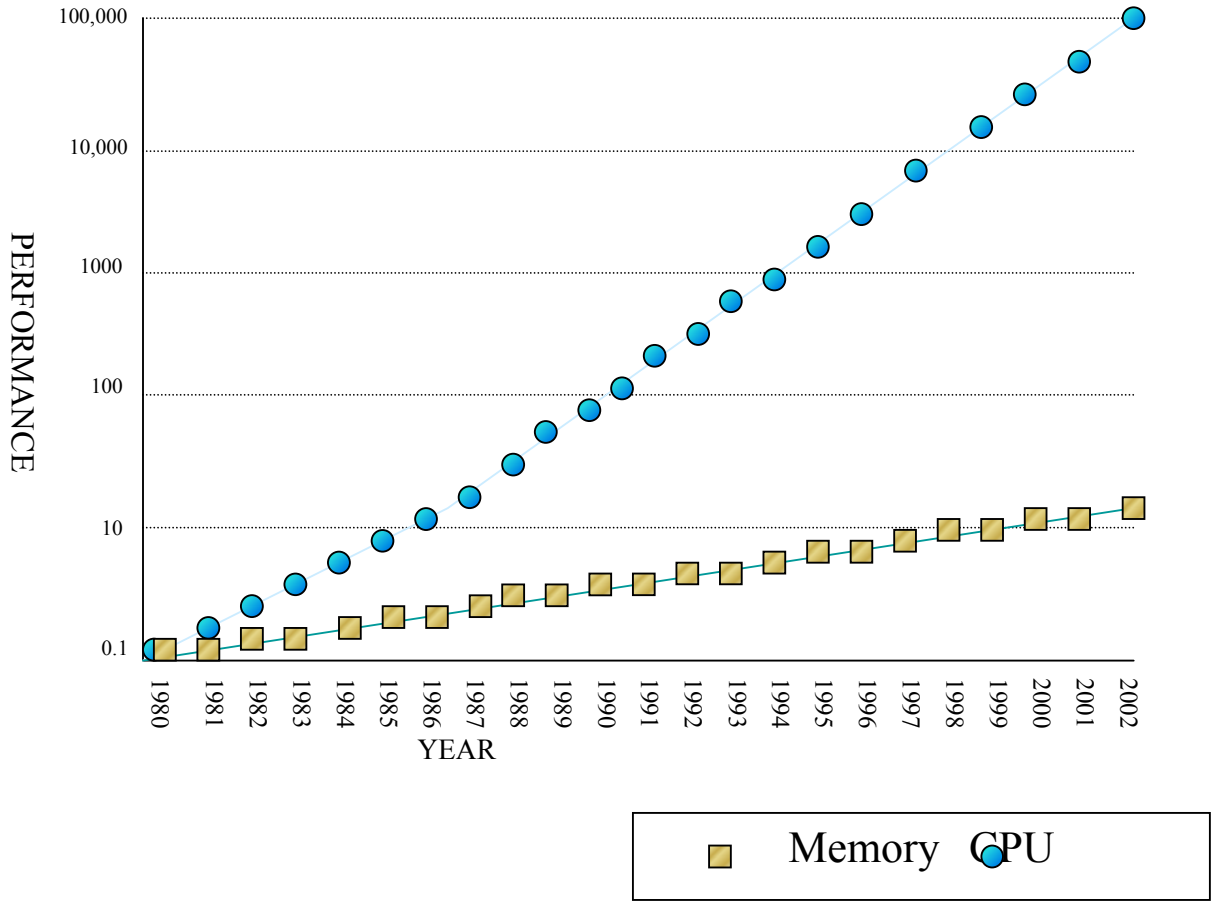


Figure . It plots CPU performance projections against the historical performance improvement in main memory access time .Clearly there is a processor memory performance gap that computer architects try to close

Where can a block be placed in a cache? The restrictions on where a block is placed create three categories of cache organization:

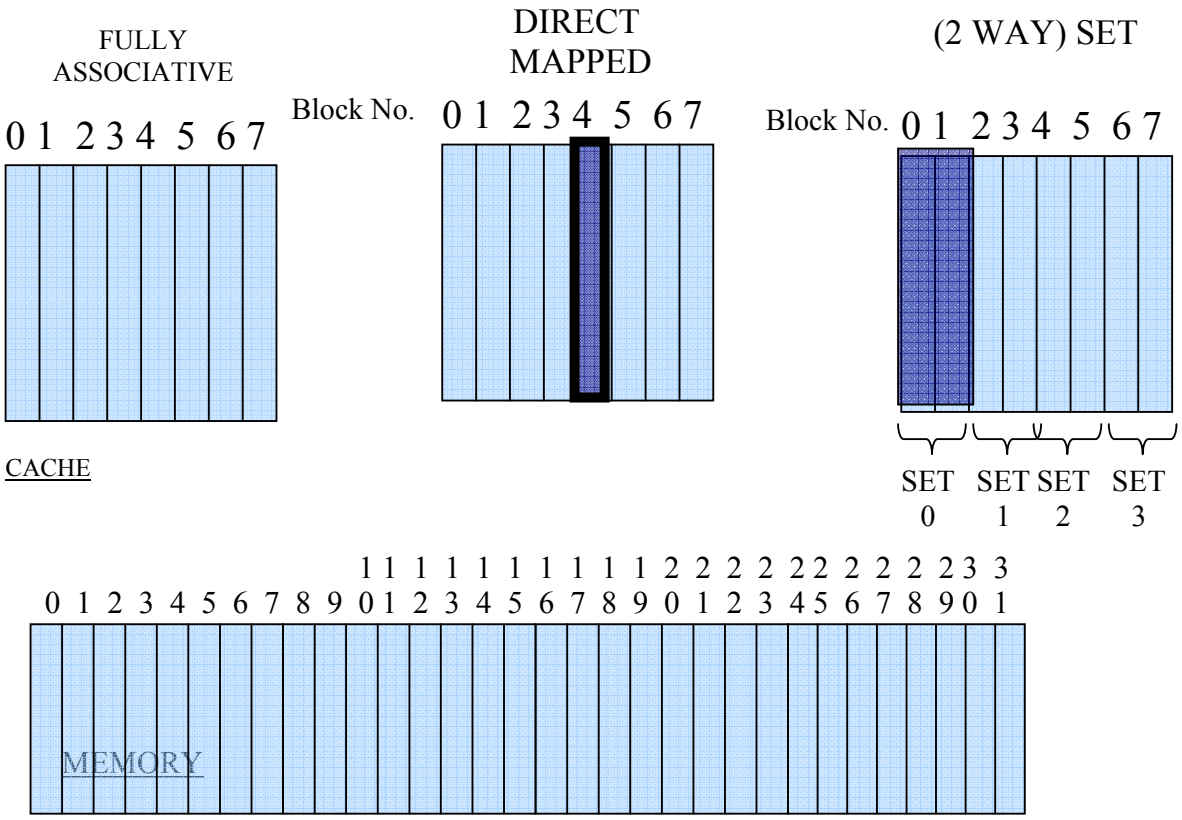


Figure. This example cache has eight block frames and memory has 32 blocks Real caches contain hundreds of block frames and real memories contain millions of blocks. The set-associative organization has four sets with two blocks per ser, called two-way set as associative. Assume that there is nothing in the cache and that the block address in questions identifies lower-level block 12. The three options for caches are shown left to right. In associative, block 12 from the lower level can go into any of the eight block frame 4 (12 modules). Set associative, which has some of both features, allows the block to be placed anywhere se 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either block 0 or block 1 of the cache.

- If each block has only one place it can appear in the cache, the cache is said be direct mapped. The mapping is usually

  (Block address) MOD (Number of block in cache)
- If a block can be placed anywhere in the cache, the cache is said to be fully associative.
- If a block can be placed in a restricted set of places in the cache, the cache is said to be set associative. A set is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be

place anywhere within that set. The set is usually chosen by bit selection; that is,

(Block address) MOD (Number of sets in cache)

If there are n blocks in a set, the cache placement is called n-way set associative.

How a block is found if it is in the cache? Caches have and address tag on each block frame that gives the block address. The tag of every cache block that might contain the desired information is checked to see if it matches the block address from the CPU. As a rule, all possible tags are searched in parallel because speed is critical. There must be a way to know that a cache block does not have valid information. The most common procedure is to add a valid bit to the tag to say whether or not this entry contains a valid address. If the bit is not set, there cannot be a match on this address. Before proceeding to the next question, let's explore the

<table>
<tr><td colspan="2">Block address</td><td rowspan="2">Block offset</td></tr>
<tr><td>Tag</td><td>Index</td></tr>
</table>

Figure. The three portions of an address in set-associative or direct-mapped cache. The tag is used to check all the blocks in the set and the index is used to select these. The block is the address of the desired data within the block.

If the total cache size is kept the same, increasing associatively increases the number of blocks per set, thereby decreasing the size of the index and increasing the size of the tag.

Which block should be replaced on a cache miss?When a miss occurs, the cache controller must select a block to be replaced with the desired data. A benefit of direct mapped placement is that hardware decisions are simplified in fact, so simple that there is no choice. Only one block frame is checked for a hit and only that block can be replaced. With fully associative or set associative placement, there are many blocks to choose from on a miss. There are two primary strategies employed for selecting which block to replace.

- *Random.* To spread allocation uniformly, candidate blocks are randomly selected.
- *Least recently used (LRU).*To reduce the chance of throwing out information that will be needed soon accesses to blocks are recorded. The block replaced is the one that has been unused for the longest time. LRU makes use of a corollary of locality.

What happens on a write? Reads dominate processor cache accesses. All instruction accesses are reads and most instructions don't write to memory. The write policies often distinguish cache designs. There are two basic options when writing to the cache.

- *Write through (or store through)*. The information is written to both the block in the cache and to the block in the lower level memory.
- *Write back (also called copy back or store in)*. The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

Since the data are not needed on a write, there are two common options on a write miss.

- *Write allocate (also called fetch on write).* The block is loaded on a write miss followed by the write hit actions above. This is similar to a read miss.
- *No write allocate(also called write around).* The block is modified in the lower level and not loaded into the cache.

Cache Performance. The temptation for evaluating memory hierarch performance is to concentrate on miss rate. A check measure of memory hierarchy performance is the average time to access memory.

Average memory access time = Hit time + Miss rate x Miss penalty

Reducing cache Misses. Most cache research has concentrated on reducing the miss rate, so that is where we start our exploration. To gain better insights into the causes of misses, we start with a model that sorts all misses into three simple categories;

- *Compulsory.* The very first access to a block cannot be in the cache, so the block must be brought into the cache. There are called cold start misses of first reference misses.
- *Capacity.* If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur because of blocks being discarded and later retrieved.
- *Conflict.* If the block placement strategy is set associative or direct mapped conflict misses (in addition to compulsory and capacity misses) will occur be cause a block can be discarded and later retrieved if too many block s map to the set. These are also called collision misses or interference misses.

First Miss Rate Reduction Technique: Larger Block Size This simplest way to reduce miss rate is increase the block size. Large block sizes will reduce compulsory misses. This reduction occurs because the principle of locality has two components temporal locality and spatial locality. Large blocks take advantage of spatial locality.
At the same time, large blocks increase the miss penalty. Since they reduce the number of blocks in the cache, large blocks may increase conflict misses and even capacity misses if the cache is small.

Second Miss Rate Reduction Technique: Higher Associatively
There are two general rules of thumb; the first is that eight ways set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. The second observation called the 2; 1 cache rule of thumb and found on the front inside cover, is that direct mapped cache of size N has about the same miss rate as a 2 way set associative cache of size N/2.

Third Miss Rate Reduction Technique: Victim Caches. One solution that reduces conflict misses without impairing block rate is to add a small, fully associative cache between a cache and its refill path .

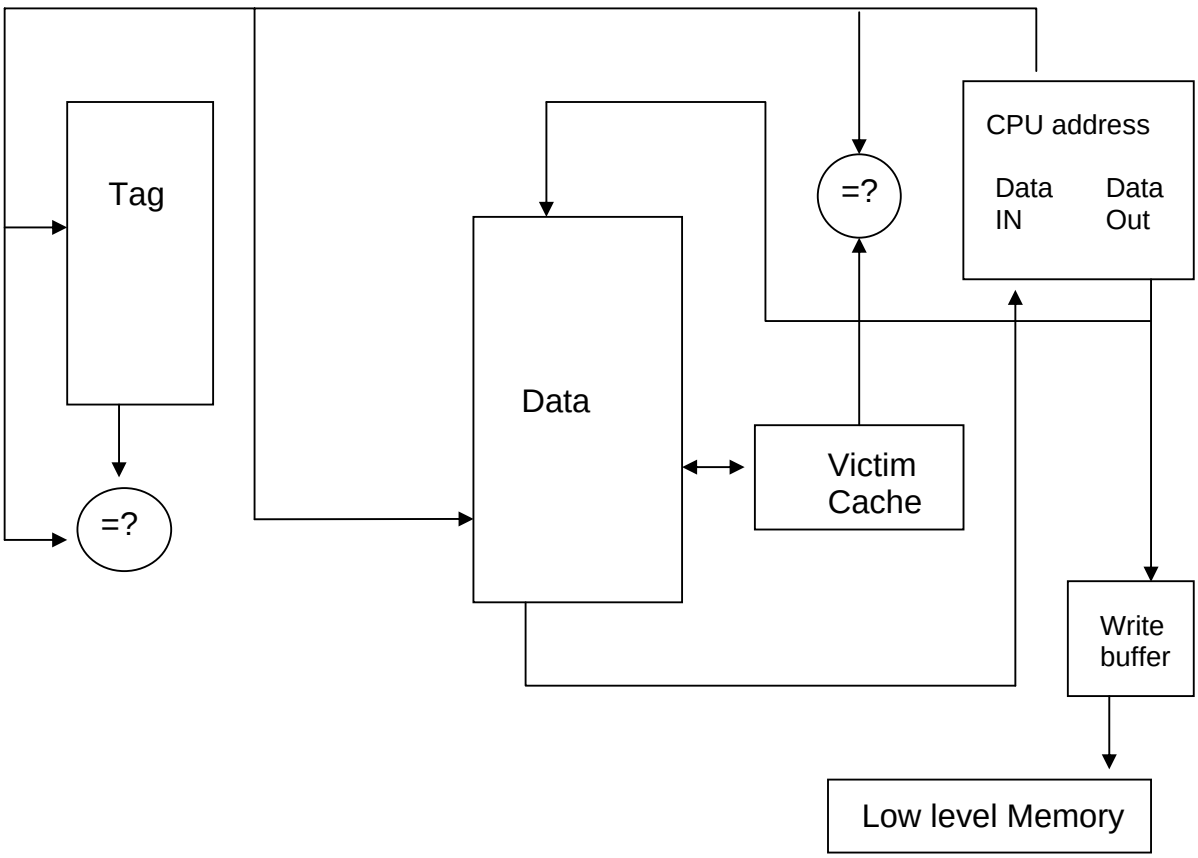


Figure.Placement of victim cache in the memory hierarchy

Fourth Miss Rate Reduction Technique: Pseudo-Associative Caches. A cache access proceeds just as in the direct-mapped cache for a hit. On a miss however, before going to the next lower level of the memory hierarchy another cache entry is of checked to see if it matches there. A simple way is to invert the most significant it of the index field to find the other block in the "pseudo set".

Pseudo-associative caches then have one fast and one slow hit time corresponding to a regular hit and a pseudo hit in addition to the miss penalty. The danger is if many of the fast hit times of the direct-mapped cache became slow hit times in the pseudo-associative cache then the performance would be degraded by this optimization. Hence it is important to be able to indicate for cache set which block should be the fast hit and which should be the slow one; one way is simply to swap the contents of block.

Fifth Miss Rate Reduction Technique: HW prefetching of instructions and data. This technique prefetches the items before they are requested by the processor. Both instructions and data can be prefetched directly into the cache Prefetching relies on utilizing memory bandwidth that otherwise would be a used and can actually lower performance if it interferes with demand miss. Help from compilers can reduce useless prefetching.

Sixth Miss Rate Reduction Technique: Compiler-controlled prefetching. This makes sense only if the processor can proceed while the prefetched data are being fetched, that is the caches continue to supply instructions and data while waiting for the prefetched data to return. Such it memory cache is called a nonblocking cache or lockup-free cache; we'll discuss it in more detail later.
Like hardware-controlled prefetching, the goal is to overlap execution with the prefetching of data. Issuing prefetch instructions incurs an instruction overhead, however, so care must be taken to ensure that such overheads do not exceed the benefits. By concentrating on references that are likely to be cache misses, programs can avoid unnecessary prefetches while improving average memory access time significantly.

Seventh Miss Rate Reduction Technique: Compiler Optimizations. This magical reduction comes from optimized software the hardware de-signer's favourite's solution. The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy to see if compile time optimizations can improve performance. Once again research is split between improvements in instruction misses and improvements in data misses.

Reducing Cache Miss Penalty

Reducing cache misses has been the traditional focus of cache research, but the cache performance formula assures us that improvements in miss penalty can be just a beneficial as improvements in miss rate.

First Miss Penalty Reduction Technique: Giving Priority to Read Misses over Writes. With a write-through cache the most important improvement is a write buffer of the proper size. Write buffer, however, do complicate memory accesses in that they might hold the updated value of a location needed on a read miss.
The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. A write buffer of a few words in a write-through cache will almost always have data in the buffer on a miss, thereby increasing the read miss penalty.
The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty bold to a buffer, then read memory, and then write memory. This way the CPU real for which the processor is probably waiting, will finish sooner. Similar to the situation above, if a read miss occurs, the processor can either stall until the buffer an empty or check the addresses of the words in the buffer for conflicts.

Second Miss Penalty Reduction Technique: Sub-block placement for Reduced Miss penalty. Suppose we are designing a cache that must fit one the chip. We may find that our tags are too large eight because they don't fit on the chip or because the are too slow. A simple solution is go to large blocks, which reduces tag store without decreasing the amount of information you can store in the cache of course the miss rate will likely improve, but the increase in miss penalty and make the large blocks a bad decision.
One solution is called sub-block placement. A valid bit is added to unit's storage than the full block, called sub-blocks. Only a single sub-block need be read a miss. The

valid bits specify some parts of the block as valid and some as invalid, so a match of the tag doesn't mean the word is necessary in the cache as the valid bit for that word must also be on.

Third Miss Penalty Reduction Technique: Early restart and Critical Word First

- *Early restart*- As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.
- *Critical word first*- Request the missed word first from memory and send it to the CPU as soon as it arrives, let the CPU continue while filling the rest of the words in the block. Critical-word first fetch is also called wrapped fetch and requested word first.

Fourth Miss Penalty Reduction Technique: Nonblocking caches to Reduce Stalls on Cache Misses. The potential benefits of this scheme are to allow the data cache to continue to supply cache hits during a miss. This "hit under miss" optimization reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of CPU.

Fifth Miss Penalty Reduction Technique: Second-Level Caches Adding another level of between the original cache and memory, the first level cache can be small cache to match the clock cycle time of the fast CPU, while the second-level cache be large enough to capture many accesses that would go to main memory thereby lessening the effective miss penalty. Summarizing the second-level cache considerations, the essence of cache sign is balancing fast hits and few misses. Most optimizations that help one hit the order. For second-level caches, there are many fewer hits than in the first level cache, so the emphasis shifts to fewer misses. This insight leads in large caches with fighter associativity and larger blocks.

Reducing Hit Time

Hit time is critical because it affects the clock rate of the processor, on memory machines today the cache access time limits the clock cycle rate, even machines that take multiple clock cycles to access the cache, Hence a fast hit time is multiplied in importance beyond the average memory access time formula be because it helps everything.

First Hit Time Reduction Technique: Small and Simple Caches

A time-consuming portion of a cache hit is using the index portion of the address to read the tag memory and then compare it to the address. Guideline suggests that smaller hardware is faster, and a small cache century helps the hit time. It is also critical to keep the cache small enough to hit of the same chip as the processor to avoid the time penalty of going off-chip. A main benefit of direct-mapped caches is that the signer can overlap the tag check with the transmission of the data. The effectively reduces hit time. Hence the pressure of a fast clock cycle encourages and simple cache designs for first-level caches.

Second Hit Time Reduction Technique: Avoiding Address Translation During Indexing of the Cache. Even a small and simple cache must cope with the translation

of a virtual address from the CPU to a physical address to access memory. Processors treat main memory as just another level of the memory hierarchy and thus the address of the virtual memory that exists on disk must be mapped onto the main memory.
The guideline of making the common case fast suggest that we use virtual addresses for the cache, since hits are much more common than misses. Such caches are termed virtual; caches, with physical cache used to identify the traditional cache that uses physical addresses. Virtual addressing eliminates address translation time from a cache hit. Then why doesn't everyone build virtually addressed caches? One reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed .Keeping caches small and simple and techniques to avoid delays of address translation will make both read hits and write hits faster. The next subsection concentrates only on writes.

Third Hit Time Reduction Technique: Pipelining Writes for Fast write Hits. Write hits usually take longer than read hits because the tag must be checked before writing the data; otherwise the wrong address would be written. One technique, used by the Alpha AXP 21064 and other machines, pipelines the writes . First tags and data are split so that they can be addressed independently. On a write, the cache compares the tag with the current write address, as usual. The difference comes with the write to the data portion of the cache that occurs during the tag compassion; it must be using some other address since the current write address is still being checked. The trick is that the cache uses the address and data from the previous write, which has already been determined to be a hit. Thus the logical pipeline is between writes, the second stage of the write occurs during the first stage of the next write (or during a cache miss). Therefore, writes can be performed back to back at one per clock cycle because the CPU does not have to wait for the tag check before writing. Reads play no part in this pipeline since they already operate in parallel with the tag check.
.

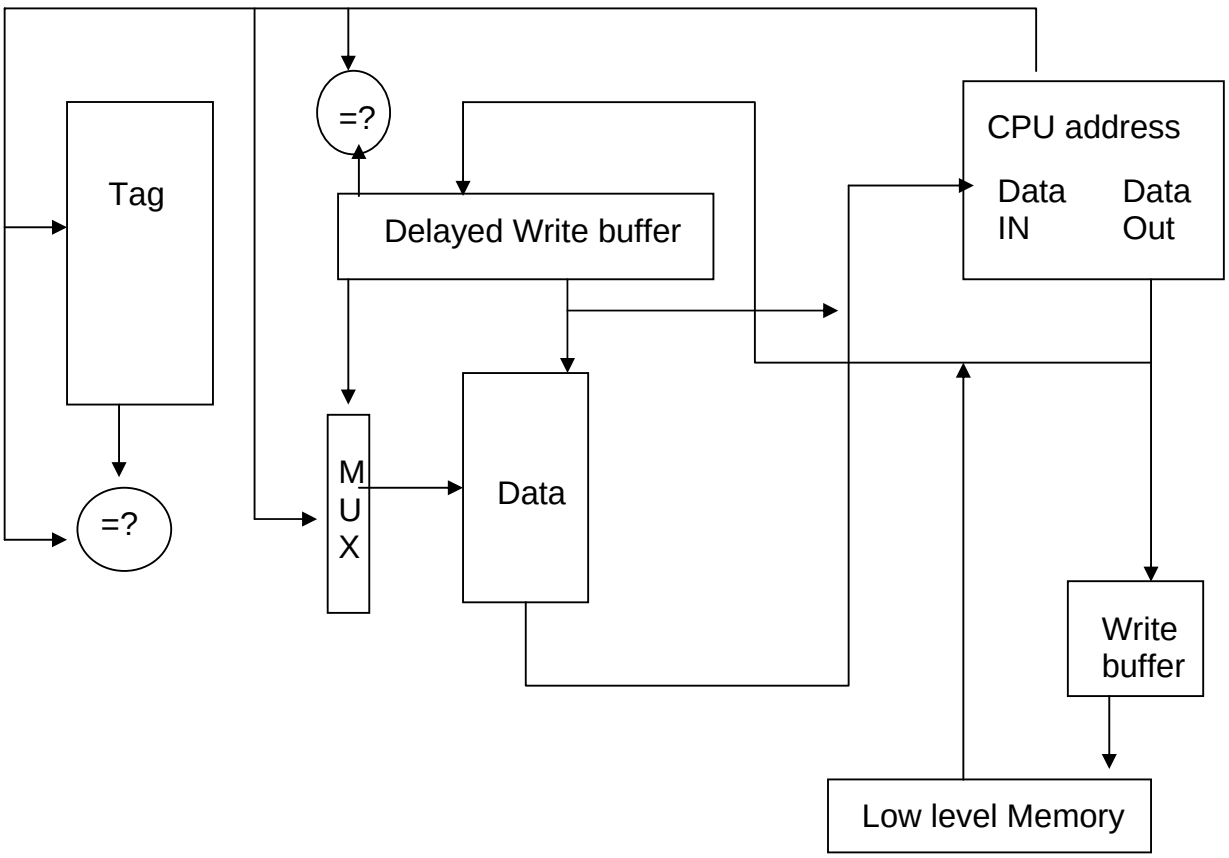


Figure The hardware organization of pipelined writes.

Cache Optimization Summary The techniques discussed above to improve miss rate, miss penalty , and hit time generally impact the other components of the average memory access time as well as the complexity of the memory hierarchy.

Technique	Miss rate	Miss penalty	Hit time
Larger block size	+	-	
Higher associatively	+		-
Victim cache	+		
Pseudo associative cache	+		
HW prefetching of instructions and data	+		
Compiler techniques to reduce cache misses	+		
Giving priority to read misses over writes		+	
Sub-block placement		+	
Early restart and critical word first		+	
Non-blocking caches		+	
Second level caches		+	
Small and simple caches	-		+
Avoiding address translation using indexing of the cache			+
Pipelining writes for fast write			+
Compiler controlled prefetching	+		

Note : Table above summaries these techniques where '+ 'meaning that the technique improves the factor, '-' meaning it hurts that factor, and blank meaning it has no impact.

Main Memory

"... the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory....Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large ."

Maurice Wilkes , Memories of a Computer Pioneer(1985)

Main memory is the next level down in the hierarchy .Main memory, satisfies the demands of cache and serves as the I\O interface, as it is the destination of input as well as the source for the output. Performance measures of memory emphasis both latency and bandwidth.Traditionally,main memory latency is the primary concern of the cache, while main memory bandwidth is the primary concern of I\O. With the popularity of second level caches and their larger block sizes, main memory bandwidth becomes important to caches as well.

REFERENCES

1. ARCHIBALD, J., AND BAER, J.-L. Cache coherence protocols: Evaluation using a multiprocessor simulation model. ACM Trans. Comput. Syst. 4, 4 (Nov. 1986), 273-298.

2. AGARWAL, A. Analysis of cache performance for operating systems and multiprogramming. Ph D. dissertation, Stanford Univ., 1987.

3. BAER, J.-L., AND SAGER, G. Dynamic improvement of locality in virtual memory systems. IEEE Trans. Softw. Eng. SE-2, 1 (Mar. 1976), 54-62.

4. BELADY, L. A. A study of replacement algorithms for virtual storage computers. IBM Syst. J. 5, 2 (1966), 78-101.

5. GECSEI, J. Determining hit ratios in multilevel hierarchies. IBM J. Res. Deu. 28,4 (July 1974), 316-327.

6. GOODMAN, J. R. Using cache memory to reduce processor-memory traffic. In Proceedings of the 10th Znternationa2 Symposium on Computer Architecture. (Stockholm, June, 1983). pp. 124-131.

7. GREENBERG, B. S. An experimental analysis of program reference patterns in the multics virtual memory. MAC Tech. Rep.-127, Cambridge, Mass., Jan. 1974.

8. HILL, M. D., AND SMITH, A. J. Experimental evaluation of on-chip microprocessor cache memories. In Proceedings of the 11 th Annual Symposium on Computer Architecture (Ann Arbor, Mich., June, 1984). pp. 158-166.

9. HILL, M. Aspects of cache memory and instruction buffer performance. Ph.D. dissertation, Univ. of California, Berkeley, 1987

10. HILL, M. D. AND SMITH, A. J. Evaluating associativity in cpu caches. IEEE Trans. Comput. 38, 12 (Dec. 1989), 1612-1630.

11. IEEE P896.1. Draft Standard, Backplane Bus (Futurebus). Nov. 1986. 17. KATZ, R., EGGERS, S., WOOD, D. A., PERKINS, C., AND SHELDON, R. G. Implementing a cache consistency protocol. In Proceedings of the 12th International Symposium on Computer Architecture (June 1985). pp. 276-283.

12. LAHA, S., PATEL, J. H., AND IYER, R. K. Accurate low-cost methods for performance evaluation of cache memory systems. IEEE Trans. Comput. 37, 11 (Nov. 1988), 1325-1336.

13. LIN, Y-B., BAER, J.-L., AND LAZOWSKA, E. D. Tailoring a parallel trace-driven simulation technique to specific multiprocessor cache coherence protocols. In Distributed Simulation Proceedings of the 1989 SCS Eastern Conference (Tampa, Fla., March 1989), pp. 185-190.

14. LIPTAY, J. S. Structural aspects of the system/360 Model 85, part II: The cache. IBM Syst. J. 7, 1 (1968), 15-21.

15. PUZAK, T. R. Cache-Memory Design Ph.D. dissertation, Univ. of Massachusetts, 1985.

16. MATTSON, R. L., GECSEI, J., SLUTZ, D., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. IBM Syst. J. 9, 2 (1970), 78-117.

17. OLKEN, F. Efficient methods for calculating the success function of fixed space replacement policies. Master's thesis, Univ. of California, Berkeley, Calif., May 1981.

18. SLUTZ, D. R., AND TRAIGER, I. L. Determination of hit ratios for a class of staging hierarchies. IBM Res. Rep. RJ 1044, May 1972.

19. SMITH, A. J. Two methods for the efficient analysis of memory trace data. IEEE Trans. Softw. Eng. SE-3, 1 (Jan. 1977), 94-101.

20. SMITH, A. J. Sequential program prefetching in memory hierarchies. Computer 11, 2 (Dec. 1978), 7-21.

21. SMITH, A. J. Cache memories. ACM Comput. Suru. 14,3 (Sept. 1982), 473-530.

22. SMITH, A. J. Cache evaluation and the impact of workload choice. In Proceedings of the 12th Annual Symposium on Computer Architecture (Boston, Mass., June 1985). pp. 64-73. 34. SMITH, A. J. Disk cache-Miss ratio analysis and design considerations.

23. SMITH, A. J. Two methods for the efficient analysis of memory address trace data. IEEE Trans. Softw. Eng. 3, 1 (Jan. 1977), 94-101.

24. SWEAZEY, P., AND SMITH A. J. A class of compatible cache consistency protocols and their support by the IEEE future bus, In Proceedings of the 13th Symposium on Computer Architecture (Tokyo, June 1986), pp. 414-423.
ACM Trans. Comput. Syst., Aug. 1985, 161-203.

25. THOMPSON, J. Efficient analysis of caching systems. Ph.D. dissertation, Univ. of California, Berkeley; also Tech. Rep. UCB/CSD 87/3'74, Oct. 1987.

26. THOMPSON, J. G., AND SMITH, A. J. Efficient (stack) algorithms for analysis of write-back and sector memories. ACM Trans. Comput. Syst. 7, 1 (Feb. 1989), 78-116.

27. TOKUNAGA, T., HIRAI, Y., AND YAMAMOTO, S. Integrated disk cache systems with file adaptive control. In Proceedings of the IEEE Computer Society Conference (Washington, DC, Sept. 1980). IEEE, New York, 1980, pp. 412-416.
28. TRAIGER, I. L., AND SLUTZ, D. R. One pass techniques for the evaluation of memory hierarchies. IBM Res. Rep. RJ 892, Yorktown Heights, N.Y., July 1971.