CHAPTER 6

Memory Systems

6.1 Introduction

The function of a computer's memory system is simple—writing, storing, and reading data stored at addressable locations in electronic, magnetic, or other media. Nevertheless, in modern practice, the implementation complexity of memory systems rivals that of processing. Furthermore, the resources devoted to memory systems dwarfs those spent on processing. Even most of the transistors used in modern microprocessors are memory system components: registers, translation lookaside buffers (TLBs), one or two level-one caches and, increasingly, level-two cache tags and/or data. Burks, Goldstein, and von Neumann foresaw this trend in 1946 [2]:

> Ideally one would desire an indefinitely large memory such that any particular . . . word would be immediately available—i.e., in a time which is somewhat or considerably shorter than the operation time of a fast electronic multiplier. This may be assumed to be practical at the level of about 100 μsec. Hence the availability time for a word in memory should be 5 to 50 μsec. It is equally desirable that words may be replaced with new words at about the same rate. It does not seem possible physically to achieve such capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

These words remain prophetic half a century later as gigahertz superscalar processors read and write words at a rate seven orders of magnitude faster than Burks et al. considered. What has changed is our understanding of how to engineer memory hierarchies.

In theory, memory hierarchies can be characterized by Hennessy and Patterson's [7] four questions that assume the "upper" level is caching the "lower" level and data are moved between them in "blocks":

- Where can a block be placed in the upper level? (Block placement)
- How is a block found if it is in the upper level? (Block identification)
- Which blocks should be replaced on a miss? (Block replacement)
- What happens on a write? (Write strategy)

In practice, memory hierarchy design bifurcates into design issues for hardware caches and virtual memory. Other examples of what could be considered memory hierarchy design, such as register allocation, file I/O, tape archiving, and proxy server URL caching are, in practice, not considered part of the memory system.

For these reasons, this chapter will concentrate on the development of hardware caches, virtual memory, and then some interactions between them. We assume that readers are familiar with caches and virtual memory through the reading of textbooks such as Hennessy and Patterson [7, ch. 5]. Therefore, we will not devote space to introducing the basic concepts. Some readers many also notice that our selection of readings does not include some of the great survey articles that taught many people about caches and virtual memory. Notable among these are Smith's 1982 cache survey [17] and Denning's 1970 virtual memory survey [5]. We instead concentrate on the primary sources that have developed the field. Nevertheless, Smith and Denning's surveys are good reading for those interested in seeing historical snapshots of the state of the art.

Even though this chapter focus on architectural issues, it is important to remember the fundamental role of the technologies of memory. Modern computers are not possible without economical large memories. The first technology to meet the challenge was the ferrite core memory developed in the late 1940s by MIT's Project Whirlwind. Core used a two-dimensional array of magnetic donuts with row and column control signals that constructively interacted only at the addressed bit. Core was eventually supplanted by a second memory technology still in unchallenged dominance today: semiconductor memory. Furthermore, the characteristics of semiconductor memories deployed continues to change rapidly as packages and protocols evolve (e.g., SDRAMs, Rambus, and integrated circuits that combine logic and dynamic RAM [16]). These economical memory technologies—together with non-volatile magnetic storage (e.g., disks)—are arguably more critical to economical computers than the comparatively few transistors used for processing.

6.2 Cache Origins

A hardware cache is an application of memory hierarchy principles yielding a hardware upper level that caches information from main memory (or a lower level cache) in a manner functionally transparent to software. Block placement, block identification, block replacement, and write strategy are handled by hardware. Block placement is usually limited to much less than the whole cache (direct-mapped or set-associative) to ease the implementation of associative block identification. Blocks are identified by explicitly storing either a main memory address (physical address) or a program address (virtual address). Block replacement uses simple hardware algorithms, such as least-recently used (LRU), pseudo-LRU, or random. Writes either update memory on replacements (write-back) or on all writes (write-through). Write misses may or may not allocate new cache blocks.

6.2.1 Wilkes's "Slave Memories and Dynamic Storage Allocation" [23]

Maurice Wilkes invented the cache with the seminal observation that Gordon Scarrott's instruction buffering ideas could be applied to data as well as instructions.[1] Wilkes presents his ideas for data caches ("slave memory") in a 1.5-page 1965 paper. The paper includes three direct-mapped cache designs that are not important in and of themselves. What is important is that the paper brought out the potential of caching to the world, in general, and the IBM System/360 Model 85 designers, in particular.

6.2.2 Liptay's "Structural Aspects of the System/360 Model 85, Part II: The Cache" [12]

The first commercial realization of the cache concept came just 3 years later in the IBM System/360 Model 85 [12]. The Model 85's cache was so effective that it derailed interest in the much more complex out-of-order microarchitectural techniques of the cache-less Model 91 (e.g., Tomasulo's Algorithm) [1].

Liptay's paper is important, because it describes the first commercial cache and the first cache performance studies. In modern terms, the Model 85's 16 Kbyte cache is divided into 16 fully associative 1 Kbyte blocks (called "sectors" in the paper). Large blocks reduce the amount of expensive tag logic. Each block is further

[1]Personal communication with Maurice V. Wilkes on 10 August 1998. Gordon Scarrott was with International Computers and Tabulators, the major British computer company of the period.

subdivided into sixteen 64-byte subblocks (called "blocks" in the paper) with per-subblock valid bits. A block hit and subblock miss loads a 64-byte subblock and sets its valid bit. A block miss replaces the LRU block and performs a subblock miss. The cache uses write-through with no write allocation, so block replacement is trivial.

Performance evaluations used trace-driven simulation to find an average hit ratio of 96.8% and mean performance relative to an ideal system of 81%. Also studied were alternative caches of larger size, larger block size, more limited associativity, and a pseudo-LRU replacement algorithm. This work was very impressive for appearing only 3 years after the first cache paper.

6.3 Cache Advances

Of the hundreds of memory system papers that have been written, it is hard to select just a few. These next three represent key qualitative advances in different dimensions. Kroft introduces lockup-free or non-blocking caches. Goodman outlines a snooping cache coherence protocol. Jouppi gives some techniques made practical as custom VLSI replaces discrete RAM chips at the fast end of the memory hierarchy.

6.3.1 Kroft's "Lockup-Free Instruction Fetch/Prefetch Cache Organization" [11]

Until the 1990s, a cache miss stalled the processor until the miss was handled. This approach was so common it was usually an unstated assumption. For two reasons, handling one miss at time—called a *blocking* or *lockup* cache—becomes problematic when cache miss occurences become very large relative to the number of cache hits. First, the processor and cache cannot hide the latency of a cache miss by overlapping it with other work. Second, the memory bandwidth available is limited to the cache block size divided by the cache miss latency [18]. On the other hand, building an n-way *non-blocking* or *lockup-free cache* that allows up to n outstanding misses provides both latency tolerance and memory bandwidth up to n times that of a blocking cache. Of course, a nonblocking cache is useless unless its processor is capable of doing useful work, usually including memory references, while one or more misses are outstanding.

Kroft presented the first design for a nonblocking cache in 1981, a full decade before common commercial use. The paper focuses on specific details of one design, but this design has served as a basis for many commercial designs that followed. Key are *n miss information/status holding* (MSHR) registers. Each MSHR register holds information pertinent to one outstanding miss. On a cache reference, MSHRs are searched concurrently with the main cache to suppress a main memory reference on a cache miss, but MSHR hit. Specific MSHR state is needed to keep track of the block's address, where the block will go into the cache, how much of the block has been returned from memory so far, which words have outstanding reads (including transaction identifiers), and which words have been (partially or totally) overwritten by stores. Kroft reports, without analysis, that four MSHR registers can be supported with a 10% increase to overall cache cost. Today, most caches are nonblocking, and we expect the trend toward nonblocking to continue.

6.3.2 Goodman's "Using Cache Memory to Reduce Processor-Memory Traffic" [6]

Entering the 1980s, caches were prized for reducing effective memory latency in uniprocessors. Caches were also used in small-way multiprocessors (e.g., two-way systems from IBM) where coherence was maintained by generating "cross invalidations" on every write. There were also complex (in late 1970s technology) schemes for using a centralized directory to reduce the negative effects of frequent invalidations [21].

Goodman's paper entered with two contributions. First, he observed that caches can be used to reduce bandwidth (as well as latency). This observation is typical of influential ideas. It changed the way people think sufficiently that it now seems too obvious to be a contribution. Goodman noted that bandwidth is reduced more by using write-back instead of write-through. A write-back cache with B-word blocks, a miss ratio m, a fraction d block dirty at replacement time multiplies the bandwidth required from/to memory by a factor of $m \times B \times (1 + d)$. The factor is much less than 1.0 for reasonable miss ratios. In addition, Goodman advocated using subblocking so that small subblocks could be transferred on misses (transfer blocks), whereas address tags are associated with larger blocks (address blocks).

Goodman's caches could reduce bandwidth requirements enough that a multiprocessor could use a bus as an interconnect, provided the cache coherence problem could be simply solved for write-back caches. Goodman's second contribution was to solve this problem with the *write-once* protocol, the first snooping coherence protocol. On the first write to a block, the write-once protocol performs a write-through that also invalidates other cached copies. On subsequent writes, write-back is used. Write-once spawned a series of alternative snooping protocols that were then unified with Sweazey and Smith's MOESI framework [19]. Today, snooping on a bus is considered a solved research problem and is a solution used in all but the largest commercial multiprocessors.

6.3.3 Jouppi's "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers" [9]

Memory system performance is frequently improved by devoting more resources to larger or more levels of traditional caches. Alternatively, one could spend some of those resources on special buffers and additional logic to accelerate traditional caches. This alternative does not work so well at the board level, because the chips for small special buffers could use as much board area as that of a traditional cache. The special buffer alternative is much more viable in future systems, where many caches are implemented in custom logic on microprocessor chips. Here the die area of special buffers is roughly proportion to special buffer capacity. Thus, adding 1-Kbyte buffer (plus logic for using it) next to a 64-Kbyte cache might increase die area only 1–3%. So what special buffers are useful? Jouppi gives some initial answers, but, we believe, researchers of the future will find many more.

Specifically, Jouppi proposes two types of special buffers: *victim caches* and *stream buffers* (and *miss caches* that are obsoleted by victim caches). A victim cache is a small, fully associative buffer behind a cache that holds recently replaced blocks and can provide them much faster than memory can (or the next level of cache). A victim cache reduces the negative effect of conflict misses by providing conflicting blocks faster than misses to memory can. A stream buffer prefetches blocks from memory into the stream buffer at addresses sequentially after a miss. Blocks are moved from the stream buffer into the cache only if they are actually referenced. Multiple stream buffers can prefetch after multiple misses. Stream buffers can be extended to nonunit stride prefetching [13]. Thus, stream buffers reduce both capacity and compulsory misses.

6.3.4 Cache Directions

Three other major cache trends are also apparent. First, one level of cache is increasingly being replaced by two levels. This change is occurring because memory is getting much slower relative to processors, causing caches to spend more time missing. One option was to make caches larger so they would miss less often. Very large caches, however, could slow down the common case of a cache hit. Thus, designers searched for another way to reduce the miss time. A solution was to add a level-two cache to exploit locality in level-one cache misses [14]. Multiple levels of caches. however, make maintaining coherence more difficult, as we will see for Wang, Baer, and Levy's paper [22].

Second, instruction cache design is getting much more complex, because modern superscalar processors can require many instructions per cycle. Furthermore, increased issues widths and branch speculation make it possible that instructions fetched in a given cycle must come from two or more basic blocks. This forced researchers toward multiported caches [24] and trace caches that contiguously cache dynamically contiguous instructions that may not be contiguous in memory [15].

Third, multiprocessor cache issues are moving well beyond simple snooping, as discussed in the shared-memory multiprocessor papers of Chapter 9.

6.4 Virtual Memory

Virtual memory is an application of memory hierarchy principles that allows main memory to cache information from magnetic disks or other secondary storage media. Virtual memory systems use a combination of operating system and hardware support to move and cache data in blocks called *pages*. Page placement is done by the operating system, usually by considering all free page frames as equivalent. Page identification begins by placing all code and data in a virtual address space that is unconcerned with the level (memory or disk) or location (physical address or disk address) at which the information actually resides. Virtual memory systems could follow caches and do page identification by explicitly storing virtual addresses with each page frame. Instead, most modern virtual memory systems use a level of indirection through a page table to translate a virtual address into main memory's physical addresses. Page tables are large and must be stored in memory. To avoid doing one (or more) extra memory references per memory reference, recent translations are cached in a special hardware cache called a *translation lookaside buffer* (TLB). Thus, the common case of page identification is performed completely in hardware by the TLB. Page replacement is handled by the operating system, usually using some approximation to LRU. Write strategy is always write-back with write-allocation.

We have selected two papers pertinent to virtual memory. Kilburn et al. describe the first implementation of virtual memory in the Manchester Atlas. Clark and Emer describe the virtual memory state of the art circa 1980.

6.4.1 Kilburn et al.'s "One-Level Storage System" [10]

Virtual memory was both first proposed and first implemented in the Manchester Altas. Kilburn et al. describe the Atlas in general and its virtual memory system in particular. Even though the general description provides a history lesson, we will concentrate on the virtual memory aspects described mostly in sections 3 and 5. What is most important is that the Altas introduced many seminal concepts: address translation, demand paging, page-fault interrupts, reference bits, and multiprogramming made more transparent by virtual memory.

Specifically, programs running on the Atlas use virtual addresses to transparently access 50-bit (four character) words actually stored in a 16-K-word main memory (ferrite core "central store") or on a 96-K-word magnetic drum. Main memory held 32 512-word pages. The virtual address for each page was stored in a "page address register" (PAR). The 32 PARs can be thought of as "cache tags" for memory's 32 pages or as 32-entry fully associative TLB that maps all memory. On a program memory reference, the 32 PARs are associatively searched. On a match ("equivalence"), memory data is returned. If there is not a match, an interrupt is generated that context-switches the processor to a page fault handler. If main memory has a free page, the handler loads the page from drum and resumes the program. Otherwise, it must first replace a page. The Altas includes reference bits (use digits) to aid page replacement but uses a replacement policy much more complex that a modern approximations to LRU (the learning program). It probably took experimental data from actual operation to inform designers that a simpler policy would suffice.

6.4.2 Clark and Emer's "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement" [9]

Clark and Emer describe and evaluate a state-of-the-art virtual memory system from the late-1970s: the memory system of DEC VAX-11/780. The VAX-11 architecture was the first widely successful 32-bit architecture designed specifically with virtual memory in mind and represents the state of the art some twenty years after the Atlas. Clark and Emer introduce how the VAX-11 page tables translate 32-bit virtual addresses without occupying too much physical memory, describe how the VAX-11/780's TLB operates, and present influential performance results from system monitoring.

VAX-11 divided a 32-bit virtual address space into three 1-Gbyte regions: system space (S0), a process space for program text and heap (P0), and process space for stack (P1). Paging was done with 512-byte pages (probably too small even for 1978). Translation information for pages was stored in a page table entry (PTE) in a linear page table. The index within a page table for a page's PTE could be obtained by selecting high-order virtual address bits. The S0 page table resided at a well-known physical address. P0 and P1 are process specific and reside at a well-known address in S0 virtual space. A context switch can change the P0/P1 page tables but not the S0 page table. P0/P1 page table overhead is reduced two ways. First, P0 and P1 grow toward each other and have limit registers to avoid allocating PTEs for unallocated pages. Second, P0/P1 PTEs reside in S0 virtual space and can be paged. This means, however, that a user reference can suffer two page faults—one for the user page and one for the corresponding PTE—but that the physical memory devoted to page tables is dramatically reduced from the naive 32 Mbytes (4 bytes/PTE × 4 Gbytes / 512 bytes/PTE).

All VAX-11 implementations accelerated address translation with a TLB. The VAX-11/780's TLB contained 128 translations. It was two-way set-associative with random replacement. Half the sets were devoted to S0 translations and half to P0/P1 translations. The P0/P1 were flushed on context switches. Data references access the TLB before accessing the VAX-11/780's cache. Instruction references saved the translation of the current instruction page and bypassed the TLB on sequential instructions that did not cross a page boundary.

The paper used a combination of system monitoring and trace-driven simulation to evaluate the VAX-11/780's TLB. Results were so influential that they are now mostly well known: Operating system reference miss ratios are much higher than user miss ratios, data misses per instruction are consistently greater than instruction misses per instruction, "double" TLB misses were rare, and the VAX-11/780's TLB was adequate (adding only 0.7 cycles to the VAX-11/780's 10 cycles per instruction).

6.4.3 Virtual Memory Directions

We see three major trends in virtual memory systems. First, some TLBs must translate enormous regions of memory; for example, to avoid TLB misses on accesses to multiple-gigabyte database buffer pools. Moreover, TLBs are now on microprocessor chips that must be deployed in both personal computers and large servers. A "one-size-fits-all" TLB is often overkill for a personal computer and is still inadequate for a large server. One answer is to augment the virtual memory system to also support large, aligned superpages (e.g., 4 Mbytes). Superpages, however, complicate the operating system and the TLB and are not (yet) easy to use in general [20].

Second, commercial microprocessors are making the transition from 32-bit to 64-bit virtual addresses. Larger addresses make it more difficult to do translation quickly. To make matters worse, many modern language systems wish to do dynamic memory allocation sparsely distributed about the virtual address space. These trends make more desirable page table designs that perform address translation via hashing physical addresses [3, 8]. This is because these *inverted* or *hashed* page tables occupy memory in proportion to physical memory size, not virtual address space size.

Third, the interactions between caches and TLBs is becoming more difficult as we discuss next.

6.5 Cache and Virtual Memory Interactions

Conventionally, address translation is performed before cache access, making the cache a *physical* cache. Even so, address translation can be implemented in parallel with the first part of a cache access if the cache can be indexed with bits within the page offset (that do not change in translation). This requires that the cache size not exceed the page size times the cache's associativity. Today, however, we often want level-one caches that are larger than a typical page size (4 Kbytes or 8 Kbytes) times the associativity (1–4). These physical caches require address translation to complete before the cache access begins.

Alternatively, researchers have long argued for virtual caches that are indexed and tagged with virtual addresses [17]. Virtual caches do not require address translation on cache hits. Virtual caches, however, have not been widely deployed because of issues of synonyms and context switch requirements. *Synonyms* are two different virtual pages that map to the same physical page. Context switches force a virtual cache to deal with the same virtual page (from different contexts) mapping to different physical pages.

In our view, virtual caches may become more important in future systems. Solutions exist to the synonym and context switching problems (e.g., as discussed next by Wang et al. [22]). Today's superscalar processors can perform many instructions per cycle. Tomorrow's will do many memory references per cycle. Performing multiple address translations per cycle may prove complex enough to make virtual cache more attractive.

6.5.1 Put It All Together: Wang, Baer, and Levy's "Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy" [22]

Wang, Baer, and Levy present a case study that nicely incorporates (1) multiple levels of cache, (2) virtual caches, and (3) cache coherence. Level-one (L1) caches are virtual to ease address translation concerns. Level-two (L2) caches are physical and always contain a superset of the blocks in the L1 caches, called *multilevel inclusion*. Cache coherence is done on physical addresses by the L2 caches. L2 cache blocks maintain pointers to L1 cache blocks to solve the synonym problem and forward pertinent coherence traffic.

The specifics of Wang et al.'s solution are as follows. Each L2 cache block includes some extra state and a pointer, the *vpointer*, indicating where the block is in the level one (if it is). (Multiple pointers are needed when the L1 block size is smaller than the L2 block size.) The pointer size is usually small, because it is $\log_2$ of the number of L1 cache block less the number of index bits that do not change in address translation. Coherence requests only go to the L1 cache for valid vpointers. Furthermore, an L1 miss that finds a vpointer to another L1 block knows it has encountered an active synonym, which it remaps or moves. L1 blocks contain a second valid bit, *sv*, which is reset on a context switch. Blocks with valid set and sv reset have valid data whose mapping needs to be reverified with the L2 cache. Thus, Wang et al.'s solution maintains three invariants: (1) An L1 block always has a corresponding L2 block that points back to it; (2) a block is present in the L1 cache at (at most) one virtual address even if synonyms exist; and (3) an L1 virtual tag is valid for this context if sv is set.

The inclusion of Wang et al.'s design cache study should not be construed as a prediction that most future memory hierarchies will follow their design. In particular, many current systems use physical level-one caches and do not maintain inclusion, especially for instruction caches.

6.6 References

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine philosophy and instruction-handling" *IBM Journal*, 11(1):8–24, Jan. 1967.

[2] A. W. Burks, H. H. Goldstine, and J. von Neumann "Preliminary discussion of the logical design of an electronic computing instrument," Tech. rep., U.S. Army Ordinance Department, 1946.

[3] A. Chang and M. F. Mergen, "801 storage: Architecture and programming," *ACM Transactions on Computer Systems*, 6(1):28–50, Feb. 1988.

[4] D. W. Clark and J. S. Emer, "Performance of the VAX-11/780 translation buffer: Simulation and measurement," *ACM Transactions on Computer Systems*, 3(1):31–62, Feb. 1985.

[5] P. J. Denning, "Virtual memory," *ACM Computing Surveys*, 2(3):153–189, Sept. 1970.

[6] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 124–131, 1983.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1996.

[8] J. Huck and J. Hays, "Architectural support for translation tables management in large address space machines," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 39–50, May 1993.

[9] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *The 17th Annual International Symposium on Computer Architecture*, pp. 364–373, May 1990.

[10] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," *IRE Transactions*, EC-11(2):223–235, 1962.

[11] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," *Proceedings of the 8th Annual Symposium on Computer Architecture*, pp. 81–87, May 1981.

[12] J. S. Liptay, "Structural aspects of the system/360 Model 85, part II: The cache," *IBM Systems Journal*, 7(1):15–21, 1968.

[13] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 24–33, Apr. 1994.

[14] S. Przybylski, M. Horowitz, and J. Hennessy, "Characteristics of performance-optimal multi-level cache hierarchies," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 114–121, June 1989.

[15] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," *Proceedings of the International Symposium on Microarchitecture*, pp. 24–34, Dec. 1996.

[16] K. Sakamura, "Special issue on advanced DRAM technology," *IEEE Micro*, 17(6), 1997.

[17] A. J. Smith, "Cache memories," *ACM Computing Surveys*, 14(3):473–530, 1982.

[18] G. Sohi and M. Franklin, "High-bandwidth data memory systems for superscalar processors," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 53–62, Santa Clara, CA, 1991.

[19] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE futurebus," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 414–423, June 1986.

[20] M. Talluri and M. D. Hill, "Surpassing the TLB performance of superpages with less operating system support," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 171–182, San Jose, CA, 1994.

[21] C. K. Tang, "Cache system design in the tightly compled multiprocessor system," *Proceedings of the AFIPS National Computing Conference*, pp. 749–753, June 1976.

[22] W.-H. Wang, J.-L. Baer, and H. M. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 140–148, June 1989.

[23] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, 1965.

[24] T.-Y. Yeh, D. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and branch address cache," *Proceedings of the 1993 ACM International Conference on Supercomputing*, pp. 51–61, July 1993.