Cache Memories

ALAN JAY SMITH

University of California, Berkeley, California 94720

Cache memories are used in modern, medium and high-speed CPUs to hold temporarily those portions of the contents of main memory which are (believed to be) currently in use. Since instructions and data in cache memories can usually be referenced in 10 to 25 percent of the time required to access main memory, cache memories permit the execution rate of the machine to be substantially increased. In order to function effectively, cache memories must be carefully designed and implemented. In this paper, we explain the various aspects of cache memories and discuss in some detail the design features and trade-offs. A large number of original, trace-driven simulation results are presented. Consideration is given to practical implementation questions as well as to more abstract design issues.

Specific aspects of cache memories that are investigated include: the cache fetch algorithm (demand versus prefetch), the placement and replacement algorithms, line size, store-through versus copy-back updating of main memory, cold-start versus warm-start miss ratios, multicache consistency, the effect of input/output through the cache, the behavior of split data/instruction caches, and cache size. Our discussion includes other aspects of memory system architecture, including translation lookaside buffers. Throughout the paper, we use as examples the implementation of the cache in the Amdahl 470V/6 and 470V/7, the IBM 3081, 3033, and 370/168, and the DEC VAX 11/780. An extensive bibliography is provided.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids; C.O. [**Computer Systems Organization**]: General; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Buffer memory, paging, prefetching, TLB, store-through, Amdahl 470, IBM 3033, BIAS

INTRODUCTION

Definition and Rationale

Cache memories are small, high-speed buffer memories used in modern computer systems to hold temporarily those portions of the contents of main memory which are (believed to be) currently in use. Information located in cache memory may be accessed in much less time than that located in main memory (for reasons discussed throughout this paper). Thus, a central processing unit (CPU) with a cache memory needs to spend far less time waiting for instructions and operands to be fetched and/or stored. For example, in typical large, high-speed computers (e.g., Amdahl 470V/7, IBM 3033), main memory can be accessed in 300 to 600 nanoseconds; information can be obtained from a cache, on the other hand, in 50 to 100 nanoseconds. Since the performance of such machines is already limited in instruction execution rate by cache memory access time, the absence of any cache memory at all would produce a very substantial decrease in execution speed.

Virtually all modern large computer sys-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0010-4892/82/0900-0473 $00.75

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

CONTENTS

INTRODUCTION
Definition and Rationale
Overview of Cache Design
Cache Aspects
1. DATA AND MEASUREMENTS
1.1 Rationale
1.2 Trace-Driven Simulation
1.3 Simulation Evaluation
1.4 The Traces
1.5 Simulation Methods
2 ASPECTS OF CACHE DESIGN AND OPERATION
2.1 Cache Fetch Algorithm
2.2 Placement Algorithm
2.3 Line Size
2.4 Replacement Algorithm
2.5 Write-Through versus Copy-Back
2.6 Effect of Multiprogramming Cold-Start and Warm-Start
2.7 Multicache Consistency
2.8 Data/Instruction Cache
2.9 Virtual Address Cache
2.10 User/Supervisor Cache
2.11 Input/Output through the Cache
2.12 Cache Size
2.13 Cache Bandwidth, Data Path Width, and Access Resolution
2.14 Multilevel Cache
2.15 Pipelining
2.16 Translation Lookaside Buffer
2.17 Translator
2.18 Memory-Based Cache
2.19 Specialized Caches and Cache Components
3 DIRECTIONS FOR RESEARCH AND DEVELOPMENT
3.1 On-Chip Cache and Other Technology Advances
3.2 Multicache Consistency
3.3 Implementation Evaluation
3.4 Hit Ratio versus Size
3.5 TLB Design
3.6 Cache Parameters versus Architecture and Workload
APPENDIX EXPLANATION OF TRACE NAMES
ACKNOWLEDGMENTS
REFERENCES

tems have cache memories; for example, the Amdahl 470, the IBM 3081 [IBM82, REIL82, GUST82], 3033, 370/168, 360/195, the Univac 1100/80, and the Honeywell 66/80. Also, many medium and small size machines have cache memories; for example, the DEC VAX 11/780, 11/750 [ARMS81], and PDP-11/70 [STRE76, SNOW78], and the Apollo, which uses a Motorolla 68000 two to four years, circuit speed and density will progress sufficiently to permit cache memories in one chip microcomputers. (On-chip addressable memory is planned for the Texas Instruments 99000 [LAFF81, ELEC81].) Even microcomputers could benefit substantially from an on-chip cache, since on-chip access times are much smaller than off-chip access times. Thus, the material presented in this paper should be relevant to almost the full range of computer architecture implementations.

The success of cache memories has been explained by reference to the "property of locality" [DENN72]. The property of locality has two aspects, temporal and spatial. Over short periods of time, a program distributes its memory references nonuniformly over its address space, and which portions of the address space are favored remain largely the same for long periods of time. This first property, called temporal locality, or locality by time, means that the information which will be in use in the near future is likely to be in use already. This type of behavior can be expected from program loops in which both data and instructions are reused. The second property, locality by space, means that portions of the address space which are in use generally consist of a fairly small number of individually contiguous segments of that address space. Locality by space, then, means that the loci of reference of the program in the near future are likely to be near the current loci of reference. This type of behavior can be expected from common knowledge of programs: related data items (variables, arrays) are usually stored together, and instructions are mostly executed sequentially. Since the cache memory buffers segments of information that have been recently used, the property of locality implies that needed information is also likely to be found in the cache.

Optimizing the design of a cache memory generally has four aspects:

(1) Maximizing the probability of finding a memory reference's target in the cache (the hit ratio),
(2) minimizing the time to access information that is indeed in the cache (access time),

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

(4) minimizing the overheads of updating main memory, maintaining multicache consistency, etc.

(All of these have to be accomplished under suitable cost constraints, of course.) There is also a trade-off between hit ratio and access time. This trade-off has not been sufficiently stressed in the literature and it is one of our major concerns in this paper. In this paper, each aspect of cache memories is discussed at length and, where available, measurement results are presented. In order for these detailed discussions to be meaningful, a familiarity with many of the aspects of cache design is required. In the remainder of this section, we explain the operation of a typical cache memory, and then we briefly discuss several aspects of cache memory design. These discussions are expanded upon in Section 2. At the end of this paper, there is an extensive bibliography in which we have attempted to cite all relevant literature. Not all of the items in the bibliography are referenced in the paper, although we have referred to items there as appropriate. The reader may wish in particular to refer to BADE79, BARS72, GIBS67, and KAPL73 for other surveys of some aspects of cache design. CLAR81 is particularly interesting as it discusses the design details of a real cache. (See also LAMP80.)

Overview of Cache Design

Many CPUs can be partitioned, conceptually and sometimes physically, into three parts: the I-unit, the E-unit, and the S-unit. The I-unit (instruction) is responsible for instruction fetch and decode. It may have some local buffers for lookahead prefetching of instructions. The E-unit (execution) does most of what is commonly referred to as *executing* an instruction, and it contains the logic for arithmetic and logical operations. The S-unit (storage) provides the memory interface between the I-unit and E-unit. (IBM calls the S-unit the PSCF, or processor storage control function.)

The S-unit is the part of the CPU of primary interest in this paper. It contains several parts or functions, some of which are shown in Figure 1. The major compo-

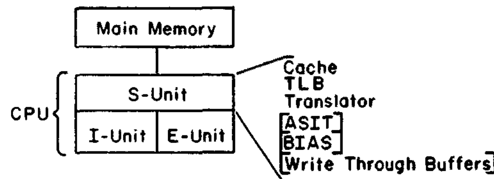

Figure 1. A typical CPU design and the S-unit.

There is usually a translator, which translates virtual to real memory addresses, and a TLB (translation lookaside buffer) which buffers (caches) recently generated ⟨virtual address, real address⟩ pairs. Depending on machine design, there can be an ASIT (address space identifier table), a BIAS (buffer invalidation address stack), and some write-through buffers. Each of these is discussed in later sections of this paper.

Figure 2 is a diagram of portions of a typical S-unit, showing only the more important parts and data paths, in particular the cache and the TLB. This design is typical of that used by IBM (in the 370/168 and 3033) and by Amdahl (in the 470 series). Figure 3 is a flowchart that corresponds to the operation of the design in Figure 2. A discussion of this flowchart follows.

The operation of the cache commences with the arrival of a virtual address, generally from the CPU, and the appropriate control signal. The virtual address is passed to both the TLB and the cache storage. The TLB is a small associative memory which maps virtual to real addresses. It is often organized as shown, as a number of groups (sets) of elements, each consisting of a virtual address and a real address. The TLB accepts the virtual page number, randomizes it, and uses that hashed number to select a set of elements. That set of elements is then searched associatively for a match to the virtual address. If a match is found, the corresponding real address is passed along to the comparator to determine whether the target line is in the cache. Finally, the replacement status of each entry in the TLB set is updated.

If the TLB does not contain the ⟨virtual address, real address⟩ pair needed for the translation, then the translator (not shown in Figure 2) is invoked. It uses the high-order bits of the virtual address as an entry

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

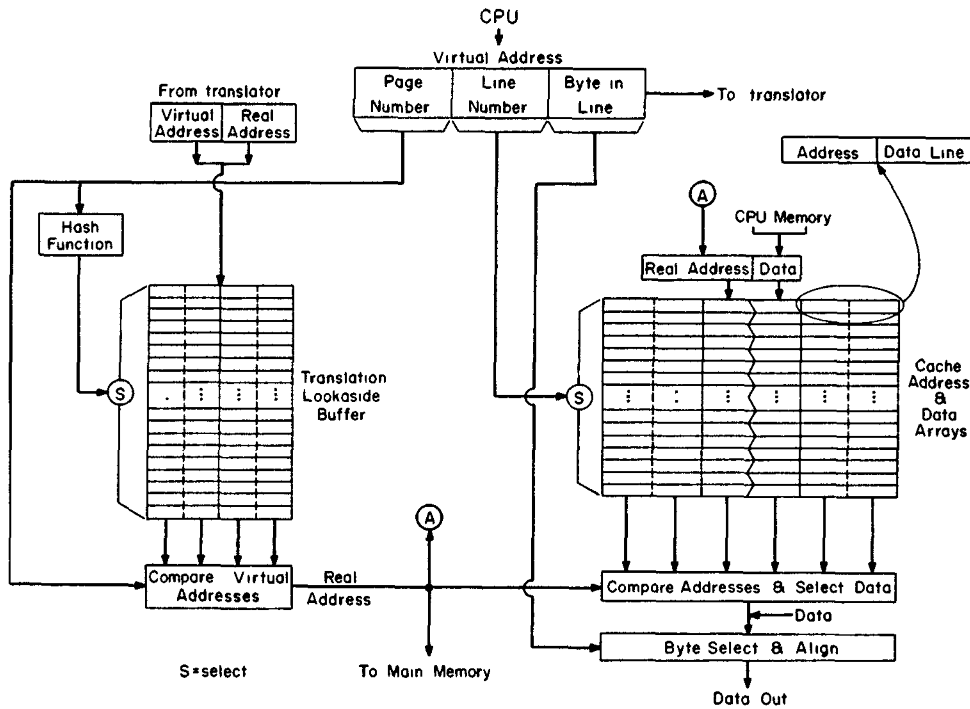

Figure 2. A typical cache and TLB design.

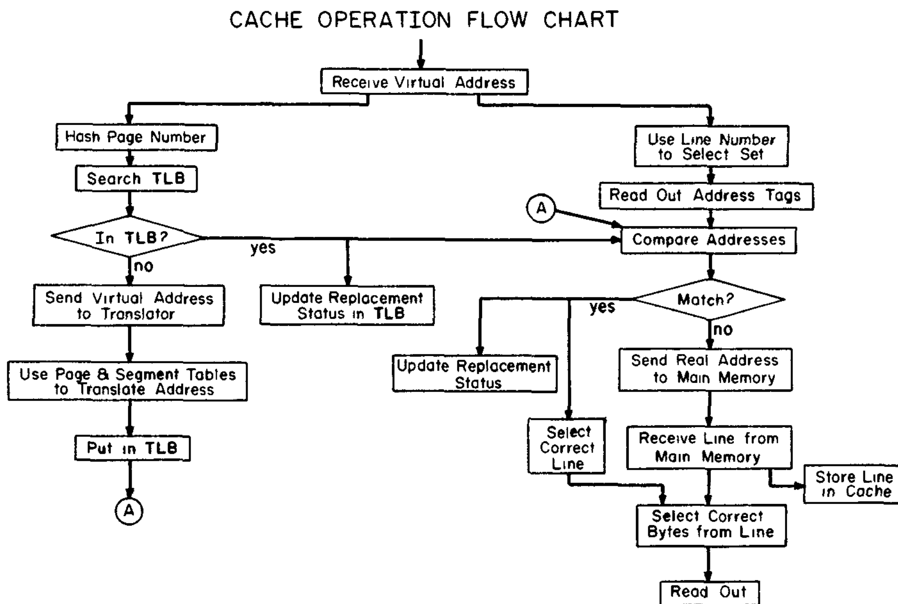

Figure 3. Cache operation flow chart.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

process and then returns the address pair to the TLB (which retains it for possible future use), thus replacing an existing TLB entry.

The virtual address is also passed along initially to a mechanism which uses the middle part of the virtual address (the line number) as an index to select a set of entries in the cache. Each entry consists primarily of a real address tag and a line of data (see Figure 4). The line is the quantum of storage in the cache. The tags of the elements of all the selected set are read into a comparator and compared with the real address from the TLB. (Sometimes the cache storage stores the data and address tags together, as shown in Figures 2 and 4. Other times, the address tags and data are stored separately in the "address array" and "data array," respectively.) If a match is found, the line (or a part of it) containing the target locations is read into a shift register and the replacement status of the entries in the cache set are updated. The shift register is then shifted to select the target bytes, which are in turn transmitted to the source of the original data request.

If a miss occurs (i.e., addresss tags in the cache do not match), then the real address of the desired line is transmitted to the main memory. The replacement status information is used to determine which line to remove from the cache to make room for the target line. If the line to be removed from the cache has been modified, and main memory has not yet been updated with the modification, then the line is copied back to main memory; otherwise, it is simply deleted from the cache. After some number of machine cycles, the target line arrives from main memory and is loaded into the cache storage. The line is also passed to the shift register for the target bytes to be selected.

Cache Aspects

The cache description given above is both simplified and specific; it does not show design alternatives. Below, we point out some of the design alternatives for the cache memory.

Cache Fetch Algorithm. The cache fetch algorithm is used to decide when to bring information into the cache. Several possibilities exist: information can be fetched on demand (when it is needed) or prefetched (before it is needed). Prefetch algorithms attempt to guess what information will soon be needed and obtain it in advance. It is also possible for the cache fetch algorithm to omit fetching some information (selective fetch) and designate some information, such as shared writeable code (semaphores), as unfetchable. Further, there may be no fetch-on-write in systems which use write-through (see below).

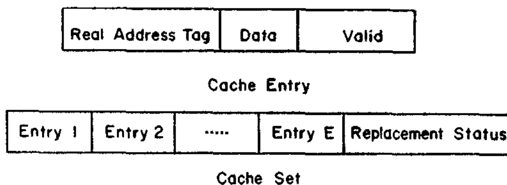

Figure 4. Structure of cache entry and cache set.

Cache Placement Algorithm. Information is generally retrieved from the cache associatively, and because large associative memories are usually very expensive and somewhat slow, the cache is generally organized as a group of smaller associative memories. Thus, only one of the associative memories has to be searched to determine whether the desired information is located in the cache. Each such (small) associative memory is called a set and the number of elements over which the associative search is conducted is called the set size. The placement algorithm is used to determine in which set a piece (line) of information will be placed. Later in this paper we consider the problem of selecting the number of sets, the set size, and the placement algorithm in such a set-associative memory.

Line Size. The fixed-size unit of information transfer between the cache and main memory is called the line. The line corresponds conceptually to the page, which is the unit of transfer between the main memory and secondary storage. Selecting the line size is an important part of the memory system design. (A line is also sometimes referred to as a *block*.)

Replacement Algorithm. When information is requested by the CPU from main memory and the cache is full, some information in the cache must be selected for

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.