Architectural Tradeoffs in the Design of MIPS-X

Paul Chow and Mark Horowitz

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

The design of a RISC processor requires a careful analysis of the tradeoffs that can be made between hardware complexity and software. As new generations of processors are built to take advantage of more advanced technologies, new and different tradeoffs must be considered. We examine the design of a second generation VLSI RISC processor, MIPS-X.

MIPS-X is the successor to the MIPS project at Stanford University and like MIPS, it is a single-chip 32-bit VLSI processor that uses a simplified instruction set, pipelining and a software code reorganizer. However, in the quest for higher performance, MIPS-X uses a deeper pipeline, a much simpler instruction set and achieves the goal of single cycle execution using a 2-phase, 20 MHz clock. This has necessitated the inclusion of an on-chip instruction cache and careful consideration of the control of the machine. Many tradeoffs were made during the design of MIPS-X and this paper examines several key areas. They are: the organization of the on-chip instruction cache, the coprocessor interface, branches and the resulting branch delay, and exception handling. For each issue we present the most promising alternatives considered for MIPS-X and the approach finally selected. Working parts have been received and this gives us a firm basis upon which to evaluate the success of our design.

Introduction

The first generation reduced instruction set processors (IBM 801[1], RISC[2,3] and MIPS[4,5]) have shown the importance of making the correct tradeoffs across the boundary that separates hardware complexity and software functionality. Hardware should only be used to support features that clearly improve performance. As implementation technology improves, new features can be considered and new tradeoffs must be made.

The goal of the MIPS-X project was to combine a new technology, a 2μm, 2-level metal CMOS process, with the knowledge and experience gained from the first generation RISC machines, to build a single processor with a peak rate of 20 MIPS and then to use 6-10 of these processors as the nodes in a shared memory multiprocessor. The resulting machine would be about two orders of magnitude more powerful than a VAX 11/780 minicomputer.

We describe here the design of the single processor, MIPS-X. The overriding principle was to keep the design as simple as possible. The original MIPS team was heavily involved in the initial architectural discussions, and they helped steer MIPS-X away from the kinds of trouble that they faced with MIPS. The major areas of concern were control related, of which the most important were considered to be instruction decode and exception handling. Both were not considered early enough in the MIPS design and created difficult implementation problems in the final chip.

The design of the instruction format was straightforward since we religiously adhered to a maxim given in the first working document on MIPS-X. It stated, "The goal of any instruction format should be:

1. Simple decode,
2. simple decode, and
3. simple decode.

Any attempts at improved code density at the expense of CPU performance should be ridiculed at every opportunity." Needless to say, all instruction sets considered for MIPS-X were fixed format 32-bit words and the amount of decoding was minimal. The effects of having this simple instruction format is discussed in the conclusions.

Not all areas were as stable as the instruction decode. Before presenting the major tradeoffs we made in the MIPS-X design, the next section describes the basic architecture of the processor and the following section gives an overview of the hardware and organization of the machine. This is followed by several sections, each discussing a major design issue in MIPS-X, the solution used and the rational for that decision.

MIPS-X Architecture

The goal of the MIPS-X project was to design a microprocessor with an order of magnitude more performance than the original MIPS processor. MIPS-X borrows heavily from the original MIPS design; it is again a heavily pipelined machine, and the resulting pipeline interlocks are handled by the supporting software system. MIPS-X differs from MIPS in that it aims for single-cycle execution using a much faster clock (20 MHz), a deeper pipeline and better implementation technology.

The high instruction rate means that memory bandwidth is an important consideration. At the projected clock frequency

of 20 MHz it is very difficult to satisfy instruction and data fetch requirements across the available package pins. To alleviate this problem, MIPS-X has a 2K-byte, on-chip instruction cache (Icache). Only instructions that miss in the Icache pass through the package pins. The Icache is placed above the datapath, in the area of the chip that is normally used for microcode storage and processor control. Data references and instruction references that miss in the Icache are handled by a large 64K word external cache (Ecache). The Ecache uses a shared bus to communicate with main memory. An added benefit of this two-level cache is that it provides a second port to memory; the processor can fetch an instruction from the Icache at the same time it is accessing off-chip data.

A deep pipeline is used to allow the machine to start a new instruction every cycle. Each instruction is divided into five pipeline stages. They are described in Figure 1. All control is hardwired.

IF	Instruction fetch.
RF	Instruction decode and register fetch.
ALU	ALU or shift operation
MEM	Wait for data from memory on a load and output data for a store.
WB	Write the result into the destination register.

Figure 1: MIPS-X Pipestages

The machine uses a load-store architecture; the only memory operations are explicit loads and stores. The use of the ALU cycle depends on the instruction being executed. For compute instructions, this cycle performs the desired computation, for memory instructions it is used to compute the address of the desired memory location and for branch instructions, it is used to compute the condition. All memory operations use the same addressing mode; the contents of a register are added to a 17-bit signed offset to produce a 32-bit address. There are 32 general purpose registers in the datapath with a 32-bit ALU and a funnel shifter for compute operations.

Although a compute instruction finishes its computation during the third pipeline cycle (ALU), the result is not written back into the register file until the last pipeline cycle. This delayed writeback is done to make instructions only change machine state during their last pipeline cycle, making exception handling much easier. Bypassing is used to reduce the number of pipeline interlocks.

All instructions are restartable so MIPS-X will support a dynamic, paged virtual memory system. To help implement such a system, MIPS-X supports both maskable and nonmaskable interrupts. For systems requiring more complex interrupt handling, an external interrupt coprocessor can be added. MIPS-X also provides two operating modes, system and user, that execute in separate address spaces to provide the protection needed to implement an operating system. The current mode is stored in the PSW and it can only be changed while executing in system mode.

A Hardware Overview

The major components of MIPS-X are the instruction cache data array, the instruction register and the datapath. The datapath is composed of the register file, the execution unit, PC unit and the tag store for the instruction cache. The organization of these parts is shown in Figure 2.

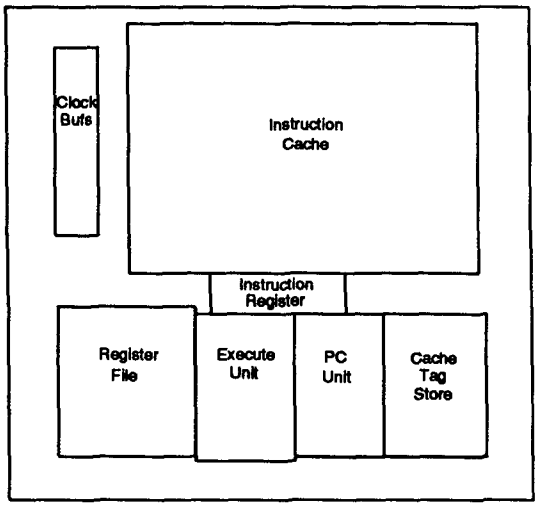

Figure 2: MIPS-X Floorplan

The instruction cache is organized as an 8-way set-associative cache, with 4 sets (rows) and 16 words in each block (line). A sub-block replacement scheme is used so there are 512 valid bits, one per word, as well as the 32 tags. These are located in the datapath to decrease the time needed to detect an instruction cache miss.

The instruction register latches the output from the instruction cache and predecodes some fields of each instruction. It also controls the flow of data during cache misses so that instructions can be written into the cache. During a cache miss, the instruction is latched in the instruction register from the data bus while it is going to the cache memory array. This latch provides a very useful testing feature by allowing the processor to run with the cache disabled.

The register file contains 31 general purpose registers and a hardwired constant zero register. It is useful to have a read-only register as a place to write unwanted data. The constant zero was chosen because it is used as a source value for many instructions such as loading immediate values by doing an add immediate to Register 0. Registers to handle two levels of bypassing and the memory data registers are also in this section.

Shifting and ALU operations are done in the execute unit. It contains a 64-bit to 32-bit funnel shifter and a 32-bit ALU. There is also a special register, called the MD register, that is used during multiplication and division instructions.

The program counter, or PC unit, contains a displacement adder for branches, an incrementer and a chain of shift registers to save the PC values of the instructions currently in execution. Having both the displacement adder and the incrementer means that as soon as the branch condition is determined the PC bus can be driven with the correct value. The PC values in the shift chain are needed to restart the machine after an exception.

In a small area above each section of the datapath is local instruction decoding and control for that section. The overall control of the machine is handled by two finite state machines located in the PC unit. One of them is used to handle Icache misses and the other one does instruction *squashing* during exceptions and branches. Squashing an instruction converts it into a *no-op* instruction.

Critical Paths

To run the processor at or above 20 MHz meant that much attention had to be paid to possible critical paths. In each cycle, we tried to minimize the number of series operations as much as possible. Whenever feasible, a signal was given a full phase to be decoded and driven from one section to another.

There were a few paths that we felt were most likely to be critical paths and we spent a lot of time concentrating on them. The most important of these involved external data fetches. In the specification for the pipeline, addresses would be computed during $\phi 1$ of the ALU cycle and driven to the address pads during $\phi 2$. The Ecache would be accessed during the MEM cycle. Even assuming that the address could be driven off the chip by the end of ALU, completing a fetch in 50 ns would be tight because of the address buffer delay, memory access time and setup time for the fetched data. Getting the result of the tag compare back in a cycle seemed impossible since this would also involve delay through some comparators. To ease the constraint on getting the tag compare back, we decided to use a *late-miss* signal. This meant that the cache would inform the processor at the beginning of the WB cycle whether the cache access during MEM was successful. If there was a miss, then the processor would effectively go back and re-execute $\phi 2$ of MEM to try the access again. This loop would continue until the cache got the data and signaled a hit. Throughout the design we had to be careful not to unnecessarily add delay to the memory-fetch path.

Other paths that we tried to optimize included the path from branch condition generation to driving the PC Bus, instruction cache hit detection, squeezing the ALU time into 1 phase to get the address out by the end of the cycle and doing register reads and writes in one cycle. The latter two were strictly circuit design issues and are not discussed any further here.

The Instruction Cache

Advances in processor architecture and VLSI technology have increased faster than the improvements in packaging technology. This has meant that high-performance VLSI processors have become memory bandwidth limited. For example, if we assume that one instruction is fetched every cycle while, on average, data is only fetched every third cycle, then MIPS-X will have an average bandwidth of 26 MWords/s and a peak bandwidth of 40 MWords/s. Clearly, on-chip memory would help to alleviate this bottleneck. For MIPS-X, we built an on-chip 512-word instruction cache and the tradeoffs made in its design are described in detail elsewhere[6]. We will only discuss the salient features here.

The instruction cache was the first part of the chip to be designed. We first fixed a die size that we felt had enough area to implement the functionality we desired yet small enough that we could expect a reasonable yield of working parts. The datapath and control would take about half of the area inside the padframe so the cache was allocated the remaining area fixing its area and aspect ratio. The other main constraint on the cache was that the cycle time had to be less than the 50ns clock cycle. Given these constraints we investigated many different floorplans and organizations, trying to minimize the average cost of an instruction fetch. This cost is a function of the cache hit rate, the miss penalty, and the cache access time.

We found that the performance of the cache was more sensitive to the the miss service time than the miss ratio. This meant that the implementation details of the cache were more important than the cache organization because the implementation affected how quickly we could determine whether an address hit in the cache. With our pipelining, this meant the difference between stalling the machine for 2 or 3 cycles on a cache miss. By placing the tag and valid-bit stores in the datapath close to the PC unit a 2-cycle miss could be realized. This lengthened the datapath by the number of cache tags and meant that we could not have smaller block sizes because more tags would make the datapath too long. However, the benefits of having fewer cache miss cycles far outweighed the slightly lower miss rates achievable by having smaller blocks.

Initial simulations of this organization yielded disappointing results. Using a set of medium size programs we achieved miss rates that averaged over 20%. We felt that real programs would have worse miss rates, pushing the cost of an instruction fetch close to 1.5 cycles. We found a way to reduce the number of cache miss cycles to 1 by writing the missed instruction into the Icache as soon as it got back onto the chip, but since accessing external data was already one of the critical paths we did not want to risk extending the cycle time to complete the write. Instead we realized that the 2 cache miss cycles could be used to fetch back 2 instructions, the one that missed and the next one to be executed. Doing this double fetch did not affect the critical path and, in fact, was easier to do than fetching back only one instruction because it minimized the disruption of the pipeline. Fetching back 2 words almost halves the miss ratio, driving down the cost of an instruction fetch to that of a single-cycle miss. The key realization here was that there was extra cache bandwidth available and that we could use it to fetch back the next instruction, significantly improving the cache miss ratio without impacting the cycle time of the machine. Fetching back more words would not be advantageous because the bandwidth of the cache is fully used.

Trace driven simulations show that with our set of large Pascal and Lisp benchmarks, the cache has an average miss rate of 12% resulting in an average instruction executing in 1.24 cycles.

The Coprocessor Interface

The coprocessor interface was considered from the very beginning of the design. It also led to some of the most interesting discussions within the MIPS-X design team. We spent considerable time trying to find an efficient interface that would give reasonable performance and still fit within the constraints of VLSI packaging and design. This problem was exacerbated by the presence of the on-chip instruction cache, since now all instructions would not be visible to the outside world.

The proposal for the first instruction set had a single bit in every instruction to specify whether the instruction was for the CPU or a coprocessor. For instructions with the coprocessor bit set, MIPS-X would perform all the addressing calculations, but would not affect any of its stored data. That is, all coprocessor memory instructions still used the processor to generate the addresses and the required control signals, while the coprocessor either acted as a source or sink of the data. To make the coprocessor instructions visible outside of the processor, a dedicated bus was required to transfer the instruction off the processor chip. This scheme had 2 disadvantages: all interprocessor communication had to go through memory, and a coprocessor bus was required. A minor concern was that half the opcode space was devoted to the coprocessor; there had to be a more efficient encoding.

The next instruction format divided the opcode space into three instruction types: memory operations, branches and compute operations. The memory and compute instructions had a 3-bit field to specify the coprocessor number, branches were only done on the main processor. If Coprocessor 0 was specified then the instruction was for the main processor, otherwise the instruction was for one of the 7 available coprocessors. To branch on a coprocessor condition, the coprocessor would first be told to assert a single input to the main processor and a *branch on coprocessor true* or *branch on coprocessor false* would be executed to test the status of that input. Several coprocessors could be connected by wire-oring their outputs. This scheme still had the problem that data transfers between processors must be done through memory.

It was then proposed that all coprocessor instructions must be non-cached, removing the need for a coprocessor bus. The issue of pins and pin bandwidth was heavily debated within the MIPS-X design team. Pins on the processor were in short supply and devoting approximately 20 of them to the coprocessor interface seemed excessive. The question was not just whether there were enough pins available. Without the coprocessor bus, MIPS-X would need only about 90 signal pins, a relatively small number by today's standards. Rather the argument focused on what would be the best use of these pins if we had them. It was not at all clear that using them for the coprocessor interface was the most effective use of the pins. To prevent coprocessor instructions from being cached, a bit in the instruction cache would be set when an instruction being loaded was detected to be a coprocessor instruction. If the bit was set during an instruction fetch that missed, the coprocessor would get the instruction off the memory bus as the main processor read the instruction from memory during the cache miss cycle.

The obvious disadvantage of this approach was that all coprocessor operations incurred an overhead from the internal cache miss. Our initial benchmarks indicated that this would not cause a significant performance loss, but when we generated traces from some floating point intensive code we realized a significant percentage of the instructions were floating point instructions. This caused a re-examination of the decision to not cache coprocessor instructions, and led to the coprocessor scheme that was finally chosen.

The opcode encoding of the machine was changed again, this time making coprocessor operations a form of memory operation or more accurately, memory instructions became a type of coprocessor instruction. Coprocessor instructions work in this scheme by using the address lines to transmit the coprocessor instruction. A memory instruction takes a 17-bit offset constant and adds it to the contents of a register to compute the memory address. If the memory system ignores the cycle, it is possible to pass the 17-bit offset constant to a coprocessor as an instruction. The instruction would include a 3-bit field to specify the coprocessor being addressed, although the processor does not need to know the format of these instructions. This scheme has several advantages over our earlier ideas. A coprocessor instruction bus is not required, since the instructions are sent out over the address pins. Only one extra pin is required to tell the memory system to ignore the cycle. Additional pins can now be used for alleviating the pin bandwidth problem in other parts of the system. Using coprocessor load and store instructions, data can be directly transferred between processors by making the coprocessor supply or read data on the data bus instead of the memory. Also, the coprocessor instructions can be cached just like all the other instructions. The disadvantages of this scheme are that there are fewer bits to specify the coprocessor instructions, and all data to and from the coprocessor's registers must be transferred through the main processor registers first before it can be sent to memory.

Having to transfer all data through the main processor registers was still thought to be inefficient for heavy floating point computation. This lead to a further modification of the instruction set to add *load floating* and *store floating* instructions. These instructions provide one special coprocessor with its own load and store instructions, which we assume will be a floating point unit (FPU). The interface now allows one special coprocessor to load and store its registers directly to memory, without passing through the main processor, in a single instruction. All other coprocessors require one extra cycle for memory loads/stores.

One final tweaking of the interface was to remove the coprocessor branch instructions. The main reason for their removal was the problem of saving state in the coprocessors across exceptions. The solution was to just read a coprocessor status register into a main processor register and then branch according to the value of that register. This change eliminated the last set of problems we had discovered with the coprocessor instructions.

By using the address lines, the resulting coprocessor interface has instructions that can be cached, does not require a large coprocessor bus, allows efficient communication between the processor registers and the coprocessor registers, and lets a single coprocessor have direct access to memory.

Branches

Having set out the initial architecture of the machine, we quickly ran into the problem of branches, and branch delays. Branches have a considerable effect on the performance of a computer especially one that is pipelined as deeply as MIPS-X. The effects of branches in a pipelined machine are particularly noticeable because branches interrupt the flow of the pipeline. Decisions about the design of the pipeline and the type of branch scheme used are not independent. Control complexity is a serious issue.

We very quickly decided to eliminate the use of condition codes in MIPS-X if possible. This decision was motivated by two facts. First, instruction trace statistics indicated that a prior compute operation infrequently generated the condition code needed for a branch. In roughly 80% of the branches an explicit compare operation must be performed to set the condition codes. A previous analysis[7] of empirical data showed that the number of instructions saved by condition codes was very small and essentially useless. Second, condition codes generate state that needs to be saved and restored during exceptions. Handling condition codes in a pipelined machine is difficult because when an exception occurs, great care must be taken to ensure that the correct condition codes are saved. It seemed to us that condition codes provide little benefit and have potential complexity problems. In particular, generating code to use condition codes efficiently is not as straightforward as one might expect. All the branch schemes considered for MIPS-X contained an explicit compare in the branch. This actually reduces the amount of control logic required because there is no need to worry about how to save this state.

Two arithmetic operations are required to execute a branch instruction. One is to compute the branch condition and the other is to compute the branch destination. A machine that uses condition codes computes the branch condition before the actual branch instruction and saves the condition in a condition code register. The first idea conceived for implementing branches in MIPS-X computed the condition in the branch instruction, but did not compute the branch destination. Instead the branch destination was made explicitly visible in the architecture. The user would have to load a register called *PC+1* with the branch destination. The branch instruction computes a condition and then selects *PC+1* or the next sequential instruction depending on the computed condition. An observation was made that many inner loops contain several forward branches due to constructs like if-then-else statements so it would be good to have several *PC+1* registers. Four was felt to be sufficient. This would allow the compiler to hoist the destination address calculations out of the loop. Without this feature, the contents of *PC+1* would have to be loaded from a register for each branch within the loop for each iteration of the loop.

This scheme still had the problem that there was some state that must be saved (the *PC+1* registers) when an exception occurred. Also, deciding how to use the *PC+1* registers could be cumbersome for the compiler system. Finally, with four special registers, it was no longer clear that this solution was easier to implement than simply including a separate adder to compute the destination while the ALU performed the comparison. At this point in the design, adding a little hardware to the datapath to make the control simpler was the wisest choice so we added the separate adder to compute the destination.

During this period we also became concerned about the effect of the branch delay slots on the machine's performance. Often in a pipelined machine one or more instructions following a branch are fetched before the result of the condition evaluation is known. If these instructions are executed, then the machine is said to have a *delayed branch* meaning the effect of the branch occurs after the actual branch instruction. The number of cycles or *delay slots* that execute after the branch instruction and before the actual branch occurs is called the *branch delay*. Filling these delay slots is not a simple task[8, 9, 10] and affects the overall performance.

In the MIPS-X pipeline, it is most straightforward to implement a branch with a delay of two. The ALU is used to compute the branch condition during the third (ALU) pipestage. Filling two delay slots did not seem very promising. Using data from MIPS instruction traces, we expected over 50% of the slots to remain empty[8]. This performance problem lead to discussions about how to reduce the branch delay to 1 cycle, and whether we could use branch prediction to help reduce the wasted cycles[11, 12].

A *quick compare*[3] was proposed as a method to reduce the branch delay. In this scheme, simple comparisons between the two source registers are done before the ALU cycle. This comparison would be performed at the end of the RF cycle by placing a comparator on the output of the register file. Only equality and sign comparisons can be obtained using this method since there is not enough time for an arithmetic operation. Other conditions such as *greater than* would require two steps. The ALU operation is done first and the result is stored in a register. This result is then used in a quick sign compare instruction.

The main question that needed to be resolved initially was what percentage of branches could be handled by a quick compare. Statistics from Katevenis's thesis indicate that by changing the compiler slightly, about 80% of all branches can be converted into quick compares[3], but this means that 20% of all branches take two cycles. Our initial statistics indicated that the number of branches that could be handled using a quick compare was between 70% and 80%.

The quick compare was eventually dropped because it could potentially lengthen the processor cycle time. The comparator circuit must operate on the source buses leading to the ALU and since the values on the buses could come from a bypass source it was possible that the buses would not be stable until late into that cycle, particularly for a previous memory fetch because the data would only be back at the very end of the cycle. For the quick compare to operate, we would need to perform a compare on these values and then use this result to select the correct address of the next instruction. The potential increase in cycle time discounted its slight advantage in the average number of cycles it takes to complete a branch. In retrospect, our decision was correct. In the final machine, the delay from the generation of the branch signal to driving the correct value on the PC Bus is long (measured to be about 20 ns). Even providing a full phase to drive this path leaves it on a critical path.

Left with a branch delay of 2, we investigated branch prediction as a way to reduce the effective branch delay. There were two prediction algorithms tried: branch cache, and static prediction. The branch cache was quickly discarded

when we discovered that it had to be fairly large (much greater than 16 entries) to get a high hit rate. It would also affect the size of our instruction cache. Besides, it never did much better than static prediction and was much more complex. Static prediction would use information at compile time (possibly with profiling) to predict which way a branch would go.

To make use of the prediction information we considered implementing *squashing*, the ability to convert an instruction into a *no-op* if the branch did not go in the predicted direction. In MIPS, the instructions in the branch delay slots are always executed. The strategy for choosing instructions is to first try to move an instruction from before the branch into the slot. If no instructions can be moved past the branch the next choice is to find instructions from the destination or the sequential path that have no effect if the branch goes the wrong way. Thus if you predict correctly, the slot performs a useful instruction and if the branch goes the other way, the slot instruction is simply wasted. The last alternative is to place a *no-op* instruction in the slot. Squashing relaxes the restriction on the second choice for instructions. It allows any instruction from the branch destination to be placed in the slot, even when there is an adverse effect if the branch goes the wrong way. The machine squashes the instruction (turns it into a *no-op*) if the branch goes the wrong way.

With squashing there are three options for dealing with the instructions in the delay slots giving three possible branch types: *no squash* where the slot instructions are always executed, *squash if don't go* where the slot instructions are executed if the branch takes and *squash if go* where the slot instructions are executed if the branch does not take. Since we decided to use static prediction, and in the static case most branches go, MIPS-X only has the first two types of branches. This requires only one bit in the instruction to specify how to deal with the instructions in the slots.

Various combinations of one and two-slot schemes with and without squashing were evaluated. The results are shown in Table 1. The *no squash* scheme is the same as used in MIPS where the instructions in the slots are always executed. The *always squash* scheme only uses the *squash if go* and *squash if don't go* actions for the instructions in the branch slots. The *squash optional* scheme includes the use of branches with *no squash* instructions in the slots as well as having branches with squashing. It can be seen that by allowing squashing the efficiency of branches is much better.

Branch Scheme	Cycles/Branch[2]
2-slot no squash	2.0
2-slot always squash	1.5
2-slot squash optional	1.3
1-slot no squash	1.4
1-slot always squash	1.3
1-slot squash optional	1.1

Table 1: Average Cycles per Branch Instruction for Various Branch Schemes

[2]If all of the branch delay slots could be filled with useful instructions, then we would achieve the ideal of a 1 cycle branch. Any no-op instructions in the branch delay slots are attributed to the cost of the branch so a branch with 2 no-ops in its two delay slots is deemed to have a cost of 3.

The scheme we finally chose uses the full compare and *squash optional* with two slots. Our initial estimates about the cost of the double slots turned out to be slightly optimistic. Where we predicted the average branch would take 1.3 cycles, results using the actual reorganizer showed that the average branch took about 1.5 cycles for small benchmarks using traditional optimization. However, we have since developed better optimization techniques and our most recent results show that even with large Pascal and Lisp benchmarks the average branch takes 1.27 cycles.

Implementing squashing was a gamble because we were not completely sure how it would affect exception handling at the time we made the commitment to use it. It turned out that they mesh together very well as described in the next section.

Exception Handling

As the design of the machine progressed, our concentration shifted from the functions the machine was going to perform to how these functions were going to be controlled. MIPS-X benefited greatly from the experience gained during the MIPS design. Handling exceptions in MIPS caused the most complexity in the machine because of the large number of possible states in the processor during an exception. These states were the result of the processor trying to complete the instructions that occurred conceptually before the fault but still in the pipeline, and reloading the partially full pipeline on a return from an exception. The goal for MIPS-X was to require as few states as possible to handle an exception so the state machine design would not be difficult. The underlining rule was to *keep it simple, stupid*[13].

In some ways exception handling in MIPS-X followed the MIPS model. Exceptions are not vectored so the exception handler must first determine the cause of the exception. On MIPS there was an on-chip *surprise register* where this information was stored. MIPS-X relies instead on a separate off-chip interrupt control unit that contains this information. The PSW does contain bits that determine whether the exception was caused by an interrupt, arithmetic overflow or a non-maskable interrupt.

MIPS-X differed from MIPS in how exceptions affected the pipeline. The MIPS exception sequence started with the pipeline being flushed of as many instructions as possible that were already executing. Then the program counter (PC) was zeroed and the return PCs saved from the PC chain. The flushing of the pipeline caused a great many extra states and added a lot of complexity.

In MIPS-X the pipeline is halted when an exception occurs. No instructions are completed. The PC is immediately set to zero and the shift chain of old PC values is frozen, saving the addresses of the instructions that are still in the pipeline. The current PSW is placed in PSWold, interrupts are turned off and the machine is placed into system mode. The exception routine, located at address zero in system space, begins execution by first saving the three PCs from the PC chain and PSWold onto the system stack. Once the state of the interrupted process is saved, then PC shifting can be enabled and interrupts unmasked if desired. The restart sequence involves reloading the PC chain with the three saved PCs and then doing three special jumps using the contents of the PC chain; the PC chain is used to store the

return addresses during the return sequence. Interrupts must be disabled both during machine state saving and restoring.

During the discussions about how branches were to be implemented, there was some concern about the effects the branch implementation would have on exception handling. The original feeling was that having more branch slots would require more state in the machine and implementing squashing branches would make the state machine even more complicated. The squash proponents argued that the hardware needed to freeze the pipeline during an exception could be used to implement squashing branches. They not only convinced the design team, they also turned out to be correct. Squashing two branch slots only requires a single extra input to the squashing finite state machine that is used to handle exceptions. Branch squashing and squashing for exceptions are very similar.

The general scheme used to no-op an instruction is quite simple. All that needs to be done is to set a bit in the destination specifier for that instruction. This bit is used by the register file to determine whether to perform a write or not. There are 2 lines in the machine that can set this bit, Exception and Squash. Exception no-ops the instructions in the ALU and MEM stages of the pipeline, while Squash no-ops the instructions currently in the IF and RF stages of the pipeline. The only added complexity occurs with the Mult/Div register and the PSW which contains the only visible state outside of the register file. Writes to these locations are also prevented by Exception and Squash.

There is only one exception generated on chip and it is a trap on overflow in the ALU or the multiplication/division hardware. At the start of the design it was felt that detecting overflows and generating a trap was too complex to do. The original solution was the concept of a *sticky overflow* bit. If an overflow occurred then the sticky overflow bit would be set in the PSW. This bit could then be checked at a later time to determine whether an overflow had occurred. This meant that it would not be possible to precisely detect the occurrence of the overflow but at least it was possible to indicate the presence of an incorrect result. We began looking for other overflow mechanisms when we discovered that the sticky overflow bit interacted badly with bypassing. Instead of making the hardware simple, it seemed to make the PSW harder to design.

Several other simple schemes were then proposed. One was a *SetOnAddOverflow* instruction that just routed the overflow bit from the ALU into the most significant bit of the ALU result. This instruction could then be used to determine whether the addition causes an overflow by simply testing for the sign of the result. Another suggestion was a *Branch on Overflow* instruction that caused a branch if the result of the branch comparison overflowed. These were minimal hardware solutions that would provide some small support for overflow detection.

At this point the exception hardware had been designed and we observed that generating a true *trap on overflow* was not difficult; in fact it was simpler than the original sticky overflow bit. We decided to abandon the sticky overflow bit for a maskable trap on overflow.

Control

Our overriding goal for the control section was to keep it as simple as possible. In part we accomplished our goal by eliminating hardware features that would complicate the machine without providing significant performance advantages. We also tried to keep a uniform view of the hardware, trying to reuse the same control mechanism for many features. Merging exceptions and squashing, and merging memory instructions and coprocessor operations were examples of this strategy. Finally, we eliminated the global controller for the machine and replaced it with a set of smaller controllers, one for each section of the datapath. We further partitioned the design so that a single designer was responsible for both the datapath and control in his section, giving each designer the incentive to make his control section simpler. Most of the machine control is simple decoders, many generated automatically using PLA generators.

One technique that MIPS-X used to great advantage was a qualified clock, called $\psi 1$, to latch the control state of the machine. This clock is the $\phi 1$ clock qualified with *not external cache miss* and *not internal cache miss*. When either cache misses, the $\psi 1$ clock does not rise, and the control state does not shift down the pipeline control latches. The lack of a $\psi 1$ clock causes the machine to execute the *previous* $\phi 2$ phase before retrying the $\phi 1$ phase. This simple technique made temporary stalling of the entire pipeline very easy, and allowed us to implement the late miss described earlier without greatly increasing the machine complexity. Since the $\psi 1$ clock is only allowed to clock control state latches, its pulse width can be quite narrow (about 10 ns). As long as the miss signal is monotonic, it is possible to detect a cache hit after the data has been latched in the machine without stalling the machine.

Together these control techniques were quite successful. The control was nicely divided among the 4 main datapath sections, with the only two finite state machines (FSMs) residing in the PC unit. These FSMs handle instruction cache misses and instruction squashing during exceptions and squashed branches. The state diagrams for the two machines are shown in Figures 3 and 4. These FSMs are implemented as simple shift registers with a very small amount of random logic and occupy less than 0.2% of the total area of the chip.

Status and Conclusions

The MIPS-X project began in earnest during the summer of 1984. By January 1985, we had settled on an initial version of the instruction set, and had written an instruction level simulator for the machine. We were able to use much of the software system that was created for MIPS for MIPS-X as well. This greatly reduced the software development effort. The compiler/simulator system generated instruction traces that we used to gather cache statistics and fine tune the architecture. By April 1985, the architecture had stabilized and work on the detailed design accelerated. We ran our first instruction through a detailed functional simulator of the entire processor during the summer. The final design was taped out at the end of April 1986 and we received first silicon back in October.

The processor was designed to run at a clock rate of 20

MHz, executing an instruction every cycle, yielding a peak performance of 20 MIPs. Timing analysis showed that the version that was shipped in April would run at about 16 MHz. Initial timing tests have shown that the part is fully functional and it runs at the projected 16 MHz clock rate. We are now fixing the critical paths so that we can achieve our goal of 20 MHz. The die is 8.5 mm by 8 mm and has a total of 108 pins of which 84 are for signals and 24 are for power and ground. There are about 150K transistors, two thirds of which are in the instruction cache. The power dissipation is less than 1 W.

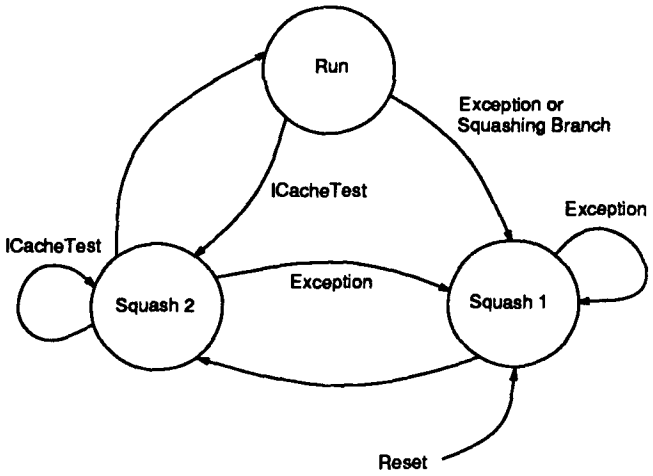

Figure 3: Squash Finite State Machine

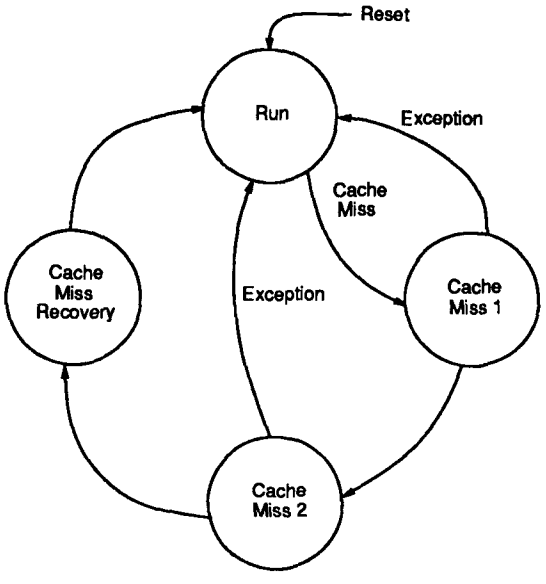

Figure 4: Cache Miss Finite State Machine

Simulations of our large Pascal benchmarks show that 15.6% of all instructions are no-ops due to unused branch delays or other pipeline interlocks that cannot be optimized away. For Lisp, this number increases slightly to 18.3% due to a larger number of jumps and many load-load interlocks caused by chasing car and cdr chains[14]. When the memory system overhead is included (delays from Icache and Ecache misses), the average instruction requires about 1.7 cycles meaning MIPS-X should have a sustained throughput above 11 MIPs. Our benchmark programs have static code sizes in the range of 50 KBytes to 270 KBytes so we cannot get exact numbers for the effects of the external cache because most of the benchmarks fit entirely. Smith's numbers[15] are not large enough so we used much larger traces[16] to derive the Ecache effects.

The performance of a machine is based on three factors: the number of instructions executed (path length), the number of cycles per instruction and the cycle time. Ideally, all three factors should be minimized but we have shown that by having simple instruction decode we can significantly decrease the latter two factors without adversely affecting the path length. Comparison of Pascal programs with a VAX 11/780 shows that MIPS-X executes about 25% more instructions but executes the programs about 14 times faster for unoptimized code. The static code size for MIPS-X is also about 25% greater than VAX code. The Stanford compiler system was used and the only difference was in the back end code generators. However, when MIPS-X code is compared to the Berkeley Pascal compiler, the path length is 80% longer and the speedup is only 10 times faster than the VAX. Much of this difference may be due to poorer code from our VAX code generator. We feel that when we get the results for optimized code, the numbers will be somewhere inbetween.

The goal of the MIPS-X project from the beginning was to learn from MIPS and design a simpler yet faster processor. The emphasis in all design decisions throughout the project was simplicity: minimize state and keep the control simple. The implementation of MIPS-X has shown that it is possible to implement a high performance microprocessor that supports coprocessors, without requiring complex control or hundreds of pins.

Acknowledgements

The MIPS-X research project has been supported by the Defense Advanced Research Projects Agency under contract #MDA903-83-C-0335. Paul Chow was partially supported by a postdoctoral fellowship from the Natural Sciences and Engineering Research Council of Canada.

Many people have contributed to the MIPS-X research effort. Malcolm Wing, Arturo Salz, Karen Huyser, Anant Agarwal, Scott McFarling, C.Y. Chu, Steve Richardson, Steve Tjiang, John Acken, Richard Simoni, Glenn Gulak, Kathy Cuderman, Takeshi Tokuda, Eugen Reithmann, Steven Przybylski, Chris Rowen, Norm Jouppi, Thomas Gross, John Gill and John Hennessy deserve special thanks for their contributions to the project.

References

1. G. Radin, "The 801 Minicomputer", *Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto*, March 1982, pp. 39-47.

2. D. Patterson and C. Sequin, "A VLSI RISC", *Computer*, September, 1982, pp. 8-21.

3. M. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI", Computer Science Division (EECS) UCB/CSD 83/141, Univ. of CA at Berkeley, October 1983.

4. J. Hennessy, et al., "The MIPS Machine", *COMPCON*, IEEE, Spring 1982, pp. 2-7.

5. S. Przybylski, T. Gross, J. Hennessy, N. Jouppi, C. Rowen, "Organization and VLSI Implementation of MIPS", *Journal of VLSI and Computer Systems*, Vol. 1, No. 2, December, 1984, pp. 170-208.

6. Anant Agarwal, Paul Chow, Mark Horowitz, John Acken Arturo Salz and John Hennessy, "On-chip Instruction Caches for High Performance Processors", *Proceedings, Stanford Conference on Advanced Research in VLSI*, March 1987, pp. 1-24.

7. J.L. Hennessy, N. Jouppi, F. Baskett, T.R. Gross and J. Gill, "Hardware/Software Tradeoffs for Increased Performance", *Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto*, March 1982, pp. 2-11.

8. Thomas Gross, *Code Optimization of Pipeline Constraints*, PhD dissertation, Stanford University, December 1983, Available as Stanford University CSL Technical Report 83-255.

9. John Hennessy and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July, 1983, pp. 422-448.

10. Scott McFarling and John Hennessy, "Reducing the Cost of Branches", *Proceedings, 13th Symposium on Computer Architecture*, June 1986, pp. 396-403.

11. J.E. Smith, "A Study of Branch Predition Strategies", *Proceedings, Eighth Symposium on Computer Architecture*, May 1981, pp. 135-148.

12. Johnny K. F. Lee, Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *Computer*, January, 1984, pp. 6-22.

13. Butler W. Lampson, "Hints for Computer System Design", *IEEE Software*, Vol. 1, No. 1, January, 1984, pp. 11-30.

14. Peter Steenkiste, *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*, PhD dissertation, Stanford University, 1987, To appear in 1987.

15. Alan Jay Smith, "Cache Memories", *Computing Surveys*, Vol. 14, No. 3, September, 1982, pp. 473-530.

16. Anant Agarwal, Richard L. Sites and Mark Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode", *13th Annual International Symposium on Computer Architecture*, IEEE, June 1986, pp. 119-127.

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.0. **[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-

CONTENTS

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 microprocessor. We believe that within 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),
(3) minimizing the delay due to a miss, and

(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 component of the S-unit is the cache memory.

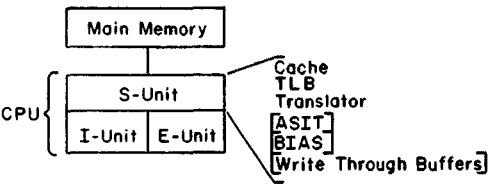

Figure 1. A typical CPU design and the S-unit.

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 into the segment and page tables for the

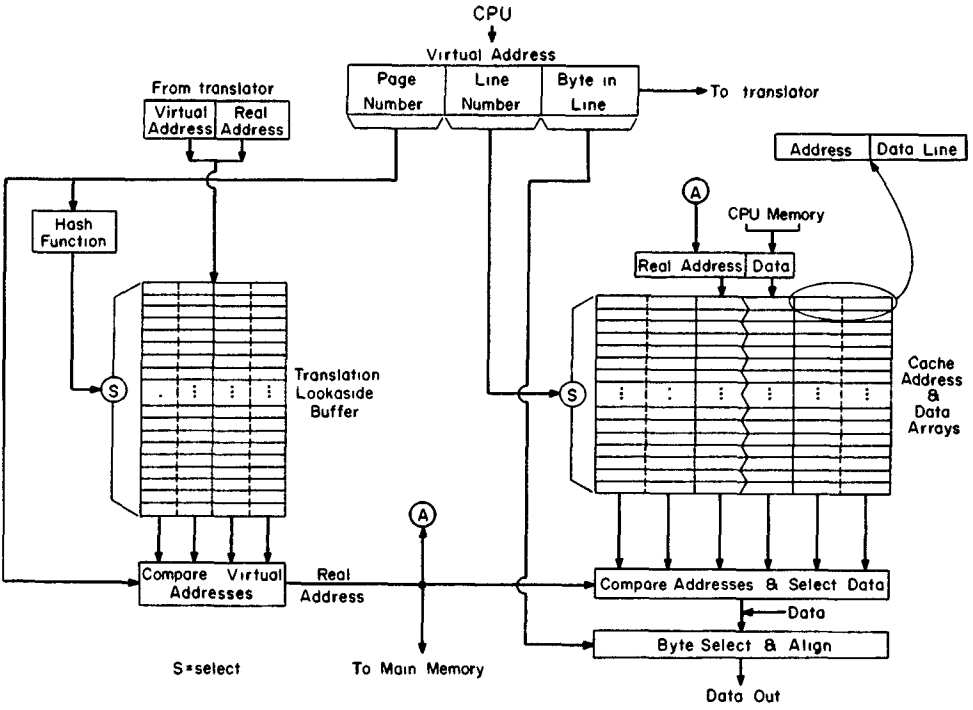

Figure 2. A typical cache and TLB design.

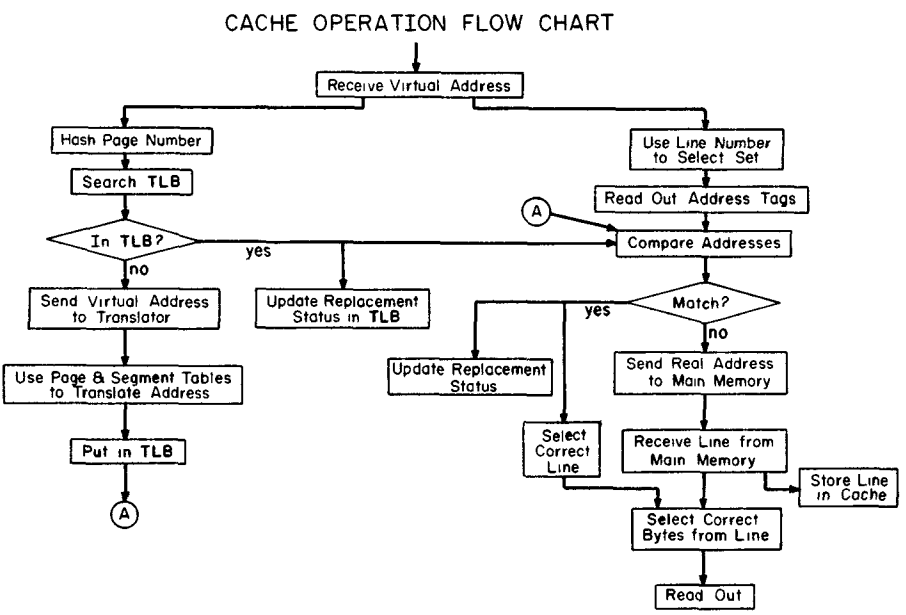

Figure 3. Cache operation flow chart.

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).

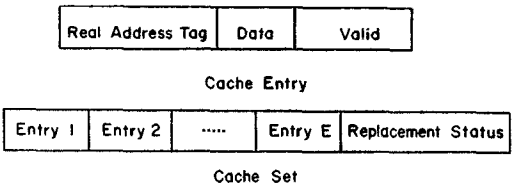

Figure 4. Structure of cache entry and cache set.

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

replacement. Various replacement algorithms are possible, such as FIFO (first in, first out), LRU (least recently used), and random. Later, we consider the first two of these.

Main Memory Update Algorithm. When the CPU performs a write (store) to memory, that operation can actually be reflected in the cache and main memories in a number of ways. For example, the cache memory can receive the write and the main memory can be updated when that line is replaced in the cache. This strategy is known as *copy-back*. Copy-back may also require that the line be fetched if it is absent from the cache (i.e., fetch-on-write). Another strategy, known as write-through, immediately updates main memory when a write occurs. Write-through may specify that if the information is in the cache, the cache be either updated or purged from main memory. If the information is not in the cache, it may or may not be fetched. The choice between copy-back and write-through strategies is also influenced by the need to maintain consistency among the cache memories in a tightly coupled multiprocessor system. This requirement is discussed later.

Cold-Start versus Warm-Start Miss Ratios and Multiprogramming. Most computer systems with cache memories are multiprogrammed; many processes run on the CPU, though only one can run at a time, and they alternate every few milliseconds. This means that a significant fraction of the cache miss ratio is due to loading data and instructions for a new process, rather than to a single process which has been running for some time. Miss ratios that are measured when starting with an empty cache are called cold-start miss ratios, and those that are measured from the time the cache becomes full are called warm-start miss ratios. Our simulation studies consider this multiprogramming environment.

User/Supervisor Cache. The frequent switching between user and supervisor state in most systems results in high miss ratios because the cache is often reloaded (i.e., cold-start). One way to address this is to incorporate two cache memories, and allow the supervisor to use one cache and the user programs to use the other. Potentially, this could result in both the supervisor and the user programs more frequently finding upon initiation what they need in the cache.

Multicache Consistency. A multiprocessor system with multiple caches faces the problem of making sure that all copies of a given piece of information (which potentially could exist in every cache, as well as in the main memory) are the same. A modification of any one of these copies should somehow be reflected in all others. A number of solutions to this problem are possible. The three most popular solutions are essentially: (1) to transmit all stores to all caches and memories, so that all copies are updated; (2) to transmit the addresses of all stores to all other caches, and purge the corresponding lines from all other caches; or (3) to permit data that are writeable (page or line flagged to permit modification) to be in only one cache at a time. A centralized or distributed directory may be used to control making and updating of copies.

Input/Output. Input/output (from and to I/O devices) is an additional source of references to information in memory. It is important that an output request stream reference the most current values for the information transferred. Similarly, it is also important that input data be immediately reflected in any and all copies of those lines in memory. Several solutions to this problem are possible. One is to direct the I/O stream through the cache itself (in a single processor system); another is to use a write-through policy and broadcast all writes so as to update or invalidate the target line wherever found. In the latter case, the channel accesses main memory rather than the cache.

Data/Instruction Cache. Another cache design strategy is to split the cache into two parts: one for data and one for instructions. This has the advantages that the bandwidth of the cache is increased and the access time (for reasons discussed later) can be decreased. Several problems occur: the overall miss ratio may increase, the two caches must be kept consistent, and self-modifying code and *execute* instructions must be accommodated.

Virtual versus Real Addressing. In computer systems with virtual memory, the cache may potentially be accessed either with a real address (real address cache) or a virtual address (virtual address cache). If real addresses are to be used, the virtual addresses generated by the processor must first be translated as in the example above (Figure 2); this is generally done by a TLB. The TLB is itself a cache memory which stores recently used address translation information, so that translation can occur quickly. Direct virtual address access is faster (since no translation is needed), but causes some problems. In a virtual address cache, inverse mapping (real to virtual address) is sometimes needed; this can be done by an RTB (reverse translation buffer).

Cache Size. It is obvious that the larger the cache, the higher the probability of finding the needed information in it. Cache sizes cannot be expanded without limit, however, for several reasons: cost (the most important reason in many machines, especially small ones), physical size (the cache must fit on the boards and in the cabinets), and access time. (The larger the cache, the slower it may become. Reasons for this are discussed in Section 2.12.). Later, we address the question of how large is large enough.

Multilevel Cache. As the cache grows in size, there comes a point where it may be usefully split into two levels: a small, high-level cache, which is faster, smaller, and more expensive per byte, and a larger, second-level cache. This two-level cache structure solves some of the problems that afflict caches when they become too large.

Cache Bandwidth. The cache bandwidth is the rate at which data can be read from and written to the cache. The bandwidth must be sufficient to support the proposed rate of instruction execution and I/O. Bandwidth can be improved by increasing the width of the data path, interleaving the cache and decreasing access time.

1. DATA AND MEASUREMENTS

1.1 Rationale

As noted earlier, our in-depth studies of some aspects of cache design and optimization are based on extensive trace-driven simulation. In this section, we explain the importance of this approach, and then discuss the presentation of our results.

One difficulty in providing definitive statements about aspects of cache operation is that the effectiveness of a cache memory depends on the workload of the computer system; further, to our knowledge, there has never been any (public) effort to characterize that workload with respect to its effect on the cache memory. Along the same lines, there is no generally accepted model for program behavior, and still less is there one for its effect on the uppermost level of the memory hierarchy. (But see AROR72 for some measurements, and LEHM78 and LEHM80, in which a model is used.)

For these reasons, we believe that it is possible for many aspects of cache design to make statements about relative performance only when those statements are based on trace-driven simulation or direct measurement. We have therefore tried throughout, when examining certain aspects of cache memories, to present a large number of simulation results and, if possible, to generalize from those measurements. We have also made an effort to locate and reference other measurement and trace-driven simulation results reported in the literature. The reader may wish, for example, to read WIND73, in which that author discusses the set of data used for his simulations.

1.2 Trace-Driven Simulation

Trace-driven simulation is an effective method for evaluating the behavior of a memory hierarchy. A trace is usually gathered by interpretively executing a program and recording every main memory location referenced by the program during its execution. (Each address may be tagged in any way desired, e.g., *instruction fetch, data fetch, data store.*) One or more such traces are then used to drive a simulation model of a cache (or main) memory. By varying parameters of the simulation model, it is possible to simulate directly any cache size, placement, fetch or replacement algorithm, line size, and so forth. Programming techniques allow a range of values for many of these parameters to be measured simulta-

neously, during the same simulation run [GECS74, MATT70, SLUT72]. Trace-driven simulation has been a mainstay of memory hierarchy evaluation for the last 12 to 15 years; see BELA66 for an early example of this technique, or see POHM73. We assume only a single cache in the system, the one that we simulate. Note that our model does not include the additional buffers commonly found in the instruction decode and ALU portions of many CPUs.

In many cases, trace-driven simulation is preferred to actual measurement. Actual measurements require access to a computer and hardware measurement tools. Thus, if the results of the experiments are to be even approximately repeatable, standalone time is required. Also, if one is measuring an actual machine, one is unable to vary most (if any) hardware parameters. Trace-driven simulation has none of these difficulties; parameters can be varied at will and experiments can be repeated and reproduced precisely. The principal advantage of measurement over simulation is that it requires 1 to 0.1 percent as much running time and is thus very valuable in establishing a genuine, workload-based, actual level of performance (for validation). Actual workloads also include supervisor code, interrupts, context switches, and other aspects of workload behavior which are hard to imitate with traces. The results in this paper are mostly of the trace-driven variety.

1.3 Simulation Evaluation

There are two aspects to the performance of a cache memory. The first is access time: How long does it take to get information from or put information into the cache? It is very difficult to make exact statements about the effect of design changes on access time without specifying a circuit technology and a circuit diagram. One can, though, indicate trends, and we do that throughout this paper.

The second aspect of cache performance is the miss ratio: What fraction of all memory references attempt to access something which is not resident in the cache memory? Every such miss requires that the CPU wait until the desired information can be reached. Note that the miss ratio is a function not only of how the cache design affects the number of misses, but also of how the machine design affects the number of cache memory references. (A memory reference represents a cache access. A given instruction requires a varying number of memory references, depending on the specific implementation of the machine.) For example, a different number of memory references would be required if one word at a time were obtained from the cache than if two words were obtained at once. Almost all of our trace-driven studies assume a cache with a one-word data path (370 words = 4 bytes, PDP-11 word = 2 bytes). The WATEX, WATFIV, FFT, and APL traces assume a two-word (eight-byte) data path. We measure the miss ratio and use it as the major figure of merit for most of our studies. We display many of these results as x/y plots of miss ratios versus cache size in order to show the dependence of various cache design parameters on the cache size.

1.4 The Traces

We have obtained 19 program address traces, 3 of them for the PDP-11 and the other 16 for the IBM 360/370 series of computers. Each trace is for a program developed for normal production use. (These traces are listed in the Appendix, with a brief description of each.) They have been used in groups to simulate multiprogramming; five such groups were formed. Two represent a scientific workload (WFV, APL, WTX, FFT, and FGO1, FGO2, FGO3, FGO4), one a business (commercial) workload (CGO1, CGO2, CGO3, PGO2), one a miscellaneous workload, including compilations and a utility program (PGO1, CCOMP, FCOMP, IEBDG), and one a PDP-11 workload (ROFFAS, EDC, TRACE). The miss ratio as a function of cache size is shown in Figure 5 for most of the traces; see SMIT79 for the miss ratios of the remaining traces. The miss ratios for each of the traces in Figure 5 are cold-start values based on simulations of 250,000 memory references for the IBM traces, and 333,333 for the PDP-11 traces.

1.5 Simulation Methods

Almost all of the simulations that were run used 3 or 4 traces and simulated multipro-

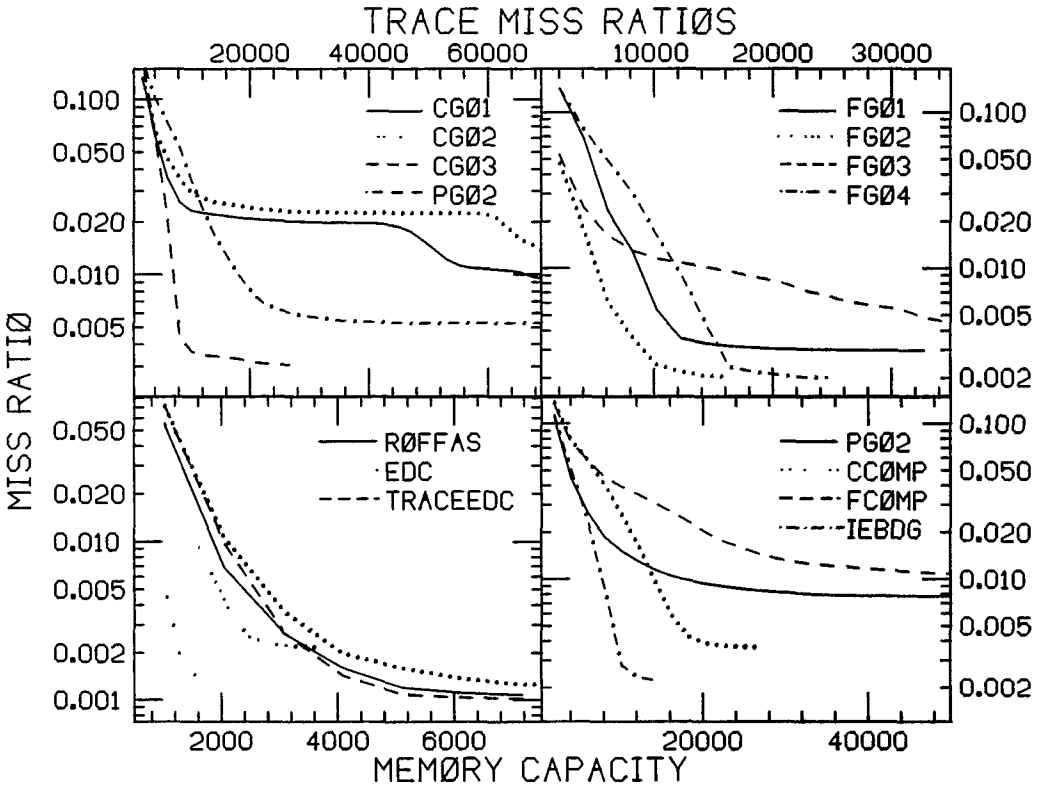

Figure 5. Individual trace miss ratios.

gramming by switching the trace in use every Q time-units (where Q was usually 10,000, a cache memory reference takes 1 time-unit, and a miss requires 10). Multiprogrammed simulations are used for two reasons: they are considered to be more representative of usual computer system operation than uniprogrammed ones, and they also allow many more traces to be included without increasing the number of simulation runs. An acceptable alternative, though, would have been to use uniprogramming and purge the cache every Q memory references. A still better idea would have been to interleave user and supervisor code, but no supervisor traces were available.

All of the multiprogrammed simulations (i.e., Figures 6, 9–33) were run for one million memory references; thus approximately 250,000 memory references were used from each of the IBM 370 traces, and 333,333 from the PDP-11 traces.

The standard number of sets in the simulations was 64. The line size was generally 32 bytes for the IBM traces and 16 bytes for the PDP-11 traces.

2. ASPECTS OF CACHE DESIGN AND OPERATION

2.1 Cache Fetch Algorithm

2.1.1 Introduction

As we noted earlier, one of the two aims of cache design is to minimize the miss ratio. Part of the approach to this goal is to select a cache fetch algorithm that is very likely to fetch the right information, if possible, before it is needed. The standard cache fetch algorithm is demand fetching, by which a line is fetched when and if it is needed. Demand fetches cannot be avoided entirely, but they can be reduced if we can sucessfully predict which lines will be needed and fetch them in advance. A cache fetch algorithm which gets information be-

fore it is needed is called a prefetch algorithm.

Prefetch algorithms have been studied in detail in SMIT78b. Below, we summarize those results and give one important extension. We also refer the reader to several other works [AICH76, BENN76, BERG78, ENGE73, PERK80, and RAU76] for additional discussions of some of these issues.

We mention the importance of a technique known as *fetch bypass* or *load-through*. When a miss occurs, it can be rectified in two ways: either the line desired can be read into the cache, and the fetch then reinitiated (this was done in the original Amdahl 470V/6 [SMIT78b]), or, better, the desired bytes can be passed directly from the main memory to the instruction unit, bypassing the cache. In this latter strategy, the cache is loaded, either simultaneously with the fetch bypass or after the bypass occurs. This method is used in the 470V/7, 470V/8, and the IBM 3033. (A wraparound load is usually used [KROF80] in which the transfer begins with the bytes accessed and wraps around to the rest of the line.)

2.1.2 Prefetching

A prefetch algorithm must be carefully designed if the machine performance is to be improved rather than degraded. In order to show this more clearly, we must first define our terms. Let the *prefetch ratio* be the ratio of the number of lines transferred due to prefetches to the total number of program memory references. And let *transfer ratio* be the sum of the prefetch and miss ratios. There are two types of references to the cache: *actual* and *prefetch lookup*. Actual references are those generated by a source external to the cache, such as the rest of the CPU (I-unit, E-unit) or the channels. A prefetch lookup occurs when the cache interrogates itself to see if a given line is resident or if it must be prefetched. The ratio of the total accesses to the cache (actual plus prefetch lookup) to the number of actual references is called the *access ratio*.

There are costs associated with each of the above ratios. We can define these costs in terms of lost machine cycles per memory reference. Let D be the penalty for a demand miss (a miss that occurs because the target is needed immediately) which arises from machine idle time while the fetch completes. The prefetch cost, P, results from the cache cycles used (and thus otherwise unavailable) to bring in a prefetched line, used to move out (if necessary) a line replaced by a prefetch, and spent in delays while main memory modules are busy doing a prefetch move-in and move-out. The access cost, A, is the penalty due to additional cache prefetch lookup accesses which interfere with the executing program's use of the cache. A prefetch algorithm is effective only if the following equation holds:

$$\begin{aligned} D * \text{miss ratio (demand)} \\ > [D * \text{miss ratio (prefetch)} \\ + P * \text{prefetch ratio} \\ + A * (\text{access ratio} - 1)] \end{aligned} \quad (1)$$

We should note also that the miss ratio when using prefetching may not be lower than the miss ratio for demand fetching. The problem here is cache *memory pollution*; prefetched lines may *pollute* memory by expelling other lines which are more likely to be referenced. This issue is discussed extensively and with some attempt at analysis in SMIT78c; in SMIT78b a number of experimental results are shown. We found earlier [SMIT78b] that the major factor in determining whether prefetching is useful was the line size. Lines of 256 or fewer bytes (such as are commonly used in caches) generally resulted in useful prefetching; larger lines (or pages) made prefetching ineffective.The reason for this is that a prefetch to a large line brings in a great deal of information, much or all of which may not be needed, and removes an equally large amount of information, some of which may still be in use.

A prefetch algorithm has three major concerns: (1) when to initiate a prefetch, (2) which line(s) to prefetch, and (3) what replacement status to give the prefetched block. We believe that in cache memories, because of the need for fast hardware implementation, the only possible line to prefetch is the immediately sequential one; this type of prefetching is also known as *one block lookahead* (OBL). That is, if line

i is referenced, only line $i + 1$ is considered for prefetching. Other possibilities, which sometimes may result in a lower miss ratio, are not feasible for hardware implementation in a cache at cache speeds. Therefore, we consider only OBL.

If some lines in the cache have been referenced and others are resident only because they were prefetched, then the two types of lines may be treated differently with respect to replacement. Further, a prefetch lookup may or may not alter the replacement status of the line examined. In this paper we have made no distinction between the effect of a reference or a prefetch lookup on the replacement status of a line. That is, a line is moved to the top of the LRU stack for its set if it is referenced, prefetched, or is the target of a prefetch lookup; LRU is used for replacement for all prefetch experiments in this paper. (See Section 2.2.2) The replacement status of these three cases was varied in SMIT78c, and in that paper it was found that such distinctions in replacement status had little effect on the miss ratio.

There are several possibilities for when to initiate a prefetch. For example, a prefetch can occur on instruction fetches, data reads and/or data writes, when a miss occurs, always, when the last nth of a line is accessed, when a sequential access pattern has already been observed, and so on. Prefetching when a sequential access pattern has been observed or when the last nth segment ($n = \frac{1}{2}, \frac{1}{4}$, etc.) of a line has been used is likely to be ineffective for reasons of timing: the prefetch will not be complete when the line is needed. In SMIT78b we showed that limiting prefetches only to instruction accesses or only to data accesses is less effective than making all memory accesses eligible to start prefetches. See also BENN82.

It is possible to create prefetch algorithms or mechanisms which employ information not available within the cache memory. For example, a special instruction could be invented to initiate prefetches. No machine, to our knowledge, has such an instruction, nor have any evaluations been performed of this idea, and we are inclined to doubt its utility in most cases. A prefetch instruction that specified the transfer of large amounts of information would run the substantial risk of polluting the cache with information that either would not be used for some time, or would not be used at all. If only a small amount of information were prefetched, the overhead of the prefetch might well exceed the value of the savings. However, some sophisticated versions of this idea might work. One such would be to make a record of the contents of the cache whenever the execution of a process was stopped, and after the process had been restarted, to restore the cache, or better, only its most recently used half. This idea is known as *working set restoration* and has been studied to some extent for paged main memories. The complexity of implementing it for cache makes it unlikely to be worthwhile, although further study is called for.

Another possibility would be to recognize when a base register is loaded by the process and then to cause some number of lines (one, two, or three) following the loaded address to be prefetched [POME80b, HOEV81a, HOEV81b]. Implementing this is easy, but architectural and software changes are required to ensure that the base registers are known or recognized, and modifications to them initiate prefetches. No evaluation of this idea is available, but a decreased miss ratio appears likely to result from its implementation. The effect could be very minor, though, and needs to be evaluated experimentally before any modification of current software or hardware is justified.

We consider three types of prefetching in this paper: (1) always prefetch, (2) prefetch on misses, and (3) tagged prefetch. *Always prefetch* means that on every memory reference, access to line i (for all i) implies a prefetch access for line $i + 1$. Thus the access ratio in this case is always 2.0. *Prefetch on misses* implies that a reference to a block i causes a prefetch to block $i + 1$ if and only if the reference to block i itself was a miss. Here, the access ratio is 1 + miss ratio. *Tagged prefetch* is a little more complicated, and was first proposed by GIND77. We associate with each line a single bit called the *tag*, which is set to one whenever the line is accessed by a program. It is initially zero and is reset to zero when

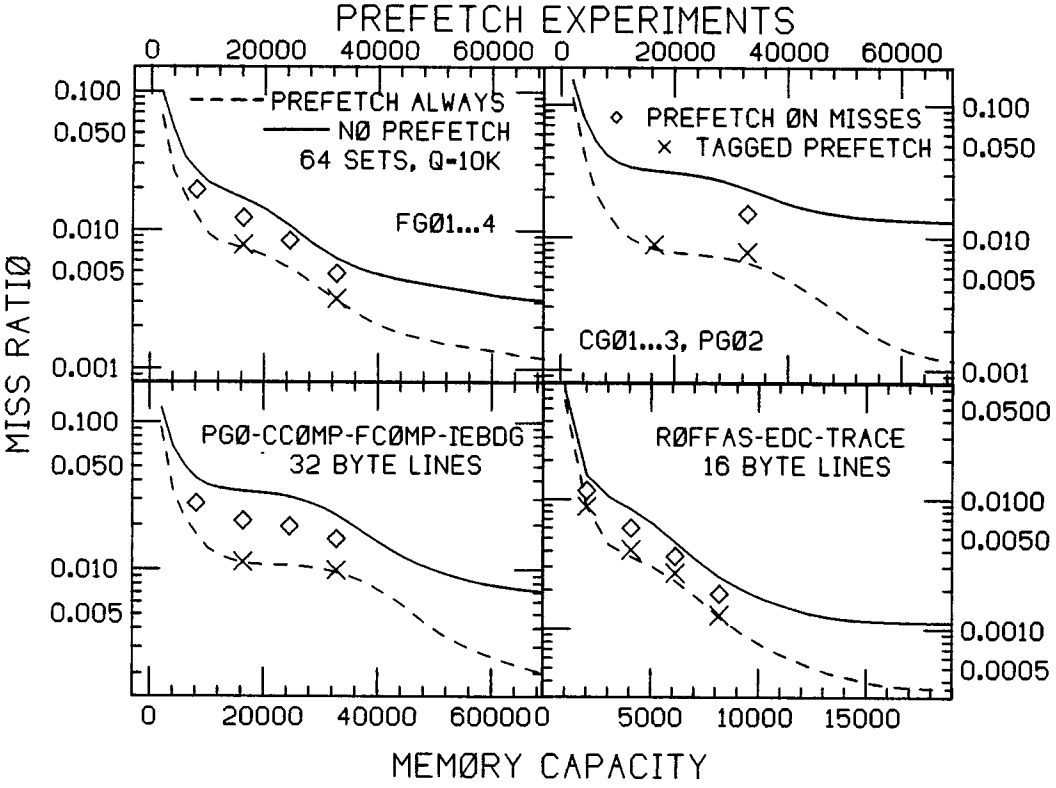

Figure 6. Comparison of miss ratios for two prefetch strategies and no prefetch.

the line is removed from the cache. Any line brought to the cache by a prefetch operation retains its tag of zero. When a tag changes from 0 to 1 (i.e., when the line is referenced for the first time after prefetching or is demand-fetched), a prefetch is initiated for the next sequential line. The idea is very similar to prefetching on misses only, except that a miss which did not occur because the line was prefetched (i.e., had there not been a prefetch, there would have been a miss to this line) also initiates a prefetch.

Two of these prefetch algorithms were tested in SMIT78b: always prefetch and prefetch on misses. It was found that always prefetching reduced the miss ratio by as much as 75 to 80 percent for large cache memory sizes, while increasing the transfer ratio by 20 to 80 percent. Prefetching only on misses was much less effective; it produced only one half, or less, of the decrease in miss ratio produced by always prefetching. The transfer ratio, of course, also increased by a much smaller amount, typically 10 to 20 percent.

The experiments in SMIT78b, while very thorough, used only one set of traces and also did not test the tagged prefetch algorithm. To remedy this, we ran additional experiments; the results are presented in Figure 6. (In this figure, 32-byte lines are used in all cases except for 16-byte lines for the PDP-11 traces, the task switch interval Q is 10K, and there are 64 sets in all cases.) It can be seen that always prefetching cut the (demand) miss ratio by 50 to 90 percent for most cache sizes and tagged prefetch was almost equally effective. Prefetching only on misses was less than half as good as always prefetching or tagged prefetch in reducing the miss ratio. These results are seen to be consistent across all five sets of traces used.

These experiments are confirmed by the results in Table 1. There we have tabulated the miss, transfer, and access ratios for the three prefetch algorithms considered, as

Table 1. Comparison of Three Prefetch Strategies[a]

Program traces	Memory size	Demand miss ratio	Always prefetch			Prefetch on misses			Tagged prefetch		
			Miss ratio	Access ratio	Transfer ratio	Miss ratio	Access ratio	Transfer ratio	Miss ratio	Access ratio	Transfer ratio
WFV, APL	16K	0.02162	0.00883	2.0	0.0297	—	—	—	0.00922	1.0233	0.01491
WTX, FFT1	32K	0.00910	0.00407	2.0	0.0152	0.00656	1.00656	0 01178	0.00405	1.0107	0.01275
ROFFAS, EDC	2K	0.0155	0.00867	2.0	0.0263	—	—	—	0.00873	1.0176	0.02245
TRACE	4K	0.00845	0.00356	2.0	0.0126	0.00593	1 0059	0.0107	0 00404	1 0098	0.01223
	6K	0.00470	0 00232	2.0	0.0085	—	—	—	0.00268	1.0059	0.00722
	8K	0.00255	0.00123	2 0	0.0047	0.00184	1.00184	0.00320	0.00127	1.0030	0.00362
CGO1, CGO2	16K	0.0341	0.00851	2.0	0.0391	0.01997	1.0197	0.03798	0.00893	1.0331	0.03892
CGO3, PGO2	32K	0.0236	0.00670	2.0	0.0331	0.0153	1.0153	0.0288	0.00780	1.0274	0.03168
FGO1, FGO2	16K	0.01702	0.00736	2.0	0 0234	0.01234	1.01234	0.02239	0.00785	1.0194	0.0240
FGO3, FGO4	32K	0.00628	0.00314	2.0	0 0108	0.00489	1 00489	0.00868	0.00320	1.0077	0.00940
PGO1, CCOMP	16K	0.0343	0.0112	2.0	0.0413	0.02087	1.02087	0.03928	0.01136	1.0333	0.0408
FCOMP, IEBDG	32K	0.0236	0.00960	2.0	0.0365	0.0163	1.0163	0.03010	0.0095	1 0286	0.0348

[a] Line size: 32 bytes for IBM 370 Traces, 16 byte lines for PDP-11 traces. Multiprogramming interval Q: 10K. Number of sets: 64.

well as the demand miss ratio for each of the sets of traces used and for a variety of memory sizes. We observe from this table that always prefetch and tagged prefetch are both very successful in reducing the miss ratio. Tagged prefetch has the significant additional benefit of requiring only a small increase in the access ratio over demand fetching. The transfer ratio is comparable for tagged prefetch and always prefetch.

It is important to note that taking advantage of the decrease in miss ratio obtained by these prefetch algorithms depends very strongly on the effectiveness of the implementation. For example, the Amdahl 470V/6-1 has a fairly sophisticated prefetch algorithm built in, but because of the design architecture of the machine, the benefit of prefetch cannot be realized. Although the prefetching cuts the miss ratio in this architecture, it uses too many cache cycles and interferes with normal program accesses to the cache. For that reason, prefetching is not used in the 470V/6-1 and is not available to customers. The more recent 470V/8, though, does contain a prefetch algorithm which is useful and improves machine performance. The V/8 cache prefetch algorithm prefetches (only) on misses, and was selected on the basis that it causes very little interference with normal machine operation. (Prefetch is implemented in the Dorado [CLAR81], but its success is not described.)

The prefetch implementation must at all times minimize its interference with regular machine functioning. For example, prefetch lookups should not block normal program memory accesses. This can be accomplished in three ways: (1) by instituting a second, parallel port to the cache, (2) by deferring prefetches until spare cache cycles are available, or (3) by not repeating recent prefetches. (Repeat prefetches can be eliminated by remembering the addresses of the last n prefetches in a small auxiliary cache. A potential prefetch could be tested against this buffer and not issued if found. This should cut the number of prefetch lookups by 80 to 90 percent for small n).

The move-in (transfer of a line from main to cache memory) and move-out (transfer from cache to main memory) required by a prefetch transfer can be buffered for performance during otherwise idle cache cycles. The main memory busy time engendered by a prefetch transfer seems unavoidable, but is not a serious problem. Also unavoidable is the fact that a prefetch may not be complete by the time the prefetched line is actually needed. This effect was examined in SMIT78b and was found to be minor although noticeable. Further comments and details of a suggested implementation are found in SMIT78b.

We note briefly that it is possible to consider the successful use of prefetching as an indication that the line size is too small; prefetch functions much as a larger line size would. A comparison of the results in Figure 6 and Table I with those in Figures 15–21 shows that prefetching on misses for 32-byte lines gives slightly better results than doubling the line size to 64 bytes. Always prefetching and tagged prefetch are both significantly better than the larger line size without prefetching. Therefore, it would appear that prefetching has benefits in addition to those that it provides by simulating a larger line size.

2.2 Placement Algorithm

The cache itself is not a user-addressable memory, but serves only as a buffer for main memory. Thus in order to locate an element in the cache, it is necessary to have some function which maps the main memory address into a cache location, or to search the cache associatively, or to perform some combination of these two. The placement algorithm determines the mapping function from main memory address to cache location.

The most commonly used form of placement algorithm is called set-associative mapping. It involves organizing the cache into S sets of E elements per set (see Figure 7). Given a memory address $r(i)$, a function f will map $r(i)$ into a set $s(i)$, so that $f(r(i)) = s(i)$. The reason for this type of organization may be oberved by letting either S or E become one. If S becomes one, then the cache becomes a fully associative memory. The problem is that the large number of lines in a cache would make a fully associative memory both slow and very expensive. (Our comments here apply

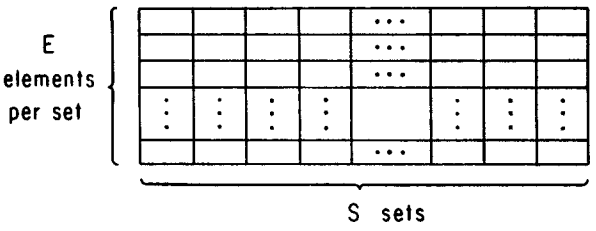

Figure 7. The cache is organized as S sets of E elements each.

to non-VLSI implementations. VLSI MOS facilitates broad associative searches.) Conversely, if E becomes one, in an organization known as direct mapping [CONT69], there is only one element per set. (A more general classification has been proposed by HARD75.) Since the mapping function f is many to one, the potential for conflict in this latter case is quite high: two or more currently active lines may map into the same set. It is clear that, on the average, the conflict and miss ratio decline with increasing E, (as $S * E$ remains constant), while the cost and access time increase. An effective compromise is to select E in the range of 2 to 16. Some typical values depending on certain cost and performance tradeoffs, are: 2 (Amdahl 470V/6-1, VAX 11/780, IBM 370/158-1), 4 (IBM 370/168-1, Amdahl 470V/8, Honeywell 66/80, IBM 370/158-3), 8 (IBM 370/168-3, Amdahl 470V/7), 16 (IBM 3033).

Another placement algorithm utilizes a *sector buffer* [CONT68], as in the IBM 360/85. In this machine, the cache is divided into 16 sectors of 1024 bytes each. When a word is accessed for which the corresponding sector has not been allocated a place in the cache (the sector search being fully associative), a sector is made available (the LRU sector—see Section 2.4), and a 64-byte block containing the information referenced is transferred. When a word is referenced whose sector is in the cache but whose block is not, the block is simply fetched. The hit ratio for this algorithm is now generally known to be lower than that of the set-associative organization (Private Communication: F. Bookett) and hence we do not consider it further. (This type of design may prove appropriate for on-chip microprocessor caches, since the limiting factor in many microprocessor systems is bus bandwidth. That topic is currently under study.)

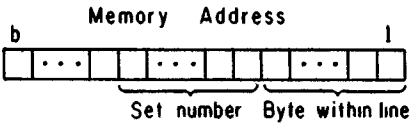

Figure 8. The set is selected by the low-order bits of the line number.

There are two aspects to selecting a placement algorithm for the cache. First, the number of sets S must be chosen while $S * E = M$ remains constant, where M is the number of lines in the cache. Second, the mapping function f, which translates a main memory address into a cache set, must be specified. The second question is most fully explored by SMIT78a; we summarize those results and present some new experimental measurements below. A number of other papers consider one or both of these questions to some extent, and we refer to reader to those [CAMP76, CONT68, CONT69, FUKU77, KAPL73, LIPT68, MATT71, STRE76, THAK78] for additional information.

2.2.1 Set Selection Algorithm

Several possible algorithms are used or have been proposed for mapping an address into a set number. The simplest and most popular is known as bit selection, and is shown in Figure 8. The number of sets S is chosen to be a power of 2 (e.g., $S = 2^k$). If there are 2^j bytes per line, the j bits $1 \cdots j$ select the byte within the line, and bits $j+1 \cdots j+k$ select the set. Performing the mapping is thus very simple, since all that is required is the decoding of a binary quantity. Bit-selection is used in all com-

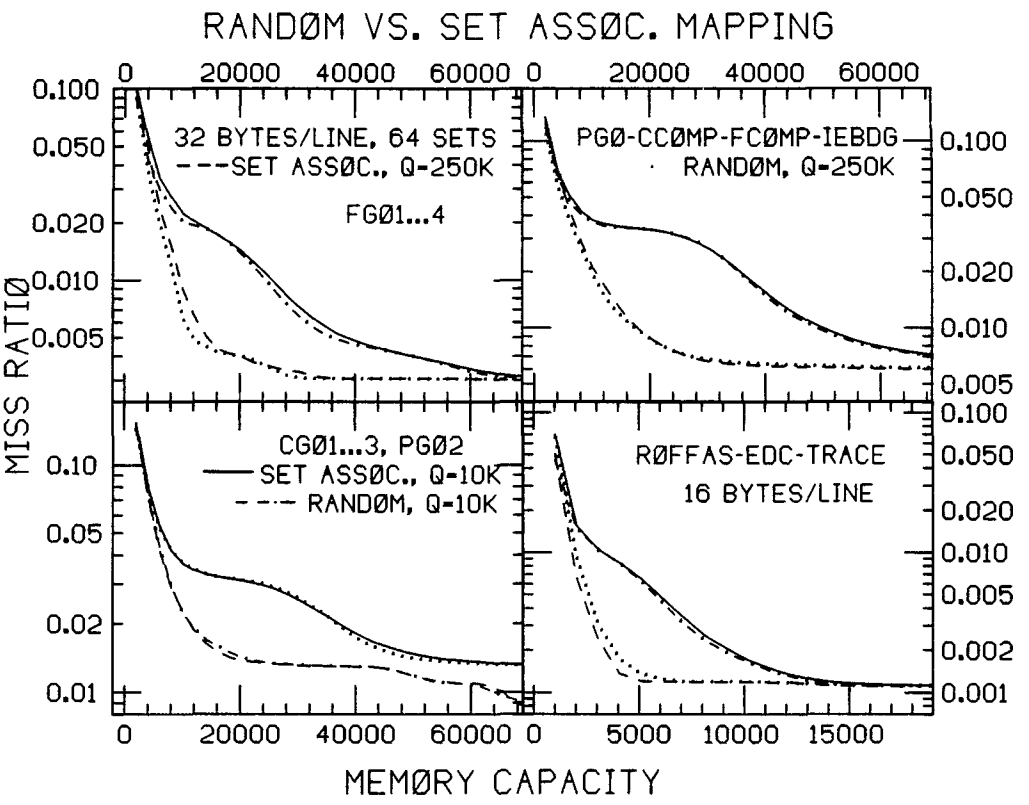

Figure 9. Comparison of miss ratios when using random or bit-selection set-associative mapping.

puters to our knowledge, including particularly all Amdahl and IBM computers.

Some people have suggested that because bit selection is not random, it might result in more conflict than a random algorithm, which employs a pseudorandom calculation (hashing) to map the line into a set. It is difficult to generate random numbers quickly in hardware, and the usual suggestion is some sort of *folding* of the address followed by exclusive oring of the bits. That is, if bits $j + 1 \cdots b$ are available for determining the line location, then these $b - j$ bits are grouped into k groups, and within each group an Exclusive Or is performed. The resulting k bits then designate a set. (This algorithm is used in TLBs—see Section 2.16.) In our simulations discussed later, we have used a randomizing function of the form $s(i) = a * r(i) \bmod 2^k$.

Simulations were performed to compare random and set-associative mapping. The results are shown in Figure 9. (32 bytes is the line size used in all cases except the PDP-11 traces, for which 16 bytes are used.) It can be observed that random mapping seems to have a small advantage in most cases, but that the advantage is not significant. Random mapping would probably be preferable to bit-selection mapping if it could be done equally quickly and inexpensively, but several extra levels of logic appear to be necessary. Therefore, bit selection seems to be the most desirable algorithm.

2.2.2 Set Size and the Number of Sets

There are a number of considerations in selecting values for the number of sets (S) and the set size (E). (We note that S and E are inversely related in the equation $S * E = M$, where M is the number of lines in the cache ($M = 2^m$).) These considerations have to do with lookup time, expense, miss ratio, and addressing. We discuss each below.

The first consideration is that most cache memories (e.g., Amdahl, IBM) are addressed using the real address of the data,

although the CPU produces a virtual address. The most common mechanism for avoiding the time to translate the virtual address to a real one is to overlap the cache lookup and the translation operation (Figures 1 and 2). We observe the following: the only address bits that get translated in a virtual memory system are the ones that specify the page address; the bits that specify the byte within the page are invariant with respect to virtual memory translation. Let there be 2^j bytes per line and 2^k sets in the cache, as before. Also, let there be 2^p bytes per page. Then (assuming bit-selection mapping), $p - j$ bits are immediately available to choose the set. If $(p - j) \geq k$, then the set can be selected immediately, before translation; if $(p - j) < k$, then the search for the cache line can only be narrowed down to a small number $2^{(k-p+j)}$ of sets. It is quite advantageous if the set can be selected before translation (since the associative search can be started immediately upon completion of translation); thus there is a good reason to attempt to keep the number of sets less than or equal to $2^{(p-j)}$. (We note, though, that there is an alternative. The Amdahl 470V/6 has 256 sets, but only 6 bits immediately available for set selection. The machine reads out both elements of each of the four sets which could possibly be selected, then after the translation is complete, selects one of those sets before making the associative search. See also LEE80.)

Set size is just a different term for the scope of the associative search. The smaller the degree of associative search, the faster and less expensive the search (except, as noted above, for MOS VLSI). This is because there are fewer comparators and signal lines required and because the replacement algorithm can be simpler and faster (see Section 2.4). Our second consideration, expense, suggests that therefore the smaller the set size, the better. We repeat, though, that the set size and number of sets are inversely related. If the number of sets is less than or equal to $2^{(p-j)}$, then the set size is greater than or equal to $2^{(m-p+j)}$ lines.

The third consideration in selecting set size is the effect of the set size on the miss ratio. In [SMIT78a] we developed a model for this effect. We summarize the results of that model here, and then we present some new experimental results.

A commonly used model for program behavior is what is known as the LRU Stack Model. (See COFF73 for a more thorough explanation and some basic results.) In this model, the pages or lines of the program's address space are arranged in a list, with the most recently referenced line at the top of the list, and with the lines arranged in decreasing recency of reference from top to bottom. Thus, referencing a line moves it to the top of the list and moves all of the lines between the one referenced and the top line down one position. A reference to a line which is the ith line in the list (stack) is referred to as a stack distance of i. This model assumes that the stack distances are independently and identically drawn from a distribution $\{q(j)\}, j = 1, \ldots, n$. This model has been shown not to hold in a formal statistical sense [LEWI71, LEWI73], but the author and others have used this model with good success in many modeling efforts.

Each set in the cache constitutes a separate associative memory. If each set is managed with LRU replacement, it is possible to determine the probability of referencing the kth most recently used item in a given set as a function of the overall LRU stack distance probability distribution $\{q(j)\}$. Let $p(i, S)$ be the probability of referencing the ith most recently referenced line in a set, given S sets. Then, we show that $p(i, S)$ may be calculated from the $\{q(i)\}$ with the following formula:

$$p(i, S) = \sum_{j=i}^{\infty} q_j \, (1/S)^{i-1} \, (S - 1/S)^{j-i} \binom{j-1}{i-1}$$

Note that $p(i, 1) = q(i)$. In SMIT78a this model was shown to give accurate predictions of the effect of set size.

Experimental results are provided in Figures 10–14. In each of these cases, the number of sets has been varied. The rather curious shape of the curves (and the similarities between different plots) has to do with the task-switch interval and the fact that round-robin scheduling was used. Thus, when a program regained control of the processor, it might or might not, depending on the memory capacity, find any of its working set still in the cache.

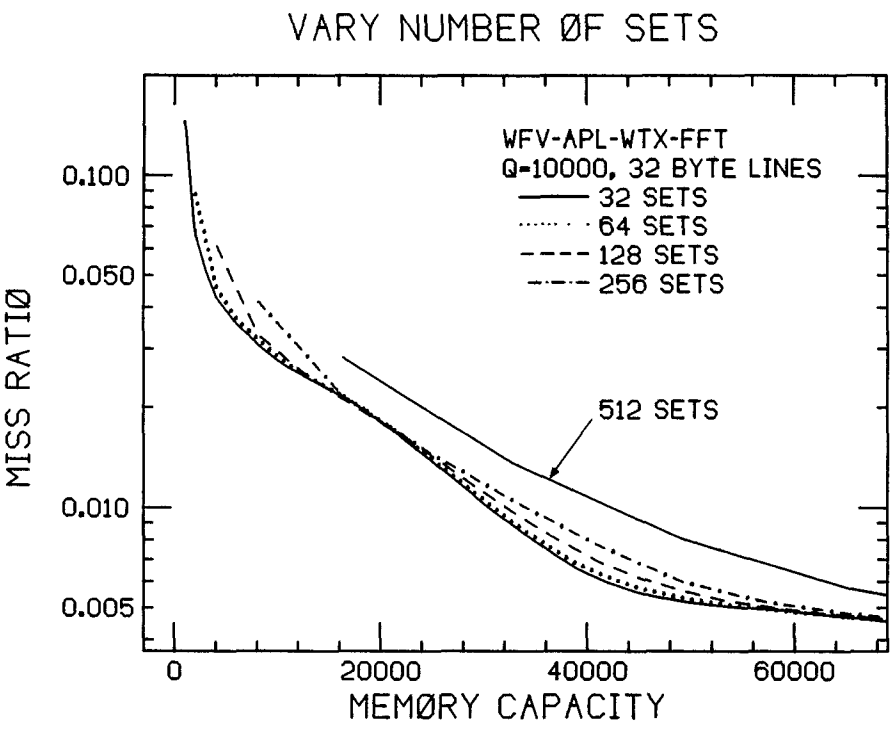

Figure 10. Miss ratios as a function of the number of sets and memory capacity.

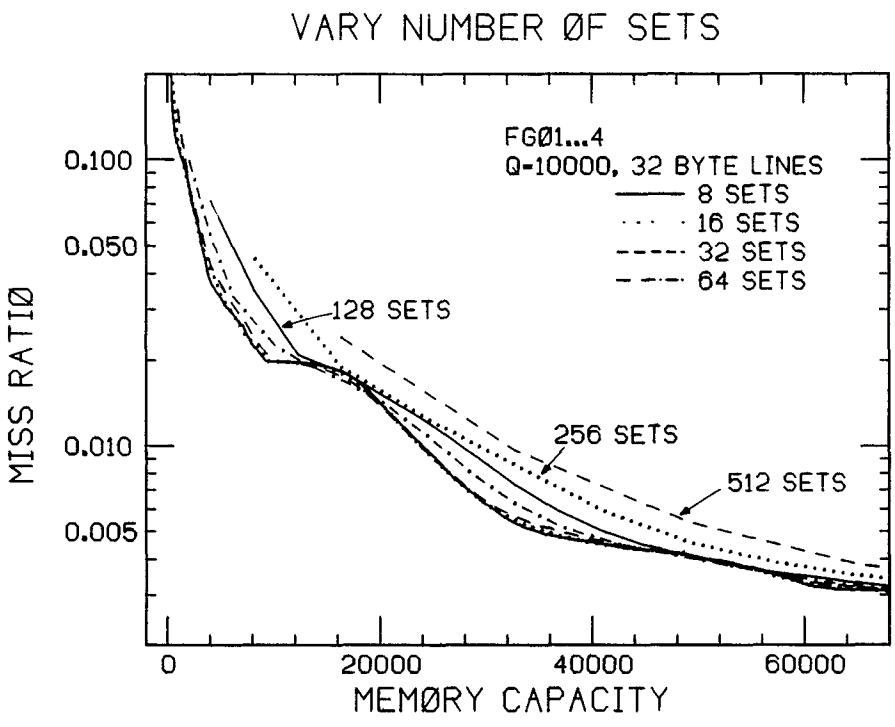

Figure 11. Miss ratios as a function of the number of sets and memory capacity

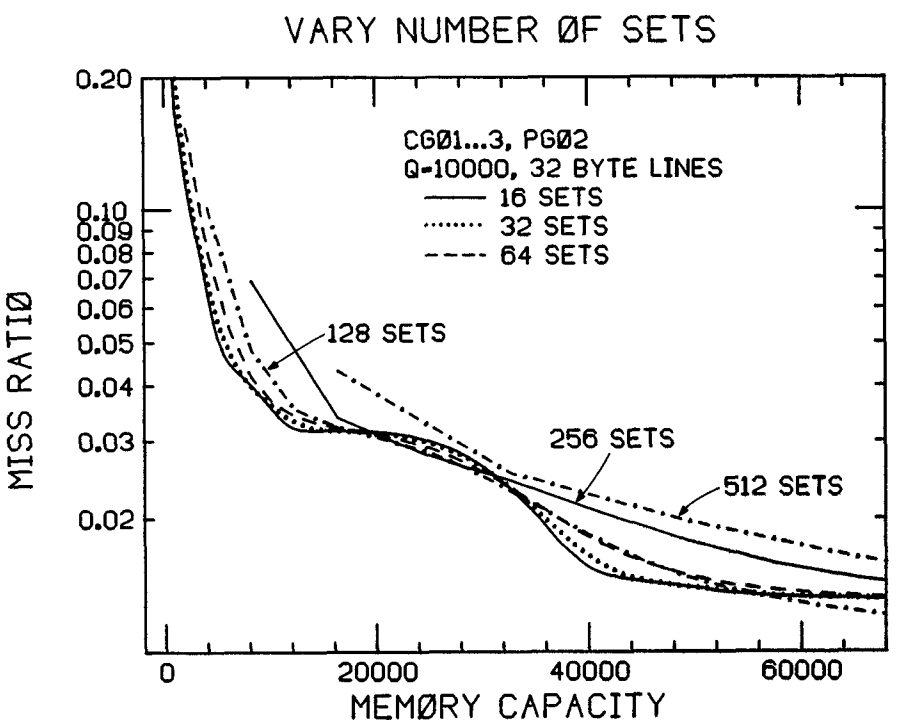

Figure 12. Miss ratios as a function of the number of sets and memory capacity.

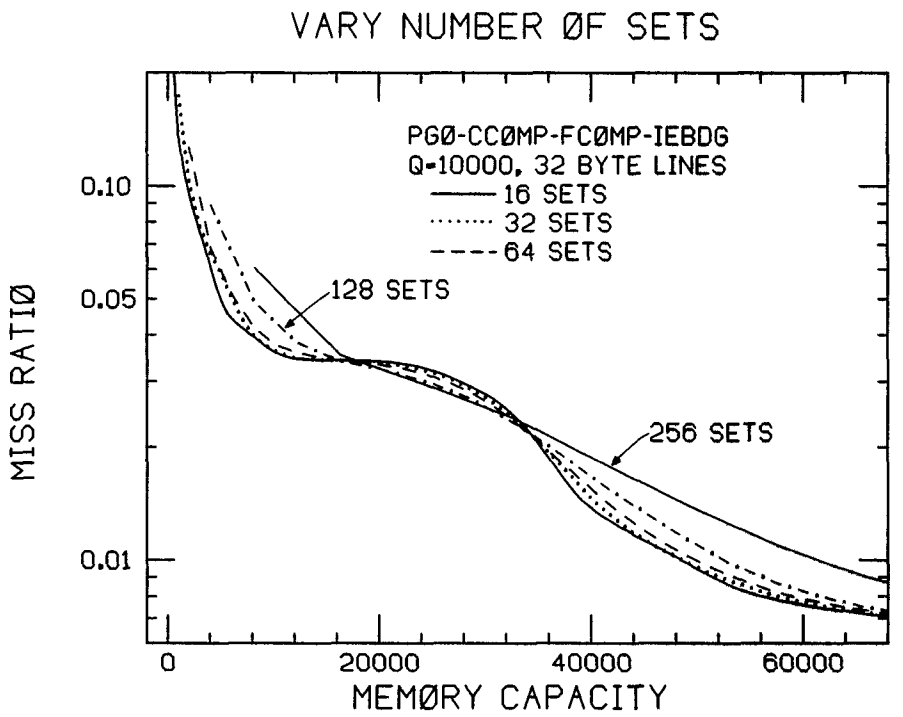

Figure 13. Miss ratios as a function of the number of sets and memory capacity.

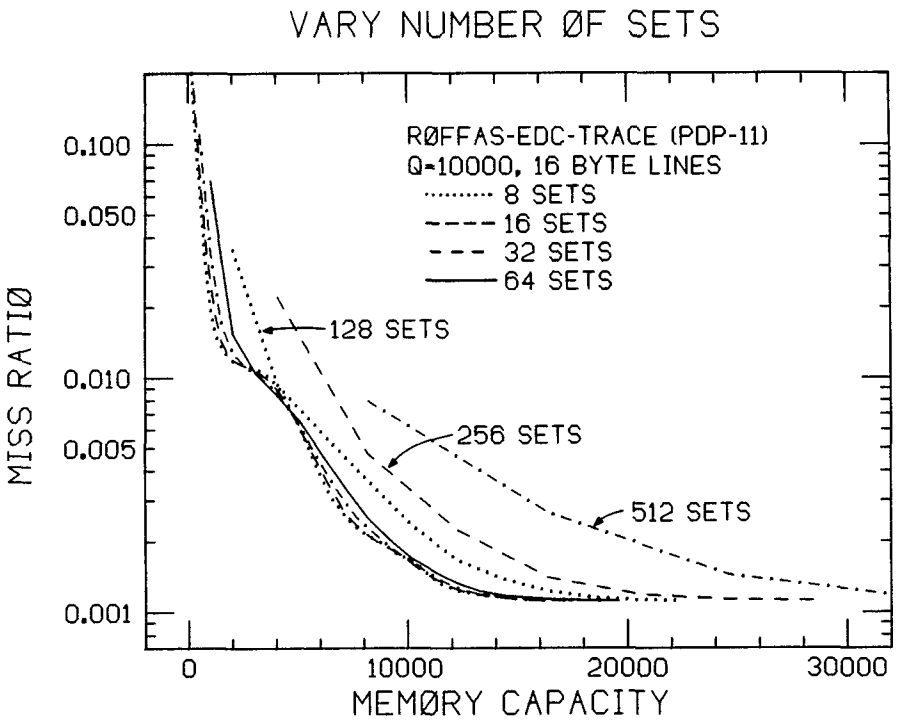

Figure 14. Miss ratios as a function of the number of sets and memory capacity.

Based on Figures 10–14, Figure 33, and the information given by SMIT78a, we believe that the minimum number of elements per set in order to get an acceptable miss ratio is 4 to 8. Beyond 8, the miss ratio is likely to decrease very little if at all. This has also been noted for TLB designs [SATY81]. The issue of maximum feasible set size also suggests that a set size of more than 4 or 8 will be inconvenient and expensive. The only machine known to the author with a set size larger than 8 is the IBM 3033 processor. The reason for such a large set size is that the 3033 has a 64-kbyte cache and 64-byte lines. The page size is 4096 bytes, which leaves only 6 bits for selecting the set, if the translation is to be overlapped. This implies that 16 lines are searched, which is quite a large number. The 3033 is also a very performance-oriented machine, and the extra expense is apparently not a factor.

Values for the set size (number of elements per set) and number of sets for a number of machines are as follows: Amdahl 470V/6 (2, 256), Amdahl 470V/7 (8, 128), Amdahl 470V/8 (4, 512), IBM 370/168-3 (8, 128), IBM 3033 (16, 64), DEC PDP-11/70 (2, 256), DEC VAX 11/780 (2, 512), Itel AS/6 (4, 128) [ROSS78], Honeywell 66/60 and 66/80 (4, 128) [DIET74].

2.3 Line Size

One of the most visible parameters to be chosen for a cache memory is the line size. Just as with paged main memory, there are a number of trade-offs and no single criterion dominates. Below we discuss the advantages of both small and large line sizes. Additional information relating to this problem may be found in other papers [ALSA78, ANAC67, GIBS67, KAPL73, MATT71, MEAD70, and STRE76].

Small line sizes have a number of advantages. The transmission time for moving a small line from main memory to cache is obviously shorter than that for a long line, and if the machine has to wait for the full transmission time, short lines are better. (A high-performance machine will use fetch bypass; see Section 2.1.1.) The small line is less likely to contain unneeded information; only a few extra bytes are brought in along

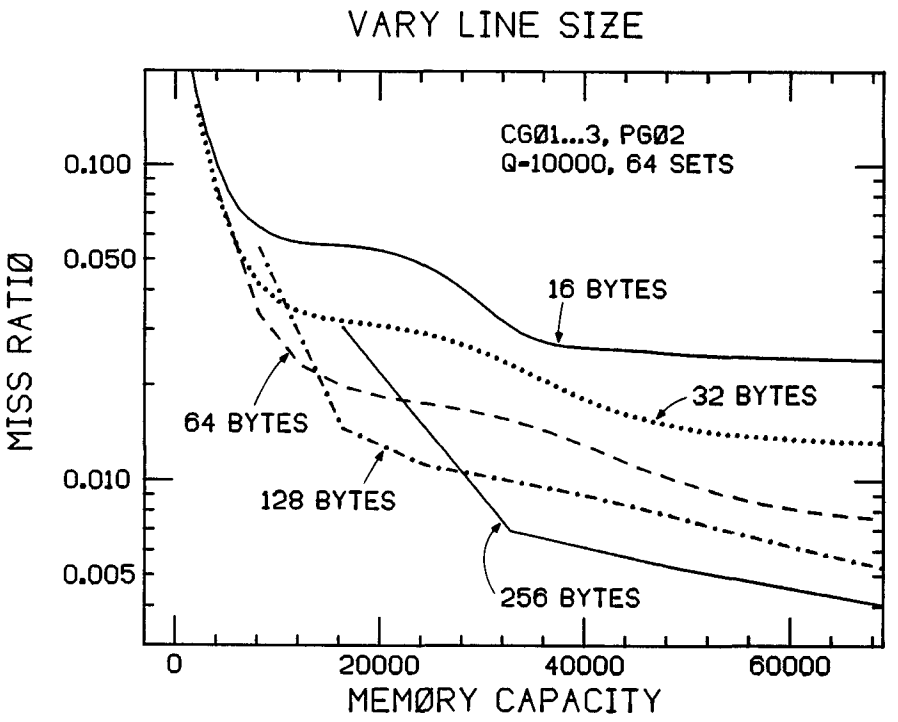

Figure 15. Miss ratios as a function of the line size and memory capacity.

with the actually requested information. The data width of main memory should usually be at least as wide as the line size, since it is desirable to transmit an entire line in one main memory cycle time. Main memory width can be expensive, and short lines minimize this problem.

Large line sizes, too, have a number of advantages. If more information in a line is actually being used, fetching it all at one time (as with a long line) is more efficient. The number of lines in the cache is smaller, so there are fewer logic gates and fewer storage bits (e.g., LRU bits) required to keep and manage address tags and replacement status. A larger line size permits fewer elements/set in the cache (see Section 2.2), which minimizes the associative search logic. Long lines also minimize the frequency of "line crossers," which are requests that span the contents of two lines. Thus in most machines, this means that two separate fetches are required within the cache (this is invisible to the rest of the machine.)

Note that the advantages cited above for both long and short lines become disadvantages for the other.

Another important criterion for selecting a line size is the effect of the line size on the miss ratio. The miss ratio, however, only tells part of the story. It is inevitable that longer lines make processing a miss somewhat slower (no matter how efficient the overlapping and buffering), so that translating a miss ratio into a measure of machine speed is tricky and depends on the details of the implementation. The reader should bear this in mind when examining our experimental results.

Figures 15–21 show the miss ratio as a function of line size and cache size for five different sets of traces. Observe that we have also varied the multiprogramming quantum time Q. We do so because the miss ratio is affected by the task-switch interval nonuniformly with line size. This nonuniformity occurs because long line sizes load up the cache more quickly. Consider two cases. First, assume that most cache misses result from task switching. In this case, long lines load up the cache more

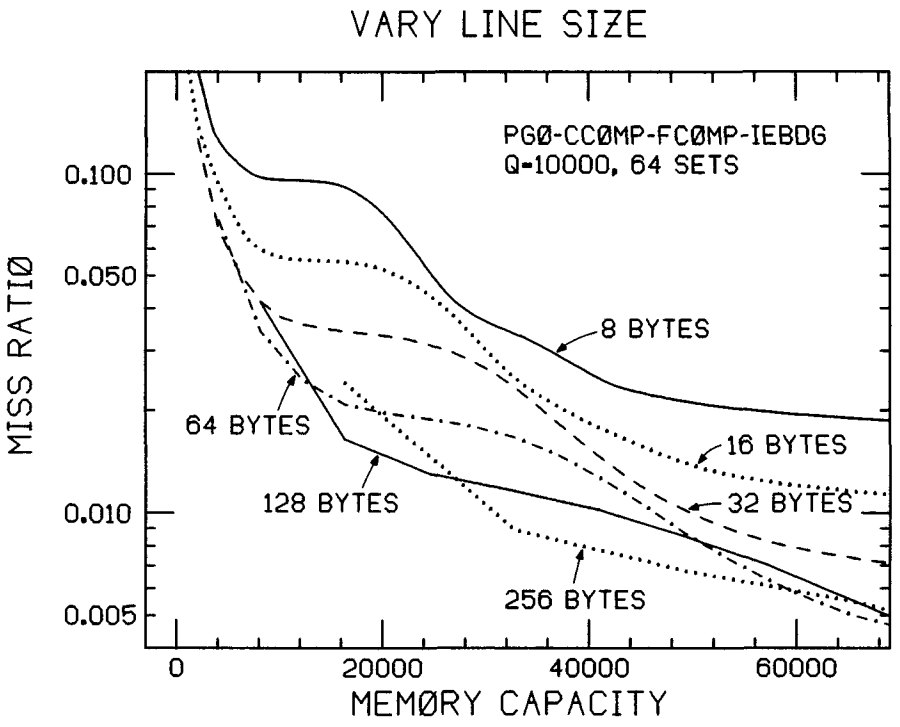

Figure 16. Miss ratios as a function of the line size and memory capacity.

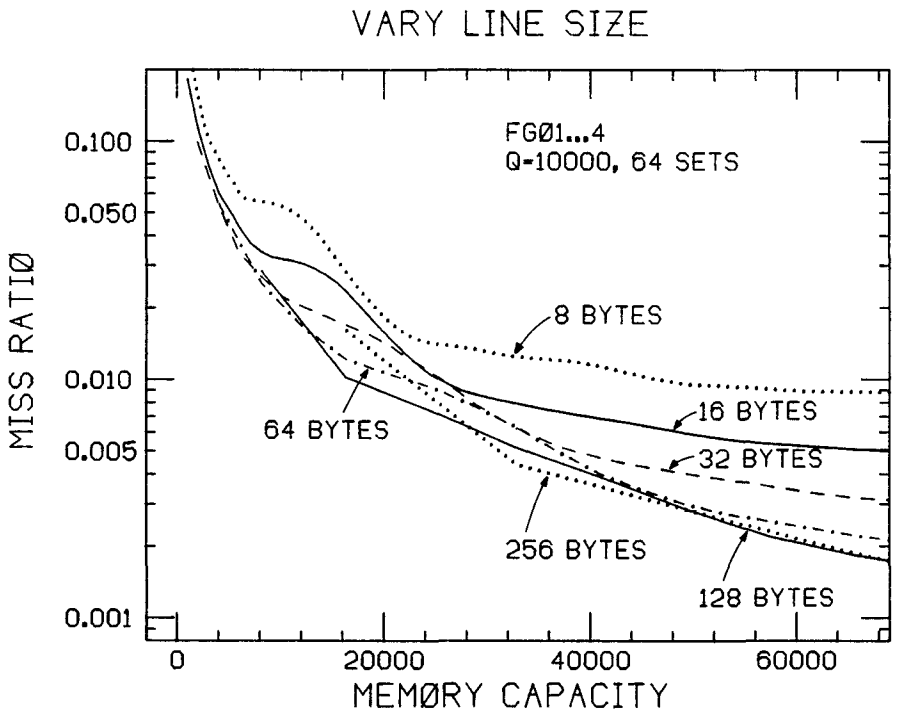

Figure 17. Miss ratios as a function of the line size and memory capacity.

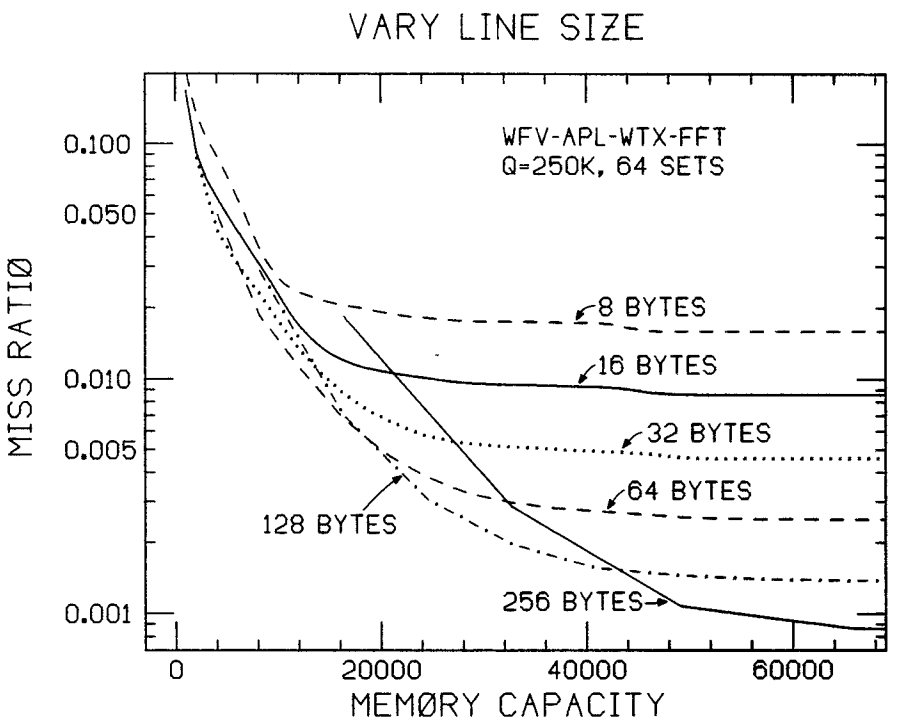

Figure 18. Miss ratios as a function of the line size and memory capacity.

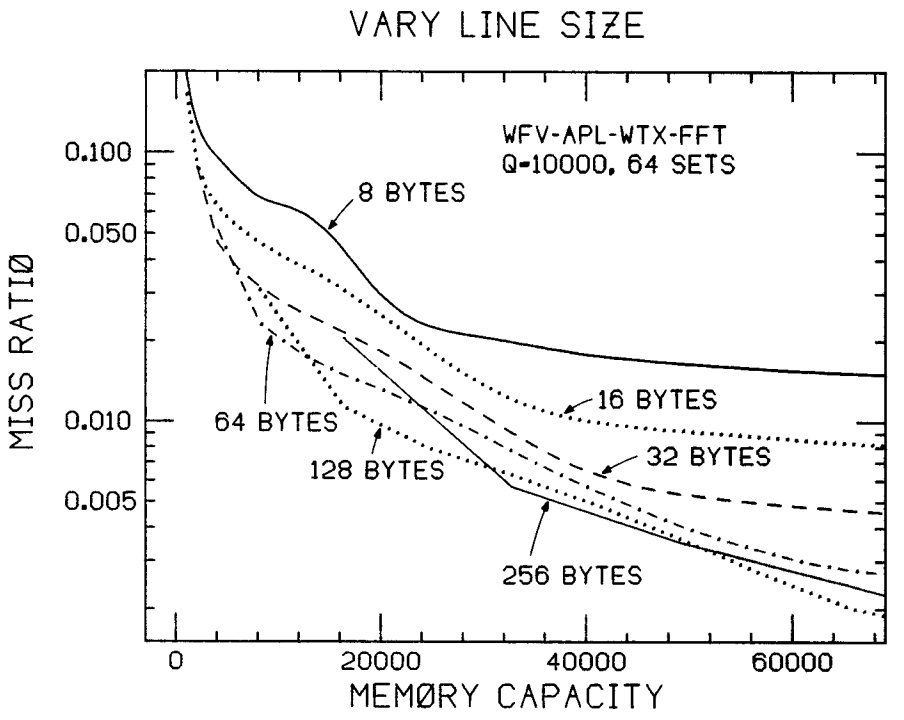

Figure 19. Miss ratios as a function of the line size and memory capacity.

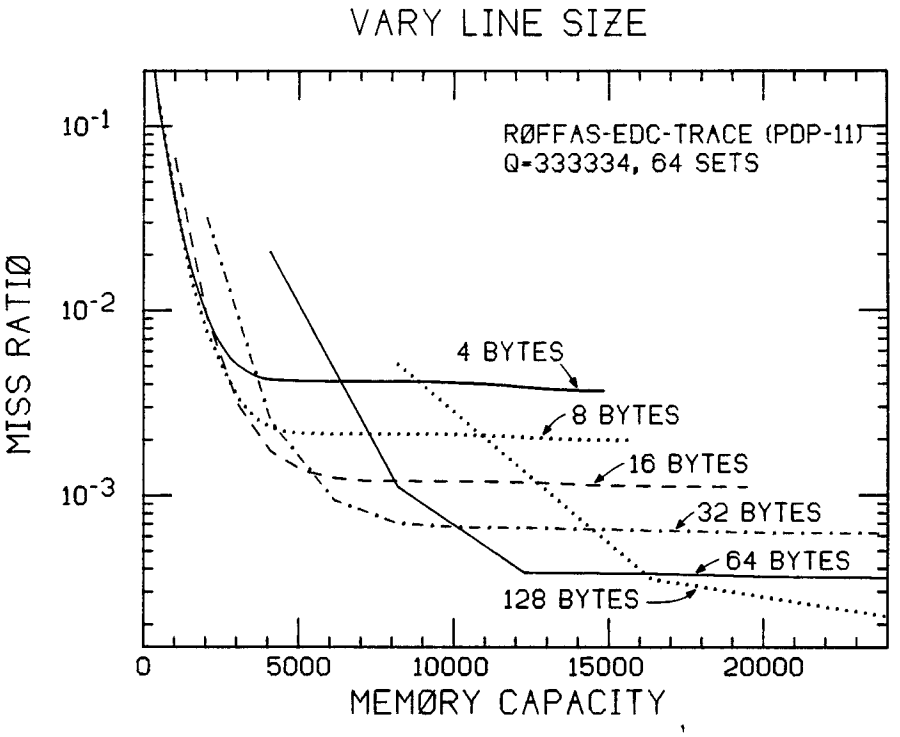

Figure 20. Miss ratios as a function of the line size and memory capacity.

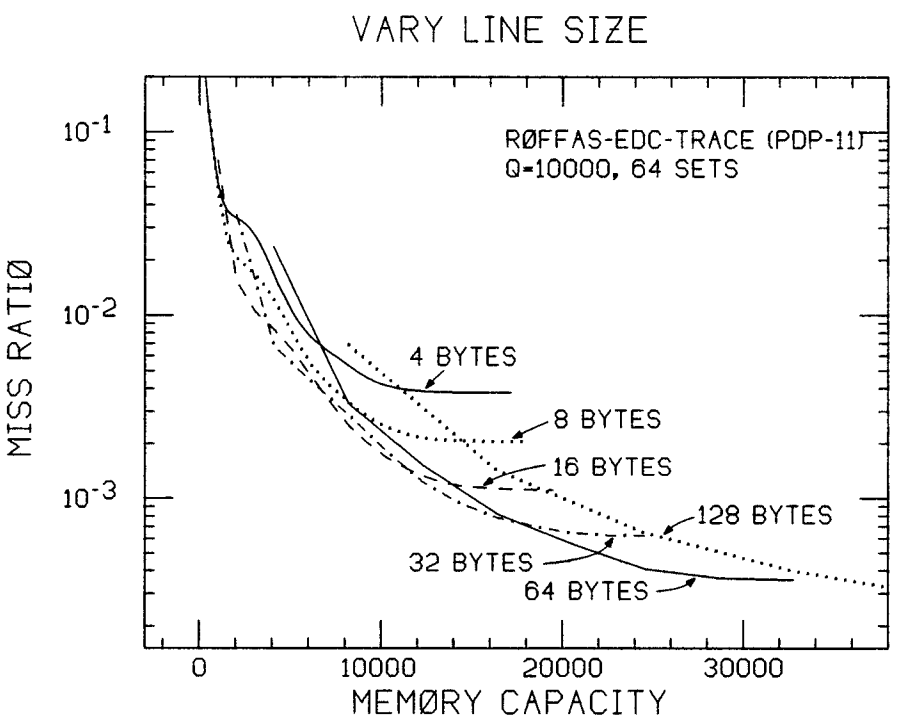

Figure 21. Miss ratios as a function of the line size and memory capacity.

quickly than small ones. Conversely, assume that most misses occur in the steady state; that is, that the cache is entirely full most of the time with the current process and most of the misses occur in this state. In this latter case, small lines cause less memory pollution and possibly a lower miss ratio. Some such effect is evident when comparing Figures 18 and 19 with Figures 20 and 21, but explanation is required. A quantum of 10,000 results not in a program finding an empty cache, but in it finding some residue of its previous period of activity (since the degree of multiprogramming is only 3 or 4); thus small lines are relatively more advantageous in this case than one would expect.

The most interesting number to be gleaned from Figures 15–21 is the line size which causes the minimum miss ratio for each memory size. This information has been collected in Table 2. The consistency displayed there for the 360/370 traces is surprising; we observe that one can divide the cache size by 128 or 256 to get the minimum miss ratio line size. This rule does not apply to the PDP-11 traces. Programs written for the PDP-11 not only use a different instruction set, but they have been written to run in a small (64K) address space. Without more data, generalizations from Figures 20 and 21 cannot be made.

In comparing the minimum miss ratio line sizes suggested by Table 2 and the offerings of the various manufacturers, one notes a discrepancy. For example, the IBM 168-1 (32-byte line, 16K buffer) and the 3033 (64-byte line, 64K buffer) both have surprisingly small line sizes. The reason for this is almost certainly that the transmission time for longer lines would induce a performance penalty, and the main memory data path width required would be too large and therefore too expensive.

Kumar [KUMA79] also finds that the line sizes in the IBM 3033 and Amdahl 470 are too small. He creates a model for the working set size w of a program, of the form $w(k) = c/k^a$, where k is the block size, and c and a are constants. By making some convenient assumptions, Kumar derives from this an expression for the miss ratio as a function of the block size. Both expressions are verified for three traces, and a is measured to be in the range of 0.45 to 0.85 over the three traces and various working set window sizes. He then found that for those machines, the optimum block size lies in the range 64 to 256 bytes.

Table 2. Line Size (in bytes) Giving Minimum Miss Ratio, for Given Memory Capacity and Traces

Traces	Quantum	Memory size (kbytes)	Minimum miss ratio line size (bytes)
CGO1	10,000	4	32
CGO2		8	64
CGO3		16	128
PGO2		32	256
		64	256
PGO	10,000	4	32
CCOMP		8	64
FCOMP		16	128
IEBDG		32	256
		64	256
FGO1	10,000	4	32
FGO2		8	64
FGO3		16	128
FGO4		32	256
		64	128
WFV	10,000	4	32
APL		8	64
WTX		16	128
FFT		32	256
		64	128
	250,000	4	32
		8	64
		16	64
		32	128
		64	256
ROFFAS	10,000	2	16
EDC		4	32
TRACE		8	16
		16	32
	333,333	2	8
		4	16
		8	32
		16	64

It is worth considering the relationship between prefetching and line size. Prefetching can function much as a larger line size would. In terms of miss ratio, it is usually even better; although a prefetched line that is not being used can be swapped out, a half of a line that is not being used cannot be removed independently. Comparisons between the results in Section 2.1 and this section show that the performance improvement from prefetching is significantly larger than that obtained by doubling the line size.

Line sizes in use include: 128 bytes (IBM 3081 [IBM82]), 64 bytes (IBM 3033), 32 bytes (Amdahl 470s, Itel AS/6, IBM 370/

168), 16 bytes (Honeywell 66/60 and 66/80), 8 bytes (DEC VAX 11/780), 4 bytes (PDP-11/70).

2.4 Replacement Algorithm

2.4.1 Classification

In the steady state, the cache is full, and a cache miss implies not only a fetch but also a replacement; a line must be removed from the cache. The problem of replacement has been studied extensively for paged main memories (see SMIT78d for a bibliography), but the constraints on a replacement algorithm are much more stringent for a cache memory. Principally, the cache replacement algorithm must be implemented entirely in hardware and must execute very quickly, so as to have no negative effect on processor speed. The set of feasible solutions is still large, but many of them can be rejected on inspection.

The usual classification of replacement algorithms groups them into usage-based versus non-usage-based, and fixed-space versus variable-space. Usage-based algorithms take the record of use of the line (or page) into account when doing replacement; examples of this type of algorithm are LRU (least recently used) [COFF73] and Working Set [DENN68]. Conversely, non-usage-based algorithms make the replacement decision on some basis other than and not related to usage; FIFO and Rand (random or pseudorandom) are in this class. (FIFO could arguably be considered usage-based, but since reuse of a line does not improve the replacement status of that line, we do not consider it as being such.) Fixed-space algorithms assume that the amount of memory to be allocated is fixed; replacement simply consists of selecting a specific line. If the algorithm varies the amount of space allocated to a specific process, it is known as a variable-space algorithm, in which case, a fetch does not imply a replacement, and a swap-out can take place without a corresponding fetch. Working Set and Page Fault Frequency [CHU76] are variable-space algorithms.

The cache memory is fixed in size, and it is usually too small to hold the working set of more than one process (although the 470V/8 and 3033 may be exceptions). For this reason, we believe that variable-space algorithms are not suitable for a cache memory. To our knowledge, no variable-space algorithm has ever been used in a cache memory.

It should also be clear that in a set-associative memory, replacement must take place in the same set as the fetch. A line is being added to a given set because of the fetch, and thus a line must be removed. Since a line maps uniquely into a set, the replaced line in that set must be entirely removed from the cache.

The set of acceptable replacement algorithms is thus limited to fixed-space algorithms executed within each set. The basic candidates are LRU, FIFO, and Rand. It is our experience (based on prior experiments and on material in the literature) that non-usage-based algorithms all yield comparable performance. We have chosen FIFO within set as our example of a non-usage-based algorithm.

2.4.2 Comparisons

Comparisons between FIFO and LRU appear in Table 3, where we show results based on each set of traces for varying memory sizes, quantum sizes, and set numbers. We found (averaging over all of the numbers there) that FIFO yields a miss ratio approximately 12 percent higher than LRU, although the ratios of FIFO to LRU miss ratio range from 0.96 to 1.38. This 12 percent difference is significant in terms of performance, and LRU is clearly a better choice if the cost of implementing LRU is small and the implementation does not slow down the machine. We note that in minicomputers (e.g., PDP-11) cost is by far the major criterion; consequently, in such systems, this performance difference may not be worthwhile. The interested reader will find additional performance data and discussion in other papers [CHIA75, FURN78, GIBS67, LEE69, and STRE76].

2.4.3 Implementation

It is important to be able to implement LRU cheaply, and so that it executes quickly; the standard implementation in software using linked lists is unlikely to be either cheap or fast. For a set size of two,

Table 3. · Miss Ratio Comparison for FIFO and LRU (within set) Replacement

Memory size (kbytes)	Quantum size	Number of sets	Miss ratio LRU	Miss ratio FIFO	Ratio FIFO/ LRU	Traces
16	10K	64	0.02162	0.02254	1.04	WFV
32	10K	64	0 00910	0.01143	1.26	APL
16	250K	64	0 00868	0.01036	1.19	WTX
32	250K	64	0.00523	0.00548	1.05	FFT
16	10K	256	0.02171	0.02235	1.03	
32	10K	256	0.01057	0.01186	1.12	
4	10K	64	0 00845	0.00947	1.12	ROFFAS
8	10K	64	0 00255	0.00344	1.35	EDC
4	333K	64	0.00173	0.00214	1.24	TRACE
8	333K	64	0.00120	0.00120	1.00	
4	10K	256	0.0218	0.02175	0.998	
8	10K	256	0.00477	0 00514	1.08	
4	333K	256	0.01624	0.01624	1.00	
8	333K	256	0.00155	0.00159	1.03	
64	10K	64	0.01335	0.01839	1.38	CGO1
128	10K	64	0.01147	0.01103	0.96	CGO2
64	10K	256	0.01461	0.01894	1.30	CGO3
128	10K	256	0.01088	0.01171	1.08	PGO2
16	10K	64	0 01702	0.01872	1.10	FGO1
32	10K	64	0.00628	0.00867	1.38	FGO2
16	10K	256	0.01888	0.01934	1.02	FGO3
32	10K	256	0.00857	0.00946	1.10	FGO4
16	10K	64	0.03428	0.03496	1.02	PGO1
32	10K	64	0.02356	0.02543	1.08	CCOMP
16	10K	256	0 03540	0.03644	1.03	FCOMP
32	10K	256	0.02394	0.02534	1.06	IEBDG
Average					1.116	

only a hot/cold (toggle) bit is required. More generally, replacement in a set of E elements can be effectively implemented with $E(E-1)/2$ bits of status. (We note that $\lceil \log 2E! \rceil$ bits of status are the theoretical minimum.) One creates an upper-left triangular matrix (without the diagonal, that is, $i + j < E$) which we will call R and refer to as $R(i,j)$. When line i is referenced, row i of $R(i,j)$ is set to 1, and column i of $R(j,i)$ is set to 0. The LRU line is the one for which the row is entirely equal to 0 (for those bits in the row; the row may be empty) and for which the column is entirely 1 (for all the bits in the column; the column may be empty). This algorithm can be easily implemented in hardware, and executes rapidly. See MARU75 for an extension and MARU76 for an alternative.

The above algorithm requires a number of LRU status bits that increases with the square of the set size. This number is acceptable for a set size of 4 (470V/8, Itel AS/6), marginal for a set size of eight (470V/7), and unacceptable for a set size of 16. For that reason, IBM has chosen to implement approximations to LRU in the 370/168 and the 3033. In the 370/168-3 [IBM75], the set size is 8, with the 8 lines grouped in 4 pairs. The LRU pair of lines is selected, and then the LRU block of the pair is the one used for replacement. This algorithm requires only 10 bits, rather than the 28 needed by the full LRU. A set size of 16 is found in the 3033 [IBM78]. The 16 lines that make up a set are grouped into four groups of two pairs of two lines. The line to be replaced is selected as follows: (1) find the LRU group of four lines (requiring 6 bits of status), (2) find the LRU pair of the two pairs (1 bit per group, thus 4 more bits), and (3) find the LRU line of that pair (1 bit per pair, thus 8 more bits). In all, 18 bits are used for this modified LRU algorithm, as opposed to the 120 bits required for a full LRU. No experiments have been published comparing these modified LRU algorithms with genuine LRU, but we would expect to find no measurable difference.

Implementing either FIFO or Rand is much easier than implementing LRU. FIFO is implemented by keeping a modulo

Table 4. Percentage of Memory References That Are Reads, Writes, and Instruction Fetches for Each Trace[a]

	Partial trace			Full trace		
Trace	Data read	Data write	IFETCH	Data read	Data write	IFETCH
WATFIV	23.3	16.54	60.12	—	15.89	—
APL	21.5	8.2	70.3	—	8.90	—
WATEX	24.5	9.07	66.4	—	7.84	—
FFT1	23.4	7.7	68.9	—	7.59	—
ROFFAS	37.5	4.96	57.6	38.3	5.4	56.3
EDC	30 4	10.3	59.2	29.8	11.0	59.2
TRACE	47.9	10.2	41.9	48.6	10.0	41.3
CGO1	41.5	34.2	24.3	42.07	34.19	23.74
CGO2	41.1	32.4	26.5	36.92	15.42	47.66
CGO3	37.7	22.5	39.8	37.86	22.55	39.59
PGO2	31.6	15.4	53.1	30.36	12.77	56.87
FGO1	29.9	17.6	52.6	30.57	11.26	58.17
FGO2	30 6	5.72	63 7	32.54	10.16	57.30
FGO3	30.0	12.8	57.2	30.60	13.25	56.15
FGO4	28.5	17.2	54.3	28 38	17.29	54.33
PGO1	29.7	19.8	50.5	28.68	16.93	54.39
CCOMP1	30.8	9.91	59.3	33.42	17.10	49.47
FCOMP1	29.5	20.7	50.0	30.80	15.51	53.68
IEBDG	39.3	28.1	32.7	39.3	28.2	32.5
Average	32.0	15.96	52 02	34.55	14.80	49.38
Stand. Dev.	7.1	8.7	13.4	5.80	7.21	10.56

[a] Partial trace results are for first 250,000 memory references for IBM 370 traces, and 333,333 memory references for PDP-11 traces. Full trace results refer to entire length of memory address trace (one to ten million memory percent references).

E (E elements/set) counter for each set; it is incremented with each replacement and points to the next line for replacement. Rand is simpler still. One possibility is to use a single modulo E counter, incremented in a variety of ways: by each clock cycle, each memory reference, or each replacement anywhere in the cache. Whenever a replacement is to occur, the value of the counter is used to indicate the replaceable line within the set.

2.5 Write-Through versus Copy-Back

When the CPU executes instructions that modify the contents of the current address space, those changes must eventually be reflected in main memory; the cache is only a temporary buffer. There are two general approaches to updating main memory: stores can be immediately transmitted to main memory (called write-through or store-through), or stores can initially only modify the cache, and can later be reflected in main memory (copy-back). There are issues of performance, reliability, and complexity in making this choice; these issues are discussed in this section. Further information can be found in the literature [AGRA77a, BELL74, POHM75, and RIS77]. A detailed analysis of some aspects of this problem is provided in SMIT79 and YEN81.

To provide an empirical basis for our discussion in this section, we refer the reader to Table 4. There we show the percentage of memory references that resulted from data reads, data writes, and instruction fetches for each of the traces used in this paper. The leftmost three columns show the results for those portions of the traces used throughout this paper; that is, the 370 traces were run for the first 250,000 memory references and the PDP-11 traces for 333,333 memory references. When available, the results for the entire trace (1 to 10 million memory references) are shown in the rightmost columns. The overall average shows 16 percent of the references were writes, but the variation is wide (5 to 34 percent) and the values observed are clearly very dependent on the source language and on the machine architecture. In SMIT79 we observed that the fraction of lines from the cache that had to be written back to main memory (in a copy-back cache) ranged from 17 to 56 percent.

Several issues bear on the trade-off between write-through and copy-back.

1. Main Memory Traffic. Copy-back almost always results in less main memory traffic since write-through requires a main memory access on every store, whereas copy-back only requires a store to main memory if the swapped out line (when a miss occurs) has been modified. Copy-back generally, though, results in the entire line being written back, rather than just one or two words, as would occur for each write memory reference (unless "dirty bits" are associated with partial lines; a dirty bit, when set, indicates the line has been modified while in the cache). For example, assume a miss ratio of 3 percent, a line size of 32 bytes, a memory module width of 8 bytes, a 16 percent store frequency, and 30 percent of all cache lines requiring a copy-back operation. Then the ratio of main memory store cycles to total memory references is 0.16 for write-through and 0.036 for copy-back.

2. Cache Consistency. If store-through is used, main memory always contains an up-to-date copy of all information in the system. When there are multiple processors in the system (including independent channels), main memory can serve as a common and consistent storage place, provided that additional mechanisms are used. Otherwise, either the cache must be shared or a complicated directory system must be employed to maintain consistency. This subject is discussed further in Section 2.7, but we note here that store-through simplifies the memory consistency problem.

3. Complicated Logic. Copy-back may complicate the cache logic. A dirty bit is required to determine when to copy a line back. In addition, arrangements have to be made to perform the copy-back before the fetch (on a miss) can be completed.

4. Fetch-on-write. Using either copy-back or write-through still leaves undecided the question of whether to fetch-on-write or not, if the information referenced is not in the cache. With copy-back, one will usually fetch-on-write, and with write-through, usually not. There are additional related possibilities and problems. For example, when using write-through, one could not only not fetch-on-write but one could choose actually to purge the modified line from the cache should it be found there. If the line is found in the cache, its replacement status (e.g., LRU) may or may not be updated. This is considered in item 6 below.

5. Buffering. Buffering is required for both copy-back and write-through. In copy-back, a buffer is required so that the line to be copied back can be held temporarily in order to avoid interfering with the fetch. One optimization worth noting for copy-back is to use spare cache/main memory cycles to do the copy-back of "dirty" (modified) lines [BALZ81b]. For write-through, it is important to buffer several stores so that the CPU does not have to wait for them to be completed. Each buffer consists of a data part (the data to be stored) and the address part (the target address). In SMIT79 it was shown that a buffer with capacity of four provided most of the performance improvement possible in a write-through system. This is the number used in the IBM 3033. We note that a great deal of extra logic may be required if buffering is used. There is not only the logic required to implement the buffers, but also there must be logic to test all memory access addresses and match them against the addresses in the address part of the buffers. That is, there may be accesses to the material contained in the store buffers before the data in those buffers has been transferred to main memory. Checks must be made to avoid possible inconsistencies.

6. Reliability. If store-through is used, main memory always has a valid copy of the total memory state at any given time. Thus, if a processor fails (along with its cache), a store-through system can often be restored more easily. Also, if the only valid copy of a line is in the cache, an error-correcting code is needed there. If a cache error can be corrected from main memory, then a parity check is sufficient in the cache.

Some experiments were run to look at the miss ratio for store-through and copy-back. A typical example is shown in Figure 22; the other sets of traces yield very similar results. (In the case of write-through, we have counted each write as a miss.) It is clear that write-through always produces a much higher miss ratio. The terms *reorder* and *no reorder* specify how the replacement status of the lines were updated. *Reorder* means that a modified line is

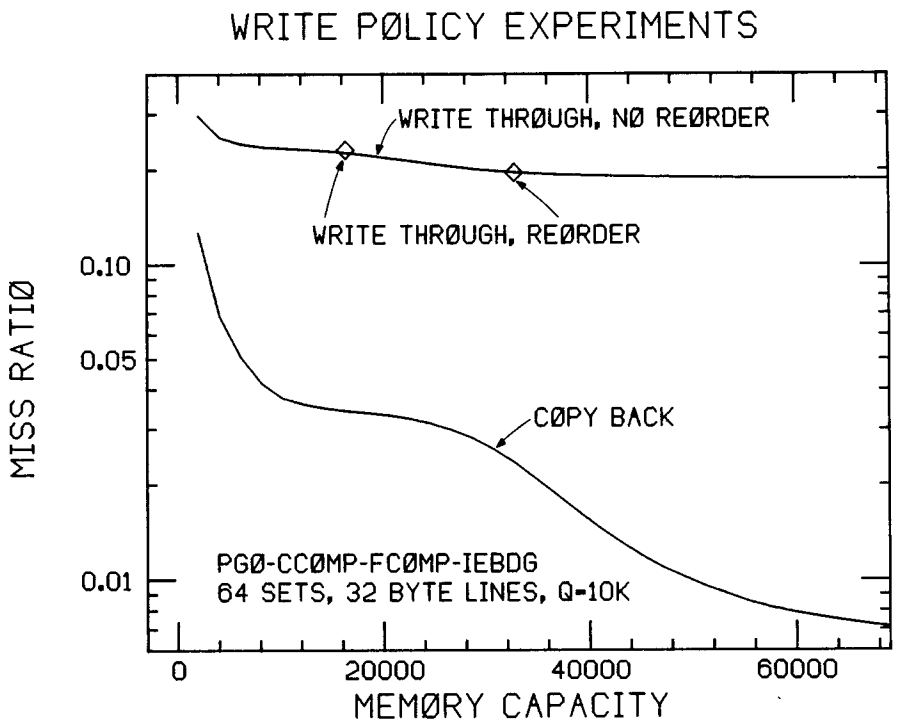

Figure 22. Miss ratio for copy-back and write-through. All writes counted as misses for write-through. *No reorder* implies replacement status not modified on write, *reorder* implies replacement status is updated on write.

moved to the top of the LRU stack within its set in the cache. *No reorder* implies that the replacement status of the line is not modified on write. From Figure 22, it can be seen that there is no significant difference in the two policies. For this reason, the IBM 3033, a write-through machine, does not update the LRU status of lines when a write occurs.

With respect to performance, there is no clear choice to be made between write-through and copy-back. This is because a good implementation of write-through seldom has to wait for a write to complete. A good implementation of write-through requires, however, both sufficient buffering of writes and sufficient interleaving of main memory so that the probability of the CPU becoming blocked while waiting on a store is small. This appears to be true in the IBM 3033, but at the expense of a great deal of buffering and other logic. For example, in the 3033 each buffered store requires a double-word datum buffer, a single buffer for the store address, and a 1-byte buffer to indicate which bytes have been modified in the double-word store. There are also comparators to match each store address against subsequent accesses to main memory, so that references to the modified data get the updated values. Copy-back could probably have been implemented much more cheaply.

The Amdahl Computers all use copy-back, as does the IBM 3081. IBM uses store-through in the 370/168 [IBM75] and the 3033 [IBM78], as does DEC in the PDP-11/70 [STRE76] and VAX 11/780 [DEC78], Honeywell in the 66/60 and 66/80, and Itel in the AS/6.

2.6 Effect of Multiprogramming: Cold-Start and Warm-Start

A significant factor in the cache miss ratio is the frequency of task switching, or inversely, the value of the mean intertask switch time, Q. The problem with task switching is that every time the active task is changed, a new process may have to be

loaded from scratch into the cache. This issue was discussed by EAST75, where the terms *warm-start* and *cold-start* were coined to refer to the miss ratio starting with a full memory and the miss ratio starting with an empty memory, respectively. Other papers which discuss the problem include EAST78, KOBA80, MACD79, PEUT77, POHM75, SCHR71, and STRE76.

Typically, a program executes for a period of time before an interruption (I/O, clock, etc.) of some type invokes the supervisor. The supervisor eventually relinguishes control of the processor to some user process, perhaps the same one as was running most recently. If it is not the same user process, the new process probably does not find any lines of its address space in the cache, and starts immediately with a number of misses. If the most recently executed process is restarted, and if the supervisor interruption has not been too long, some useful information may still remain. In PEUT77, some figures are given about the length of certain IBM operating system supervisor interruptions and what fraction of the cache is purged. (One may also view the user as interrupting the supervisor and increasing the supervisor miss ratio.)

The effect of the task-switch interval on the miss ratio cannot be easily estimated. In particular, the effect depends on the workload and on the cache size. We also observe that the proportion of cache misses due to task switching increases with increasing cache size, even though the absolute miss ratio declines. This is because a small cache has a large inherent miss ratio (since it does not hold the program's working set) and this miss ratio is only slightly augmented by task-switch-induced misses. Conversely, the inherent low miss ratio of a large cache is greatly increased, in relative terms, by task switching. We are not aware of any current machine for which a breakdown of the miss ratio into these two components has been done.

Some experimental results bearing on this problem appear in Figures 23 and 24. In each, the miss ratio is shown as a function of the memory size and task-switch interval Q (Q is the number of memory references). The figures presented can be understood as follows. A very small Q (e.g., 100, 1,000) implies that the cache is shared between all of the active processes, and that when a process is restarted it finds a significant fraction of its previous information still in the cache. A very large Q (e.g., 100,000, 250,000) implies that when the program is restarted it finds an empty cache (with respect to its own working set), but that the new task runs long enough first to fill the cache and then to take advantage of the full cache. Intermediate values for Q result in the situation where a process runs for a while but does not fill the cache; however, when it is restarted, none of its information is still cache resident (since the multiprogramming degree is four). These three regions of operation are evident in Figures 23 and 24 as a function of Q and of the cache size. (In SATY81, Q is estimated to be about 25,000.)

There appear to be several possible solutions to the problem of high cache miss ratios due to task switching. (1) It may be possible to lengthen the task-switch interval. (2) The cache can be made so large that several programs can maintain information in it simultaneously. (3) The scheduling algorithm may be modified in order to give preference to a task likely to have information resident in the cache. (4) If the working set of a process can be identified (e.g., from the previous period of execution), it might be reloaded as a whole; this is called *working set restoration*. (5) Multiple caches may be created; for example, a separate cache could be established for the supervisor to use, so that, when invoked, it would not displace user data from the cache. This idea is considered in Section 2.10, and some of the problems of the approach are indicated.

A related concept is the idea of bypassing the cache for operations unlikely to result in the reuse of data. For example, long vector operations such as a very long move character (e.g., IBM MVCL) could bypass the cache entirely [LOSQ82] and thereby avoid displacing other data more likely to be reused.

2.7 Multicache Consistency

Large modern computer systems often have several independent processors, consisting sometimes of several CPUs and sometimes of just a single CPU and several channels.

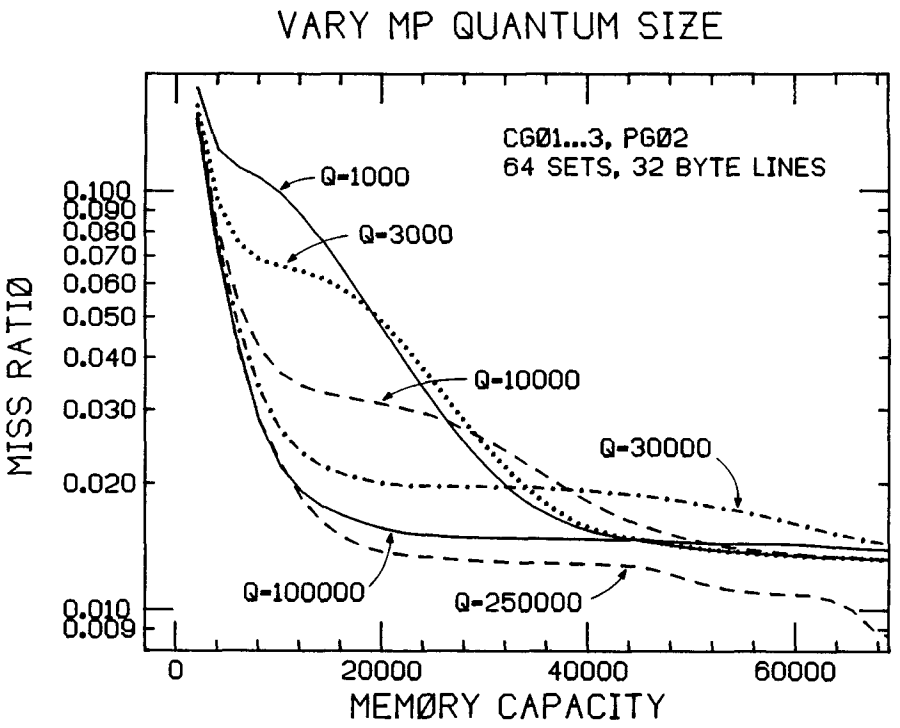

Figure 23. Miss ratio versus memory capacity for range of multiprogramming intervals Q.

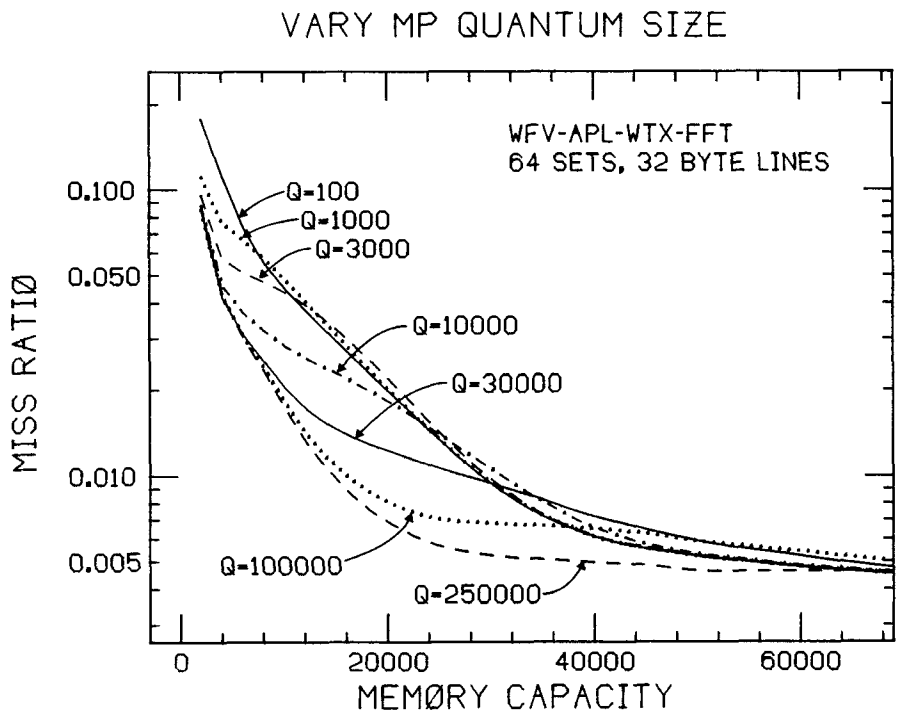

Figure 24. Miss ratio versus memory capacity for variety of multiprogramming intervals Q.

Each processor may have zero, one, or several caches. Unfortunately, in such a multiple processor system, a given piece of information may exist in several places at a given time, and it is important that all processors have access (as necessary) to the same, unique (at a given time) value. Several solutions exist and/or have been proposed for this problem. In this section, we discuss many of these solutions; the interested reader should refer to BEAN79, CENS78, DRIM81a, DUBO82, JONE76, JONE77a, MAZA77, MCWI77, NGAI81, and TANG76 for additional explanation.

As a basis for discussion, consider a single CPU with a cache and with a main memory behind the cache. The CPU reads an item into the cache and then modifies it. A second CPU (of similar design and using the same main memory) also reads the item and modifies it. Even if the CPUs were using store-through, the modification performed by the second CPU would not be reflected in the cache of the first CPU unless special steps were taken. There are several possible special steps.

1. Shared Cache. All processors in the system can use the same cache. In general, this solution is infeasible because the bandwidth of a single cache usually is not sufficient to support more than one CPU, and because additional access time delays may be incurred because the cache may not be physically close enough to both (or all) processors. This solution is employed successfully in the Amdahl 470 computers, where the CPU and the channels all use the same cache; the 470 series does not, however, permit tightly coupled CPUs. The UNIVAC 1100/80 [BORG79] permits two CPUs to share one cache.

2. Broadcast Writes. Every time a CPU performs a write to the cache, it also sends that write to all other caches in the system. If the target of the store is found in some other cache, it may be either updated or invalidated. Invalidation may be less likely to create inconsistencies, since updates can possibly "cross," such that CPU A updates its own cache and then B's cache. CPU B simultaneously updates its own cache and then A's. Updates also require more data transfer. The IBM 370/168 and 3033 processors use invalidation. A store by a CPU or channel is broadcast to all caches sharing the same main memory. This broadcast store is placed in the *buffer invalidation address stack* (BIAS) which is a list of addresses to be invalidated in the cache. The buffer invalidation address stack has a high priority for cache cycles, and if the target line is found in that cache, it is invalidated.

The major difficulty with broadcasting store addresses is that every cache memory in the system is forced to surrender a cycle for invalidation lookup to any processor which performs a write. The memory interference that occurs is generally acceptable for two processors (e.g., IBM's current MP systems), but significant performance degradation is likely with more than two processors. A clever way to minimize this problem appears in a recent patent [BEAN79]. In that patent, a BIAS Filter Memory (BFM) is proposed. A BFM is associated with each cache in a tightly coupled MP system. This filter memory works by filtering out repeated requests to invalidate the same block in a cache.

3. Software Control. If a system is being written from scratch and the architecture can be designed to support it, then software control can be used to guarantee consistency. Specifically, certain information can be designated noncacheable, and can be accessed only from main memory. Such items are usually semaphores and perhaps data structures such as the job queue. For efficiency, some shared writeable data has to be cached. The CPU must therefore be equipped with commands that permit it to purge any such information from its own cache as necessary. Access to shared writeable cacheable data is possible only within critical sections, protected by noncacheable semaphores. Within the critical regions, the code is responsible for restoring all modified items to main memory before releasing the lock. Just such a scheme is intended for the S-1 multiprocessor system under construction at the Lawrence Livermore Laboratory [HAIL79, MCWI77]. The Honeywell Series 66 machines use a similar mechanism. In some cases, the simpler alternative of making shared writeable information noncacheable may be acceptable.

4. Directory Methods. It is possible to keep a centralized and/or distributed direc-

tory of all main memory lines, and use it to ensure that no lines are write-shared. One such scheme is as follows, though several variants are possible. The main memory maintains $k + 1$ bits for each line in main memory, when there are k caches in the system. Bit i, $i = 1, \ldots, k$ is set to 1 if the corresponding cache contains the line. The $(k + 1)$th bit is 1 if the line is being or has been modified in a cache, and otherwise is 0. If the $(k + 1)$th bit is on, then exactly one of the other bits is on. Each CPU has associated with each line in its cache a single bit (called the *private* bit). If that bit is on, that CPU has the only valid copy of that line. If the bit is off, other caches and main memory may also contain current copies. Exactly one private bit is set if and only if the main memory directory bit $k + 1$ is set.

A CPU can do several things which provoke activity in this directory system. If a CPU attempts to read a line which is not in its cache, the main memory directory is queried. There are two possibilities: either but $k + 1$ is off, in which case the line is transferred to the requesting cache and the corresponding bit set to indicate this; or, bit $k + 1$ is on, in which case the main memory directory must recall the line from the cache which contains the modified copy, update main memory, invalidate the copy in the cache that modified it, send the line to the requesting CPU/cache and finally update itself to reflect these changes. (Bit $k + 1$ is then set to zero, since the request was a read.)

An attempt to perform a write causes one of three possible actions. If the line is already in cache and has already been modified, the private bit is on and the write takes place immediately. If the line is not in cache, then the main memory directory must be queried. If the line is in any other cache, it must be invalidated (in all other caches), and main memory must be updated if necessary. The main memory directory is then set to reflect the fact that the new cache contains the modified copy of the data, the line is transmitted to the requesting cache, and the private bit is set. The third possibility is that the cache already contains the line but that it does not have its private bit set. In this case, permission must be requested from the main memory directory to perform the write. The main memory directory invalidates any other copies of the line in the system, marks its own directory suitably, and then gives permission to modify the data.

The performance implications of this method are as follows. The cost of a miss may increase significantly due to the need to query the main memory directory and possibly retrieve data from other caches. The use of shared writeable information becomes expensive due to the high miss ratios that are likely to be associated with such information. In CENS78, there is some attempt at a quantitative analysis of these performance problems.

Another problem is that I/O overruns may occur. Specifically, an I/O data stream may be delayed while directory operations take place. In the meantime, some I/O data are lost. Care must be taken to avoid this problem. Either substantial I/O buffering or write-through is clearly needed.

Other variants of method 4 are possible. (1) The purpose of the central directory is to minimize the queries to the caches of other CPUs. The central directory is not logically necessary; sufficient information exists in the individual caches. It is also possible to transmit information from cache to cache, without going through main memory. (2) If the number of main memory directory bits is felt to be too high, locking could be on a page instead of on a line basis. (3) Store-through may be used instead of copy-back; thus main memory always has a valid copy and data do not have to be fetched from the other caches, but can simply be invalidated in these other caches.

The IBM 3081D, which contains two CPUs, essentially uses the directory scheme described. The higher performance 3081K functions similarly, but passes the necessary information from cache to cache rather than going through main memory.

Another version of the directory method is called the broadcast search [DRIM81b]. In this case, a miss is sent not only to the main memory but to all caches. Whichever memories (cache or main) contain the desired information send it to the requesting processor.

Liu [LIU82] proposes a multicache scheme to minimize the overhead of directory operations. He suggests that all CPUs

have two caches, only one of which can contain shared data. The overhead of directory access and maintenance would thus only be incurred when the shared data cache is accessed.

There are two practical methods among the above alternatives: method 4 and the BIAS Filter Memory version of method 2. Method 4 is quite general, but is potentially very complex. It may also have performance problems. No detailed comparison exists, and other and better designs may yet remain to be discovered. For new machines, it is not known whether software control is better than hardware control; clearly, for existing architectures and software, hardware control is required.

2.8 Data/Instruction Cache

Two aspects of the cache having to do with its performance are cache bandwidth and access time. Both of these can be improved by splitting the cache into two parts, one for data and one for instructions. This doubles the bandwidth since the cache can now service two requests in the time it formerly required for one. In addition, the two requests served are generally complementary. Fast computers are pipelined, which means that several instructions are simultaneously in the process of being decoded and executed. Typically, there are several stages in a pipeline, including instruction fetch, instruction decode, operand address generation, operand fetch, execution, and transmission of the results to their destination (e.g., to a register). Therefore, while one instruction is being fetched (from the instruction cache), another can be having its operands fetched from the operand cache. In addition, the logic that arbitrates priority between instruction and data accesses to the cache can be simplified or eliminated.

A split instruction/data cache also provides access time advantages. The CPU of a high-speed computer typically contains (exclusive of the S-unit) more than 100,000 logic gates and is physically large. Further, the logic having to do with instruction fetch and decode has little to do with operand fetch and store except for *execute* instructions and possibly for the targets of branches. With a single cache system, it is not always possible simultaneously to place the cache immediately adjacent to all of the logic which will access it. A split cache, on the other hand, can have each of its halves placed in the physical location which is most useful, thereby saving from a fraction of a nanosecond to several nanoseconds.

There are, of course, some problems introduced by the split cache organization. First, there is the problem of consistency. Two copies now exist of information which formerly existed only in one place. Specifically, instructions can be modified, and this modification must be reflected before the instructions are executed. Further, it is possible that even if the programs are not self-modifying, both data and instructions may coexist in the same line, either for reasons of storage efficiency or because of immediate operands. The solutions for this problem are the same as those discussed in the section on multicache consistency (Section 2.7), and they work here as well. It is imperative that they be implemented in such a way so as to not impair the access time advantage given by this organization.

Another problem of this cache organization is that it results in inefficient use of the cache memory. The size of the working set of a program varies constantly, and in particular, the fraction devoted to data and to instructions also varies. (A dynamically split design is suggested by FAVR78.) If the instructions and data are not stored together, they must each exist within their own memory, and be unable to share a larger amount of that resource. The extent of this problem has been studied both experimentally and analytically. In SHED76, a set of formulas are provided which can be used to estimate the performance of the unified cache from the performance of the individual ones. The experimental results were not found to agree with the mathematical ones, although the reason was not investigated. We believe that the nonstationarity of the workload was the major problem.

We compared the miss ratio of the split cache to that of the unified cache for each of the sets of traces; some of the results appear in Figures 25-28. (See BELL74 and THAK78 for additional results.) We note that there are several possible ways to split and manage the cache and the various alternatives have been explored. One could split the cache in two equal parts (labeled "SPLIT EQUAL"), or the observed miss

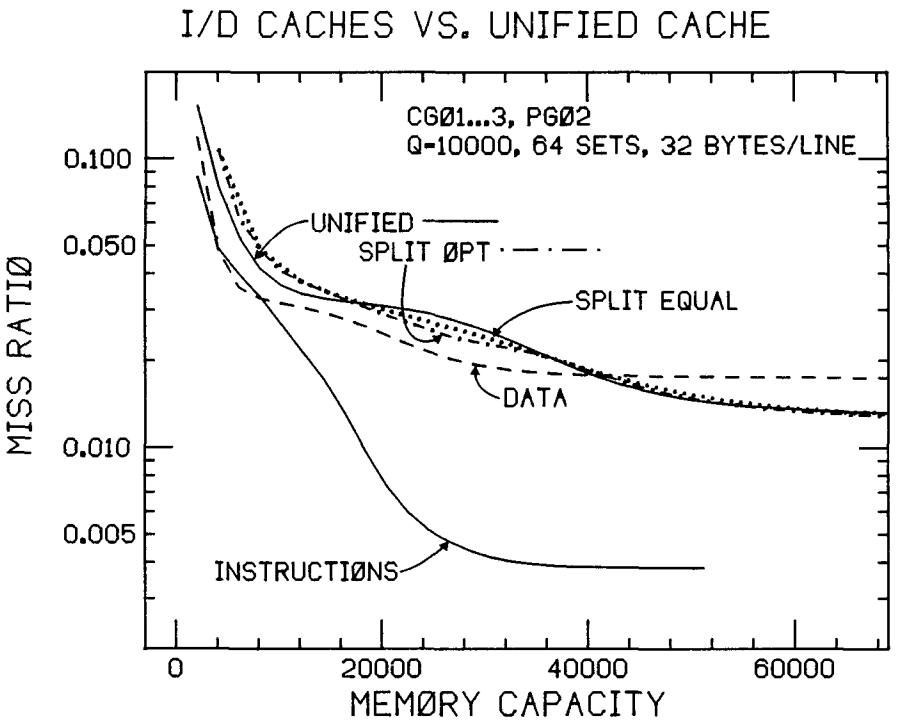

Figure 25. Miss ratio versus memory capacity for unified cache, cache split equally between instruction and data halves, cache split according to static optimum partition between instruction and data halves, and miss ratios individually for instruction and data halves.

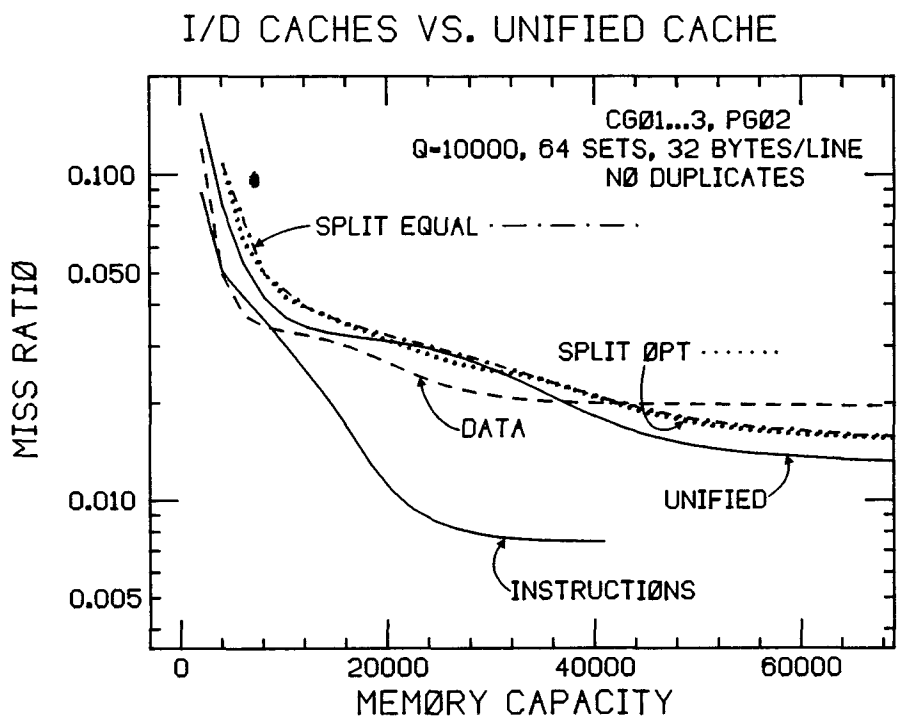

Figure 26. Miss ratio versus memory capacity for instruction/data cache and for unified cache.

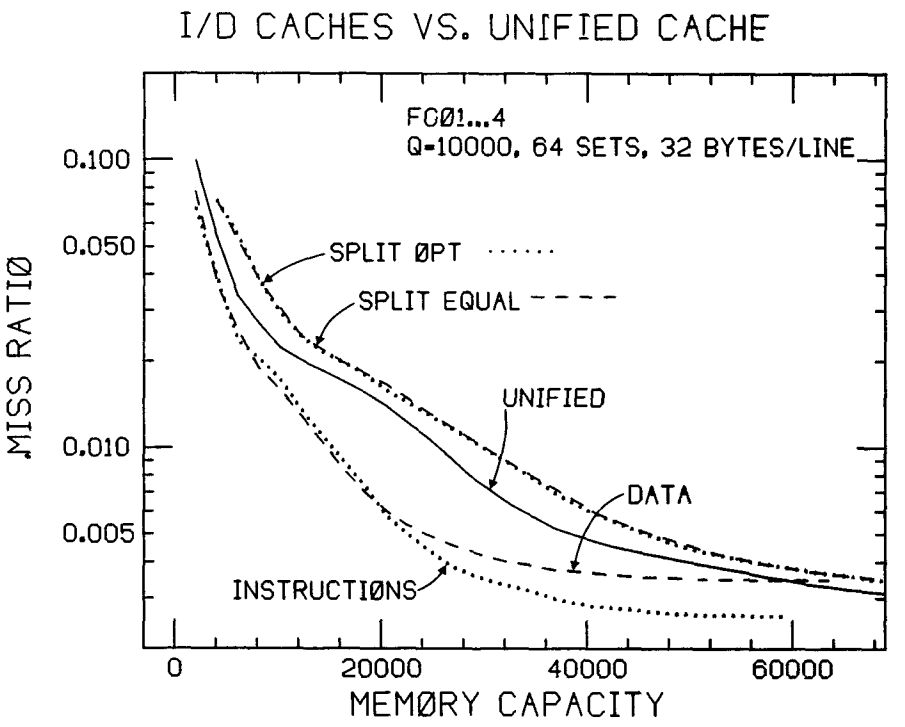

Figure 27. Miss ratio versus memory capacity for instruction/data cache and for unified cache.

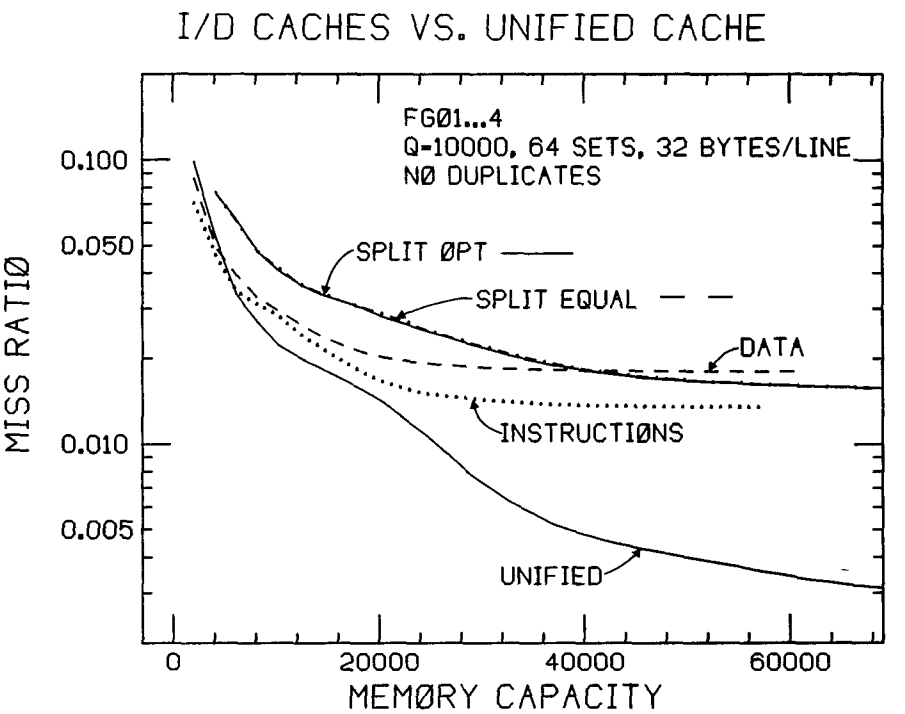

Figure 28. Miss ratio versus memory capacity for instruction/data cache and for unified cache

ratios could be used (from this particular run) to determine the optimal static split ("SPLIT OPT"). Also, when a line is found to be in the side of the cache other than the one currently accessed, it could either be duplicated in the remaining side of the cache or it could be moved; this latter case is labeled "NO DUPLICATES." (No special label is shown if duplicates are permitted.) The results bearing on these distinctions are given in Figures 25–28.

We observe in Figures 25 and 26 that the unified cache, the equally split cache, and the cache split unequally (split optimally) all perform about equally well with respect to miss ratio. Note that the miss ratio for the instruction and data halves of the cache individually are also shown. Further, comparing Figures 25 and 26, it seems that barring duplicate lines has only a small negative effect on the miss ratio.

In sharp contrast to the measurements of Figures 25 and 26 are those of Figures 27 and 28. In Figure 27, although the equally and optimally split cache are comparable, the unified cache is significantly better. The unified cache is better by an order of magnitude when duplicates are not permitted (Figure 28), because the miss ratio is sharply increased by the constant movement of lines between the two halves. It appears that lines sharing instruction and data are very common in programs compiled with IBM's FORTRAN G compiler and are not common in programs compiled using the IBM COBOL or PL/I compiler. (Results similar to the FORTRAN case have been found for the two other sets of IBM traces, all of which include FORTRAN code but are not shown.)

Based on these experimental results, we can say that the miss ratio may increase significantly if the caches are split, but that the effect depends greatly on the workload. Presumably, compilers can be designed to minimize this effect by ensuring that data and instructions are in separate lines, and perhaps even in separate pages.

Despite the possible miss ratio penalty of splitting the cache, there are at least two experimental machines and two commercial ones which do so. The S-1 [HAIL79, MCWI77] at Lawrence Livermore Laboratory is being built with just such a cache; it relies on (new) software to minimize the problems discussed here. The 801 minicomputer, built at IBM Research (Yorktown Heights) [ELEC76, RADI82] also has a split cache. The Hitachi H200 and Itel AS/6 [ROSS79] both have a split data/instruction cache. No measurements have been publicly reported for any of these machines.

2.9 Virtual Address Cache

Most cache memories address the cache using the real address (see Figure 2). As the reader recalls, we discussed (Introduction, Section 2.3) the fact that the virtual address was translated by the TLB to the real address, and that the line lookup and readout could not be completed until the real address was available. This suggests that the cache access time could be significantly reduced if the translation step could be eliminated. The way to do this is to address the cache directly with the virtual address. We call a cache organized this way a virtual address cache. The MU-5 [IBBE77] uses this organization for its name store. The S-1, the IBM 801, and the ICL 2900 series machines also use this idea. It is discussed in BEDE79. See also OLBE79.

There are some additional considerations in building a virtual address cache, and there is one serious problem. First, all addresses must be tagged with the identifier of the address space with which they are associated, or else the cache must be purged every time task switching occurs. Tagging is not a problem, but the address tag in the cache must be extended to include the address space ID. Second, the translation mechanism must still exist and must still be efficient, since virtual addresses must be translated to real addresses whenever main memory is accessed, specifically for misses and for writes in a write-through cache. Thus the TLB cannot be eliminated.

The most serious problem is that of "synonyms," two or more virtual addresses that map to the same real address. Synonyms occur whenever two address spaces share code or data. (Since the lines have address space tags, the virtual addresses are different even if the line occurs in the same place in both address spaces.) Also, the supervisor may exist in the address space of each user, and it is important that

only one copy of supervisor tables be kept. The only way to detect synonyms is to take the virtual address, map it into a real address, and then see if any other virtual addresses in the cache map into the same real address. For this to be feasible, the inverse mapping must be available for every line in the cache; this inverse mapping is accessed on real address and indicates all virtual addresses associated with that real address. Since this inverse mapping is the opposite of the TLB, we choose to call the inverse mapping buffer (if a separate one is used) the RTB or reverse translation buffer. When a miss occurs, the virtual address is translated into the real address by the TLB. The access to main memory for the miss is overlapped with a similar search of the RTB to see if that line is already in the cache under a different name (virtual address). If it is, it must be renamed and moved to its new location, since multiple copies of the same line in the cache are clearly undesirable for reasons of consistency.

The severity of the synonym problem can be decreased if shared information can be forced to have the same location in all address spaces. Such information can be given a unique address space identifier, and the lookup algorithm always considers such a tag to match the current address space ID. A scheme like this is feasible only for the supervisor since other shared code could not conceivably be so allocated. Shared supervisor code does have a unique location in all address spaces in IBM's MVS operating system.

The RTB may or may not be a simple structure, depending on the structure of the rest of the cache. In one case it is fairly simple: if the bits used to select the set of the cache are the same for the real and virtual address (i.e., if none of the bits used to select the set undergo translation), the RTB can be implemented by associating with each cache line two address tags [BEDE79]. If a match is not found on the virtual address, then a search is made in that set on the real address. If that search finds the line, then the virtual address tag is changed to the current virtual address. A more complex design would involve a separate mapping buffer for the reverse translation.

2.10 User/Supervisor Cache

It was suggested earlier that a significant fraction of the miss ratio is due to task-switching. A possible solution to this problem is to use a cache which has been split into two parts, one of which is used only by the supervisor and the other of which is used primarily by the user state programs. If the scheduler were programmed to restart, when possible, the same user program that was running before an interrupt, then the user state miss ratio would drop appreciably. Further, if the same interrupts recur frequently, the supervisor state miss ratio may also drop. In particular, neither the user nor the supervisor would purge the cache of the other's lines. (See PEUT77 for some data relevant to this problem.) The supervisor cache may have a high miss ratio in any case due to its large working set. (See MILA75 for an example.)

Despite the appealing rationale of the above comments, there are a number of problems with the user/supervisor cache. First, it is actually unlikely to cut down the miss ratio. Most misses occur in supervisor state [MILA75] and a supervisor cache half as large as the unified cache is likely to be worse since the maximum cache capacity is no longer available to the supervisor. Further, it is not clear what fraction of the time the scheduling algorithm can restart the same program. Second, the information used by the user and the supervisor are not entirely distinct, and cross-access must be permitted. This overlap introduces the problem of consistency.

We are aware of only one evaluation of the split user/supervisor cache [ROSS80]. In that case, an experiment was run on an Hitatchi M180. The results seemed to show that the split cache performed about as well as a unified one, but poor experimental design makes the results questionable. We do not expect that a split cache will prove to be useful.

2.11 Input/Output through the Cache

In Section 2.7, the problem of multicache consistency was discussed. We noted that if all accesses to main memory use the same cache, then there would be no consistency problem. Precisely this approach has been

used in one manufacturer's computers (Amdahl Corporation).

2.11.1 Overruns

While putting all input/output through the cache solves the consistency problem, it introduces other difficulties. First, there is the overrun problem. An overrun occurs when for some reason the I/O stream cannot be properly transmitted between the memory (cache or main) and the I/O device. Transmitting the I/O through the cache can cause an overrun when the line accessed by the I/O stream is not in the cache (and is thus a miss) and for some reason cannot be obtained quickly enough. Most I/O devices involve physical movement, and when the buffering capacity embedded in the I/O path is exhausted, the transfer must be aborted and then restarted. Overruns can be provoked when: (1) the cache is already processing one or more misses and cannot process the current (additional) one quickly enough; (2) more than one I/O transfer is in progress, and more active (in use) lines map into one set than the set size can accommodate; or (3) the cache bandwidth is not adequate to handle the current burst of simultaneous I/O from several devices. Overruns can be minimized if the set size of the cache is large enough, the bandwidth is high enough, and the ability to process misses is sufficient. Sufficient buffering should also be provided in the I/O paths to the devices.

2.11.2 Miss Ratio

Directing the input/output data streams through the cache also has an effect on the miss ratio. This I/O data occupies some fraction of the space in the cache, and this increases the miss ratio for the other users of the cache. Some experiments along these lines were run by the author and results are shown in Figures 29–32. IORATE is the ratio of the rate of I/O accesses to the cache to the rate of CPU accesses. (I/O activity is simulated by a purely sequential synthetic address stream referencing a distinct address space from the other programs.) The miss ratio as a function of memory size and I/O transfer rate is shown in Figures 29 and 30 for two of the sets of traces. The data has been rearranged to show more directly the effect on the miss ratio in Figures 31 and 32. The results displayed here show no clear mathematical pattern, and we were unable to derive a useful and verifiable formula to predict the effect on the miss ratio by an I/O stream.

Examination of the results presented in Figures 29–32 suggests that for reasonable I/O rates (less than 0.05; see POWE77 for some I/O rate data) the miss ratio is not affected to any large extent. This observation is consistent with the known performance of the Amdahl computers, which are not seriously degraded by high I/O rates.

2.12 Cache Size

Two very important questions when selecting a cache design are how large should the cache be and what kind of performance can we expect. The cache size is usually dictated by a number of criteria having to do with the cost and performance of the machine. The cache should not be so large that it represents an expense out of proportion to the added performance, nor should it occupy an unreasonable fraction of the physical space within the processor. A very large cache may also require more access circuitry, which may increase access time.

Aside from the warnings given in the paragraph above, one can generally assume that the larger the cache, the higher the hit ratio, and therefore the better the performance. The issue is then one of the relation between cache size and hit ratio. This is a very difficult problem, since the cache hit ratio varies with the workload and the machine architecture. A cache that might yield a 99.8 percent hit ratio on a PDP-11 program could result in a 90 percent or lower hit ratio for IBM (MVS) supervisor state code. This problem cannot be usefully studied using trace-driven simulation because the miss ratio varies tremendously from program to program and only a small number of traces can possibly be analyzed. Typical trace-driven simulation results appear throughout this paper, however, and the reader may wish to scan that data for insight. There is also a variety of data available in the literature and the reader may wish to inspect the results presented in ALSA78, BELL74, BERG76, GIBS67, LEE69,

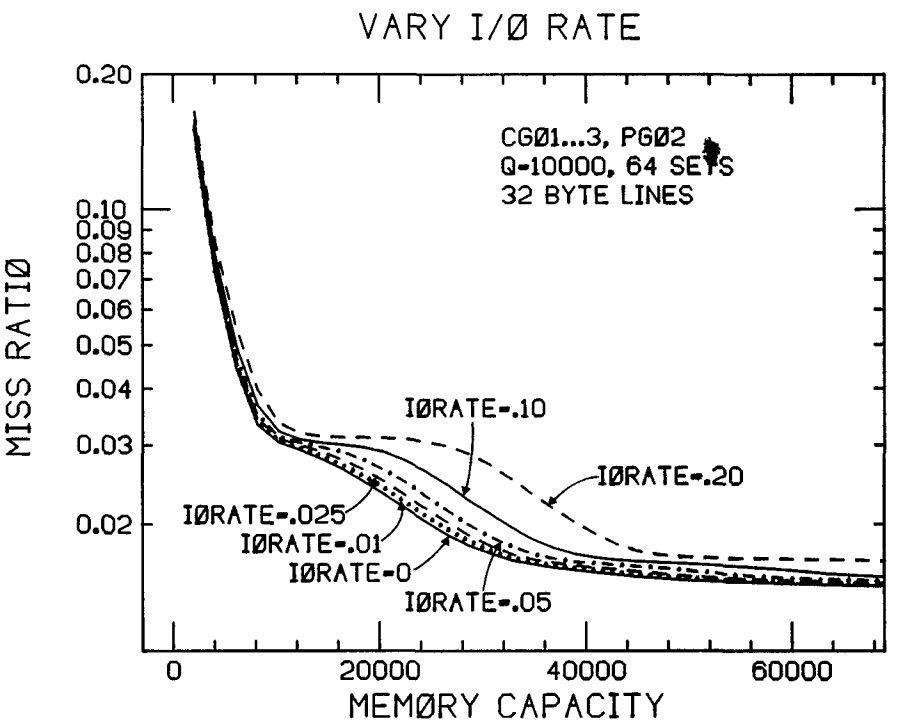

Figure 29. Miss ratio versus memory capacity while I/O occurs through cache at specified rate IORATE refers to fraction of all memory references due to I/O

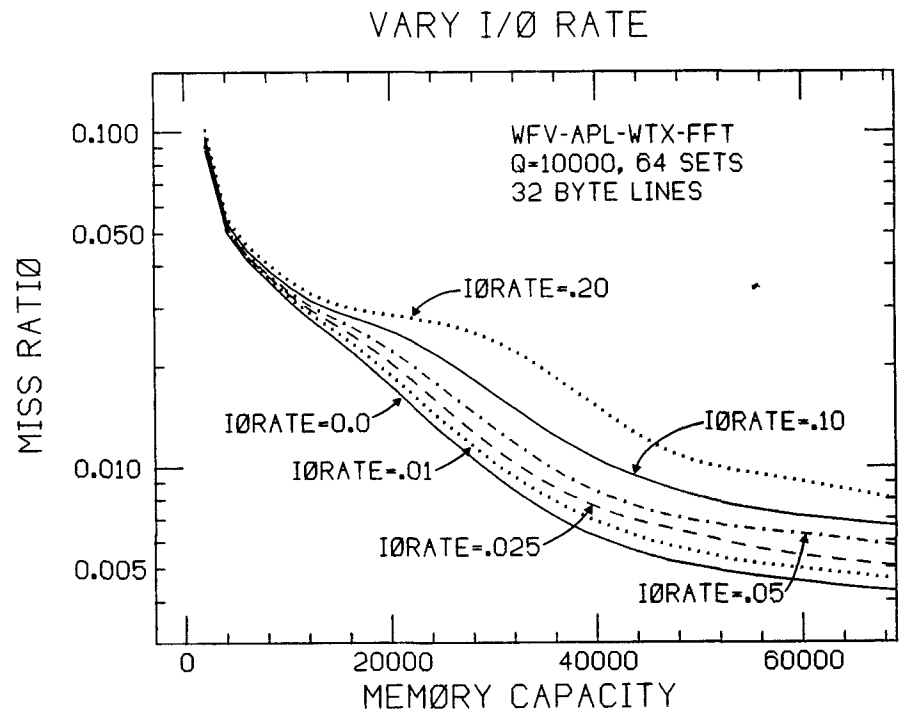

Figure 30. Miss ratio versus memory capacity while I/O occurs through cache.

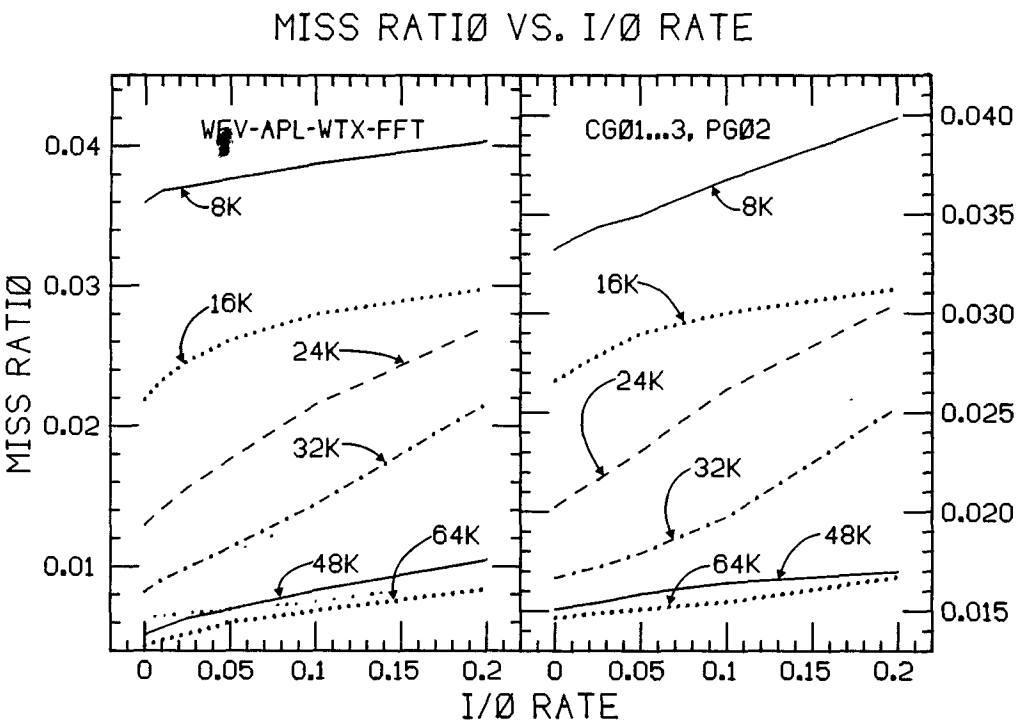

Figure 31. Miss ratio versus I/O rate for variety of memory capacities.

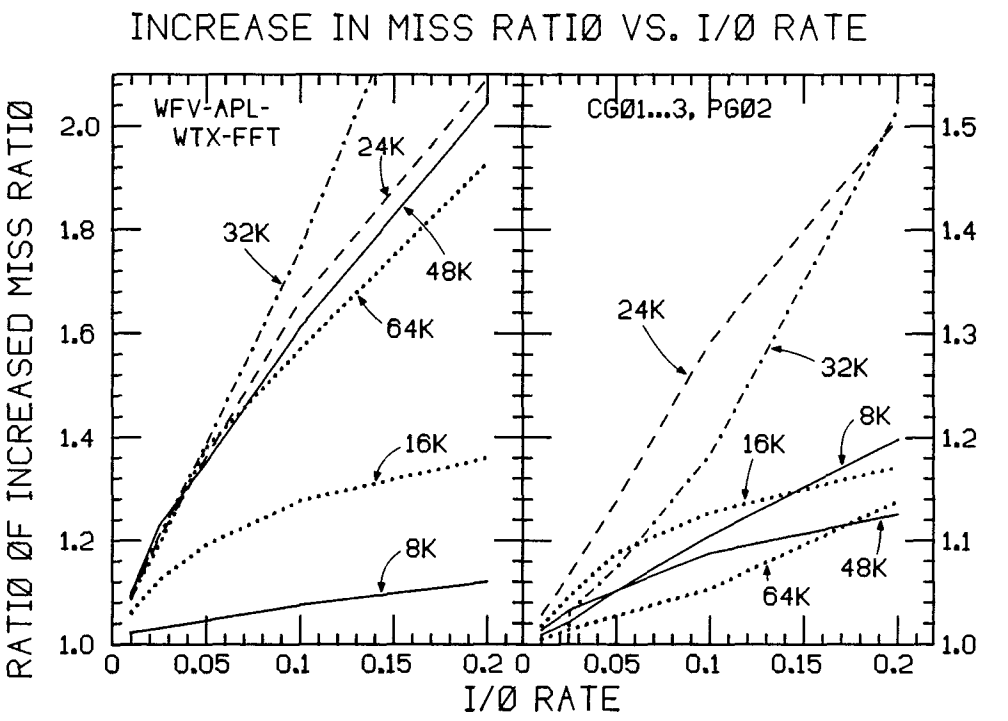

Figure 32. Miss ratio versus I/O rate for variety of memory capacities.

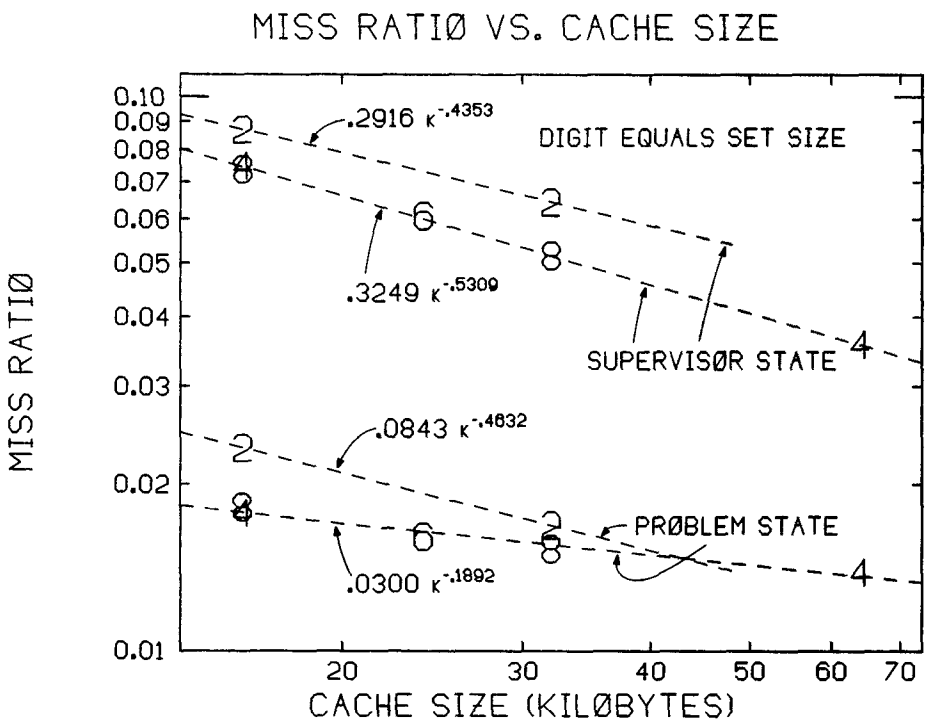

Figure 33. Miss ratio versus cache size, classified by set size and user/supervisor state. Data gathered by hardware monitor from machine running standard benchmark.

MEAD70, STRE76, THAK78, and YUVA75. Some direct measurements of cache miss ratios appear in CLAR81 and CLAR82. In the former, the Dorado was found to have hit ratios above 99 percent. In the latter, the VAX 11/780 was found to have hit ratios around 90 percent.

Another problem with trace-driven simulation is that in general user state programs are the only ones for which many traces exist. In IBM MVS systems, the supervisor typically uses 25 to 60 percent of the CPU time, and provides by far the largest component of the miss ratio [MILA75]. User programs generally have very low miss ratios, in our experience, and many of those misses come from task switching.

Two models have been proposed in the literature for memory hierarchy miss ratios. Saltzer [SALT74] suggested, based on his data, that the mean time between faults was linearly related to the capacity of the memory considered. But later results, taken on the same system [GREE74] contradict Saltzer's earlier findings. [CHOW75 and CHOW76] suggest that the miss ratio curve was of the form $m = a(c^b)$, where a and b are constants, m is the miss ratio, and c is the memory capacity. They show no experimental results to substantiate this model, and it seems to have been chosen for mathematical convenience.

Actual cache miss ratios, from real machines running "typical" workloads, are the most useful source of good measurement data. In Figure 33 we show a set of such measurements taken from Amdahl 470 computers running a standard Amdahl internal benchmark. This data is reproduced from HARD80a. Each digit represents a measurement point, and shows either the supervisor or problem state miss ratio for a specific cache size and set size; the value of the digit at each point is the set size. Examination of the form of the measurements from Figure 33 suggest that the miss ratio can be approximated over the range shown by an equation of the form $m = a(k^b)$ (consistent with CHOW75 and CHOW76),

where m is the miss ratio, a and b are constants ($b < 0$), and k is the cache capacity in kilobytes. The values of a and b are shown for four cases in Figure 33; supervisor and user state for a set size of two, and supervisor and user state for all other set sizes. We make no claims for the validity of this function for other workloads and/or other architectures, nor for cache sizes beyond the range shown. From Figure 33 it is evident, though, that the supervisor contributes the largest fraction of the miss ratio, and the supervisor state measurements are quite consistent. Within the range indicated, therefore, these figures can probably be used for a first approximation at estimating the performance of a cache design.

Typical cache sizes in use include 128 kbytes (NEC ACOS 9000), 64 kbytes (Amdahl 470V/8, IBM 3033, IBM 3081K per CPU), 32 kbytes (IBM 370/168-3, IBM 3081D per CPU, Amdahl 470V/7, Magnuson M80/43), 16 kbytes (Amdahl 470V/6, Itel AS/6, IBM 4341, Magnuson M80/42, M80/44, DEC VAX 11/780), 8 kbytes (Honeywell 66/60 and 66/80, Xerox Dorado [CLAR81]), 4 kbytes (VAX 11/750, IBM 4331), 1 kbyte (PDP-11/70).

2.13. Cache Bandwidth, Data Path Width, and Access Resolution

2.13.1 Bandwidth

For adequate performance, the cache bandwidth must be sufficient. Bandwidth refers to the aggregate data transfer rate, and is equal to data path width divided by access time. The bandwidth is important as well as the access time, because (1) there may be several sources of requests for cache access (instruction fetch, operand fetch and store, channel activity, etc.), and (2) some requests may be for a large number of bytes. If there are other sources of requests for cache cycles, such as prefetch lookups and transfers, it must be possible to accommodate these as well.

In determining what constitutes an adequate data transfer rate, it is not sufficient that the cache bandwidth exceed the average demand placed on it by a small amount. It is important as well to avoid contention since if the cache is busy for a cycle and one or more requests are blocked, these blocked requests can result in permanently wasted machine cycles. In the Amdahl 470V/8 and the IBM 3033, the cache bandwidth appears to exceed the average data rate by a factor of two to three, which is probably the minimum sufficient margin. We note that in the 470V/6 (when prefetch is used experimentally) prefetches are executed only during otherwise idle cycles, and it has been observed that not all of the prefetches actually are performed. (Newly arrived prefetch requests take the place of previously queued but never performed requests.)

For some instructions, the cache bandwidth can be extremely important. This is particularly the case for data movement instructions such as: (1) instructions which load or unload all of the registers (e.g., IBM 370 instructions STM, LM); (2) instructions which move long character strings (MVC, MVCL); and (3) instructions which operate on long character strings (e.g., CLC, OC, NC, XC). In these cases, especially the first two, there is little if any processing to be done; the question is simply one of physical data movement, and it is important that the cache data path be as wide as possible—in large machines, 8 bytes (3033, 3081, Itel AS/6) instead of 4 (470V/6, V/7, V/8); in small machines, 4 bytes (VAX 11/780) instead of 2 (PDP-11/70).

It is important to note that cache data path width is expensive. Doubling the path width means doubling the number of lines into and out of the cache (i.e., the bus widths) and all of the associated circuitry. This frequently implies some small increase in access time, due to larger physical packaging and/or additional levels of gate delay. Therefore, both the cost and performance aspects of cache bandwidth must be considered during the design process.

Another approach to increasing cache bandwidth is to interleave the cache [DRIS80, YAMO80]. If the cache is required to serve a large number of small requests very quickly, it may be efficient to replicate the cache (e.g., two or four times) and access each separately, depending on the low-order bits of the desired locations. This approach is very expensive, and to our knowledge, has not been used on any existing machine. (See POHM75 for some additional comments.)

2.13.2 Priority Arbitration

An issue related to cache bandwidth is what to do when the cache has several requests competing for cache cycles and only one can be served at a given time. There are two criteria for making the choice: (1) give priority to any request that is "deadline scheduled" (e.g., an I/O request that would otherwise abort); and (2) give priority (after (1)) to requests in order to enhance machine performance. The second criterion may be sacrificed for implementation convenience, since the optimal scheduling of cache accesses may introduce unreasonable complexity into the cache design. Typically, fixed priorities are assigned to competing cache requests, but dynamic scheduling, though complex, is possible [BLOU80].

An an illustration of cache priority resolution, we consider two large, high-speed computers: the Amdahl 470V/7 and the IBM 3033. In the Amdahl machine, there are five "ports" or address registers, which hold the addresses for cache requests. Thus, there can be up to five requests queued for access. These ports are the operand port, the instruction port, the channel port, the translate port, and the prefetch port. The first three are used respectively for operand store and fetch, instruction fetch, and channel I/O (since channels use the cache also). The translate port is used in conjunction with the TLB and translator to perform virtual to real address translation. The prefetch port is for a number of special functions, such as setting the storage key or purging the TLB, and for prefetch operations. There are sixteen priorities for the 470V/6 cache; we list the important ones here in decreasing order of access priority: (1) move line in from main storage, (2) operand store, (3) channel store, (4) fetch second half of double word request, (5) move line out from cache to main memory, (6) translate, (7) channel fetch, (8) operand fetch, (9) instruction fetch, (10) prefetch.

The IBM 3033 has a similar list of cache access priorities [IBM78]: (1) main memory fetch transfer, (2) invalidate line in cache modified by channel or other CPU, (3) search for line modified by channel or other CPU, (4) buffer reset, (5) translate, (6) redo (some cache accesses are blocked and have to be restarted), and (7) normal instruction or operand access.

2.14 Multilevel Cache

The largest existing caches (to our knowledge) can be found in the NEC ACOS 9000 (128 kbytes), and the Amdahl 470V/8 and IBM 3033 processors (64 kbytes). Such large caches pose two problems: (1) their physical size and logical complexity increase the access time, and (2) they are very expensive. The cost of the chips in the cache can be a significant fraction (5–20 percent) of the parts cost of the CPU. The reason for the large cache, though, is to decrease the miss ratio. A possible solution to this problem is to build a two-level cache, in which the smaller, faster level is on the order of 4 kbytes and the larger, slower level is on the order of 64–512 kbytes. In this way, misses from the small cache could be satisfied, not in the six to twelve machine cycles commonly required, but in two to four cycles. Although the miss ratio from the small cache would be fairly high, the improved cycle time and decreased miss penalty would yield an overall improvement in performance. Suggestions to this effect may be found in BENN82, OHNO77, and SPAR78. It has also been suggested for the TLB [NGAI82].

As might be expected, the two-level or multilevel cache is not necessarily desirable. We suggested above that misses from the fast cache to the slow cache could be serviced quickly, but detailed engineering studies are required to determine if this is possible. The five-to-one or ten-to-one ratio of main memory to cache memory access times is not wide enough to allow another level to be easily placed between them.

Expense is another consideration. A two-level cache implies another level of access circuitry, with all of the attendant complications. Also, the large amount of storage in the second level, while cheaper per bit than the low-level cache, is not inexpensive on the whole.

The two-level or multilevel cache represents a possible approach to the problem of an overlarge single-level cache, but further study is needed.

2.15 Pipelining

Referencing a cache memory is a multistep process. There is the need to obtain priority for a cache cycle. Then the TLB and the desired set are accessed in parallel. After this, the real address is used to select the correct line from the set, and finally, after the information is read out, the replacement bits are updated. In large, high-speed machines, it is common to pipeline the cache, as well as the rest of the CPU, so that more than one cache access can be in progress at the same time. This pipelining is of various degrees of sophistication, and we illustrate it by discussing two machines: the Amdahl 470V/7 and the IBM 3033.

In the 470V/7, a complete read requires four cycles, known as the P, B1, B2, and R cycles [SMIT78b]. The P (priority) cycle is used to determine which of several possible competing sources of requests to the cache will be permitted to use the next cycle. The B1 and B2 (buffer 1, buffer 2) cycles are used actually to access the cache and the TLB, to select the appropriate line from the cache, to check that the contents of the line are valid, and to shift to get the desired byte location out of the two-word (8-byte) segment fetched. The data are available at the end of the B2 cycle. The R cycle is used for "cleanup" and the replacement status is updated at that time. It is possible to have up to four fetches active at any one time in the cache, one in each of the four cycles mentioned above. The time required by a store is longer since it is essentially a read followed by a modify and write-back; it takes six cycles all together, and one store requires two successive cycles in the cache pipeline.

The pipeline in the 3033 cache is similar [IBM78]. The cache in the 3033 can service one fetch or store in each machine cycle, where the turnaround time from initial request for priority until the data is available is about $2\frac{1}{2}$ cycles ($\frac{1}{2}$-cycle transmission time to S-unit, $1\frac{1}{2}$ cycles in S-unit, $\frac{1}{2}$ cycle to return data to instruction unit). An important feature of the 3033 is that the cache accesses do not have to be performed in the order that they are issued. In particular, if an access causes a miss, it can be held up while the miss is serviced, and at the same time other requests which are behind it in the pipeline can proceed. There is an elaborate mechanism built in which prevents this out-of-order operation from producing incorrect results.

2.16 Translation Lookaside Buffer

The translation lookaside buffer (also called the translation buffer [DEC78], the associative memory [SCHR71], and the directory lookaside table [IBM78]), is a small, high-speed buffer which maintains the mapping between recently used virtual and real memory addresses (see Figure 2). The TLB performs an essential function since otherwise an address translation would require two additional memory references: one each to the segment and page tables. In most machines, the cache is accessed using real addresses, and so the design and implementation of the TLB is intimately related to the cache memory. Additional information relevant to TLB design and operation may be found in JONE77b, LUDL77, RAMA81, SATY81, SCHR71, and WILK71. Discussions of the use of TLBs (TLB chips or memory management units) in microcomputers can be found in JOHN81, STEV81, and ZOLN81.

The TLB itself is typically designed to look like a small set-associative memory. For example, the 3033 TLB (called the DLAT or directory lookaside table) is set-associative, with 64 sets of two elements each. Similarly, the Amdahl 470V/6 uses 128 sets of two elements each and the 470V/7 and V/8 have 256 sets of 2 elements each. The IBM 3081 TLB has 128 entries.

The TLB differs in some ways from the cache in its design. First, for most processes, address spaces start at zero and extend upward as far as necessary. Since the TLB translates page addresses from virtual to real, only the high-order (page number) address bits can be used to access the TLB. If the same method was used as that used for accessing the cache (bit selection using lower order bits), the low-order TLB entries would be used disproportionately and therefore the TLB would be used inefficiently. For this reason, both the 3033 and the 470 hash the address before accessing the TLB (see Figure 2). Consider the 24-bit address used in the System/370, with the bits numbered from 1 to 24 (high order to

low order). Then the bits 13 to 24 address the byte within the page (4096-byte page) and the remaining bits (1 to 12) can be used to access the TLB. The 3033 contains a 6-bit index into the TLB computed as follows. Let @ be the Exclusive OR operator; a 6-bit quantity is computed [7, (8 @ 2), (9 @ 3), (10 @ 4), (11 @ 5), (12 @ 6)], where each number refers to the input bit it designates.

The Amdahl 470V/7 and 470V/8 use a different hashing algorithm, one which provides much more thorough randomization at the cost of significantly greater complexity. To explain the hashing algorithm, we first explain some other items. The 470 associates with each address space an 8-bit tag field called the address space identifier (ASID) (see Section 2.19.1). We refer to the bits that make up this tag field as S1, . . . , S8. These bits are used in the hashing algorithm as shown below. Also, the 470V/7 uses a different algorithm to hash into each of the two elements of a set; the TLB is more like a pair of direct mapping buffers than a set-associative buffer. The first half is addressed using 8 bits calculated as follows: [(6 @ 1 @ S8), 7, (8 @ 3 @ S6), 9, (10 @ S4), 11, (12 @ S2), 5]; and the second half is addressed as [6, (7 @ 2 @ S7), 8, (9 @ 4 @ S5), 10, (11 @ S3), 12, (5 @ S1)]. There are no published studies that indicate whether the algorithm used in the 3033 is sufficient or whether the extra complexity of the 470V/7 algorithm is warranted.

There are a number of fields in a TLB entry (see Figure 34). The virtual address presented for translation is matched against the virtual address tag field (ASID plus virtual page address) in the TLB to ensure that the right entry has been found. The virtual address tag field must include the address space identifier (8 bits in the 470V/7, 5 bits in the 3033) so that entries for more than one process can be in the TLB at one time. A protection field (in 370-type machines) is also included in the TLB and is checked to make sure that the access is permissible. (Since keys are associated on a page basis in the 370, this is much more efficient than placing the key with each line in the cache.) The real address corresponding to the virtual address is the primary output of the TLB and occupies a

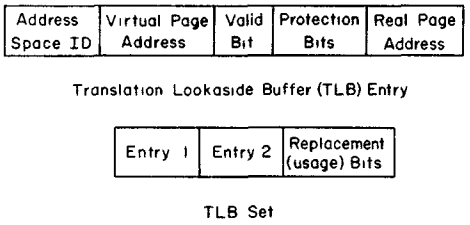

Figure 34. Structure of translation lookaside buffer (TLB) entry and TLB set.

field. There are also bits that indicate whether a given entry in the TLB is valid and the appropriate bits to permit LRU-like replacement. Sometimes, the modify and reference bits for a page are kept in the TLB. If so, then when the entry is removed from the TLB, the values of those bits must be stored.

It may be necessary to change one or more entries in the TLB whenever the virtual to real address correspondence changes for any page in the address space of any active process. This can be accomplished in two ways: (1) if a single-page table entry is changed (in the 370), the IPTE (insert page table entry) instruction causes the TLB to be searched, and the now invalid entry purged; (2) if the assignment of address space IDs is changed, then the entire TLB is purged. In the 3033, purging the TLB is slow (16 machine cycles) since each entry is actually invalidated. The 470V/7 does this in a rather clever way. There are two sets of bits used to denote valid and invalid entries, and a flag indicating which set is to be used at any given time. The set not in use is supposed to be set to zero (invalid). The purge TLB command has the effect of flipping this flag, so that the set of bits indicating that all entries are invalid are now in use. The set of bits no longer in use is reset to zero in the background during idle cycles. See Cosc81 for a similar idea.

The cache on the DEC VAX 11/780 [DEC78] is similar to but simpler than that in the IBM and Amdahl machines. A set-associative TLB (called the *translation buffer*) is used, with 64 sets of 2 entries each. (The VAX 11/750 has 256 sets of 2 entries each.) The set is selected by using the high-order address bit and the five low-order bits of the page address, so the address need not be hashed at all. Since the

higher order address bit separates the user from the supervisor address space, this means that the user and supervisor TLB entries never map into the same locations. This is convenient because the user part of the TLB is purged on a task switch. (There are no address space IDs.) The TLB is also used to hold the dirty bit, which indicates if the page has been modified, and the protection key.

Published figures for TLB performance are not generally available. The observed miss ratio for the Amdahl 470V/6 TLB is about 0.3 to 0.4 percent (Private Communication: W. J. Harding). Simulations of the VAX 11/780 TLB [SATY81] show miss ratios of 0.1 to 2 percent for TLB sizes of 64 to 256 entries.

2.17 Translator

When a virtual address must be translated into a real address and the translation does not already exist in the TLB, the *translator* must be used. The translator obtains the base of the segment table from the appropriate place (e.g., control register 1 in 370 machines), adds the segment number from the virtual address to obtain the page table address, then adds the page number (from the virtual address) to the page table address to get the real page address. This real address is passed along to the cache so that the access can be made, and simultaneously, the virtual address/real address pair is entered in the TLB. The translator is basically an adder which knows what to add.

It is important to note that the translator requires access to the segment and page table entries, and these entries may be either in the cache or in main memory. Provision must be made for the translator accesses to proceed unimpeded, independent of whether target addresses are cache or main memory resident.

We also observe another problem related to translation: "page crossers." The target of a fetch or store may cross from one page to another, in a similar way as for "line crossers." The problem here is considerably more complicated than that of line crossers since, although the virtual addresses are contiguous, the real addresses may not be. Therefore, when a page crosser occurs, two separate translations are required; these may occur in the TLB and/or translator as the occasion demands.

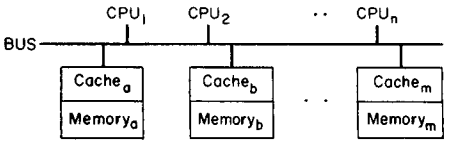

Figure 35. Diagram of computer system in which caches are associated with memories rather than with processors.

2.18 Memory-Based Cache

It was stated at the beginning of this paper that caches are generally associated with the processor and not with the main memory. A different design would be to place the cache in the main memory itself. One way to do this is with a shared bus interfacing between one or more CPUs and several main memory modules, each with its own cache (see Figure 35).

There are two reasons for this approach. First, the access time at the memory module is decreased from the typical 200–500 nanoseconds (given high-density MOS RAM) to the 50–100 nanoseconds possible for a high-speed cache. Second, there is no consistency problem even though there are several CPUs. All accesses to data in memory module i go through cache i and thus there is only one copy of a given piece of data.

Unfortunately, the advantages mentioned are not nearly sufficient to compensate for the shortcomings of this design. First, the design is too slow; with the cache on the far side of the memory bus, access time is not cut sufficiently. Second, it is too expensive; there is one cache per memory module. Third, if there are multiple CPUs, there will be memory bus contention. This slows down the system and causes memory access time to be highly variable.

Overall, the scheme of associating the cache with the memory modules is very poor unless both the main memory and the processors are relatively slow. In that case, a large number of processors could be served by a small number of memory modules with built-in caches, over a fast bus.

2.19 Specialized Caches and Cache Components

This paper has been almost entirely concerned with the general-purpose cache memory found in most large, high-speed computers. There are other caches and buffers that can be used in such machines and we briefly discuss them in this section.

2.19.1 Address Space Identifier Table

In many computers, the operating system identifier for an address space is quite long; in the IBM-compatible machines discussed (370/168, 3033, 470V), the identifier is the contents of control register 1. Therefore, these machines associate a much shorter tag with each address space for use in the TLB and/or the cache. This tag is assigned on a temporary basis by the hardware, and the correspondence between the address space and the tag is held in a hardware table which we name the Address Space Identifier Table (ASIT). It is also called the Segment Base Register Table in the 470V/7, the Segment Table Origin Address Stack in the 3033 [IBM78] and 370/168 [IBM75], and the Segment Base Register Stack in the 470V/6.

The 3033 ASIT has 32 entries, which are assigned starting at 1. When the table becomes full, all entries are purged and IDs are reassigned dynamically as address spaces are activated. (The TLB is also purged.) When a task switch occurs, the ASIT in the 3033 is searched starting at 1; when a match is found with control register 1, the index of that location becomes the address space identifier.

The 470V/6 has a somewhat more complex ASIT. The segment table origin address is hashed to provide an entry into the ASIT. The tag associated with that address is then read out. If the address space does not have a tag, a previously unused tag is assigned and placed in the ASIT. Whenever a new tag is assigned, a previously used tag is made available by deleting its entry in the ASIT and (in the background) purging all relevant entries in the TLB. (A complete TLB purge is not required.) Thirty-two valid tags are available, but the ASIT has the capability of holding up to 128 entries; thus, all 32 valid tags can usually be used, with little fear of hashing conflicts.

2.19.2 Execution Unit Buffers

In some machines, especially the IBM 360/91 [ANDE67b, IBM71, TOMA67], a number of buffers are placed internally in the execution unit to buffer the inputs and outputs of partially completed instructions. We refer the reader to the references just cited for a complete discussion of this.

2.19.3 Instruction Lookahead Buffers

In several machines, especially those without general-purpose caches, a buffer may be dedicated to lookahead buffering of instructions. Just such a scheme is used on the Cray I [CRAY76], the CDC 6600 [CDC74], the CDC 7600, and the IBM 360/91 [ANDE67a, BOLA67, IBM71]. These machines all have substantial buffers, and loops can be executed entirely within these buffers. Machines with general-purpose caches usually do not have much instruction lookahead buffering, although a few extra bytes are frequently fetched. See also BLAZ80 and KONE80.

2.19.4 Branch Target Buffer

One major impediment to high performance in pipelined computer systems is the existence of branches in the code. When a branch occurs, portions of the pipeline must be flushed and the correct instruction stream fetched. To minimize the effect of these disruptions, it is possible to implement a branch target buffer (BTB) which buffers the addresses of previous branches and their target addresses. The instruction fetch address is matched against the contents of the branch target buffer and if a match occurs, the next instruction fetch takes place from the (previous) target of the branch. The BTB can correctly predict the correct branch behavior more than 90 percent of the time [LEE82]. Something like a branch target buffer is used in the MU-5 [IBBE72, MORR79], and the S-1 [MCWI77].

2.19.5 Microcode Cache

Many modern computer systems are microcoded and in some cases the amount of microcode is quite large. If the microcode is not stored in sufficiently fast storage, it

is possible to build a special cache to buffer the microcode.

2.19.6 Buffer Invalidation Address Stack (BIAS)

The IBM 370/168 and 3033 both use a store-through mechanism in which any store to main memory causes the line affected to be invalidated in the caches of all processors other than the one which performed the store. Addresses of lines which are to be invalidated are kept in the buffer invalidation address stack (BIAS) in each processor, which is a small hardware implemented queue inside the S-unit. The DEC VAX 11/780 functions in much the same way, although without a BIAS to queue requests. That is, invalidation requests in the VAX have high priority, and only one may be outstanding at a time.

2.19.7 Input/Output Buffers

As noted earlier, input/output streams must be aborted if the processor is not ready to accept or provide data when they are needed. For this reason, most machines have a few words of buffering in the I/O channels or I/O channel controller(s). This is the case in the 370/168 [IBM75] and the 3033 [IBM78].

2.19.8 Write-Through Buffers

In a write-through machine, it is important to buffer the writes so that the CPU does not become blocked waiting for previous writes to complete. In the IBM 3033 [IBM78], four such buffers, each holding a double word, are provided. The VAX 11/780 [DEC78], on the other hand, buffers one write. (Four buffers were recommended in SMIT79.)

2.19.9 Register Cache

It has been suggested that registers be automatically stacked, with the top stack frames maintained in a cache [DITZ82]. While this is much better (faster) than implementing registers as part of memory, as with the Texas Instruments 9900 microprocessor, it is unlikely to be as fast as regular, hardwired registers. The specific cache described, however, is not general purpose, but is dedicated to holding registers; it therefore should be much faster than a larger, general-purpose cache.

3. DIRECTIONS FOR RESEARCH AND DEVELOPMENT

Cache memories are moderately well understood, but there are problems which indicate directions both for research and development. First, we note that technology is changing; storage is becoming cheaper and faster, as is processor logic. Cost/performance trade-offs and compromises will change with technology and the appropriate solutions to the problems discussed will shift. In addition to this general comment, we see some more specific issues.

3.1 On-Chip Cache and Other Technology Advances

The number of gates that can be placed on a microcomputer chip is growing quickly, and within a few years, it will be feasible to build a general-purpose cache memory on the same chip as the processor. We expect that such an on-chip cache will occur. There is research to be done in designing this within the constraints of the VLSI state of the art. (See LIND81 and POHM82.)

Cache design is also affected by the implementation technology. MOS VLSI, for example, permits wide associative searches to be implemented easily. This implies that parameters such as set size may change with changing technology. This related aspect of technological change also needs to be studied.

3.2 Multicache Consistency

The problem of multicache consistency was discussed in Section 2.7 and a number of solutions were indicated. Additional commercial implementations are needed, especially of systems with four or more CPUs, before the cost/performance trade-offs can be evaluated.

3.3 Implementation Evaluation

A number of new or different cache designs were discussed earlier, such as the split instruction/data cache, the supervisor/user

cache, the multilevel cache, and the virtual address cache. One or more implementations of such designs are required before their desirability can be fully evaluated.

3.4 Hit Ratio versus Size

There is no generally accepted model for the hit ratio of a cache as a function of its size. Such a model is needed, and it will probably have to be made specific to each machine architecture and workload type (e.g., 370 commercial, 370 scientific, and PDP-11).

3.5 TLB Design

A number of different TLB designs exist, but there are almost no published evaluations (but see SATY81). It would be useful to know what level of performance can be expected from the various designs and, in particular, to know whether the complexity of the Amdahl TLB is warranted.

3.6 Cache Parameters versus Architecture and Workload

Most of the studies in this paper have been based on IBM System 370 user program address traces. On the basis of that data, we have been able to suggest desirable parameter values for various aspects of the cache. Similar studies need to be performed for other machines and workloads.

APPENDIX. EXPLANATION OF TRACE NAMES

1. EDC PDP-11 trace of text editor, written in C, compiled with C compiler on PDP-11.
2. ROFFAS PDP-11 trace of text output and formatting program. (called ROFF or runoff).
3. TRACE PDP-11 trace of program tracer itself tracing EDC. (Tracer is written in assembly language.)
4. FGO1 FORTRAN Go step, factor analysis (1249 lines, single precision).
5. FGO2 FORTRAN Go step, double-precision analysis of satellite information, 2057 lines, FortG compiler.
6. FGO3 FORTRAN Go step, double-precision numerical analysis, 840 lines, FortG compiler.
7. FGO4 FORTRAN Go step, FFT of hole in rotating body. Double-precision FortG.
8. CGO1 COBOL Go step, fixed-assets program doing tax transaction selection.
9. CGO2 COBOL Go step, "fixed assets: year end tax select."
10. CGO3 COBOL Go step, projects depreciation of fixed assets.
11. PGO2 PL/I Go step, does CCW analysis.
12. IEBDG IBM utility that generates test data that can be used in program debugging. It will create multiple data sets of whatever form and contents are desired.
13. PGO1 PLI Go step, SMF billing program.
14. FCOMP FORTRAN compile of program that solves Reynolds partial differential equation (2330 lines).
15. CCOMP COBOL compile. 240 lines, accounting report.
16. WATEX Execution of a FORTRAN program compiled using the WATFIV compiler. The program is a combinatorial search routine.
17. WATFIV FORTRAN compilation using the WATFIV compiler. (Compiles the program whose trace is the WATEX trace.)
18. APL Execution of APL program which does plots at a terminal.
19. FFT Execution of an FFT program written in ALGOL, compiled using ALGOLW compiler at Stanford.

ACKNOWLEDGMENTS

This research has been partially supported by the National Science Foundation under grants MCS77-28429 and MCS-8202591, and by the Department of Energy under contract EY-76-C-03-0515 (with the Stanford Linear Accelerator Center).

I would like to thank a number of people who were of help to me in the preparation of this paper. Matt Diethelm, George Rossman, Alan Knowles, and Dileep Bhandarkar helped me to obtain materials describing various machines. George Rossman, Bill Harding, Jim Goodman, Domenico Ferrari, Mark Hill, and Bob

Doran read and commented upon a draft of this paper Mac MacDougall provided a number of other useful comments. John Lee generated a number of the traces used for the experiments in this paper from trace data available at Amdahl Corporation. Four of the traces were generated by Len Shustek. The responsibility for the contents of this paper, of course, remains with the author

REFERENCES

AGRA77a AGRAWAL, O. P., AND POHM, A. V. "Cache memory systems for multiprocessor architecture," in *Proc. AFIPS National Computer Conference* (Dallas, Tex. June 13-16, 1977), vol. 46, AFIPS Press, Arlington, Va , pp. 955-964.

AICH76 AICHELMANN, F. J. Memory prefetch. *IBM Tech Disclosure Bull.* **18,** 11 (April 1976), 3707-3708.

ALSA78 AL-SAYED, H. S. "Cache memory application to microcomputers," Tech. Rep. 78-6, Dep. of Computer Science, Iowa State Univ , Ames, Iowa, 1978

ANAC67 ANACKER, W., AND WANG, C. P. Performance evaluation of computing systems with memory hierarchies. *IEEE Trans Comput.* **TC-16,** 6 (Dec. 1967), 764-773.

ANDE67a ANDERSON, D. W., SPARACIO, F. J., AND TOMASULO, R. M. The IBM System/360 Model 91 Machine philosophy and instruction handling. *IBM J. Res. Dev.* **11,** 1 (Jan. 1967), 8-24

ANDE67b ANDERSON, S. F , EARLE, J. G , GOLDSCHMIDT, R. E., AND POWERS, D. M. The IBM System/360 Model 91 Floating point execution unit. *IBM J Res Dev* **11,** 1 (Jan. 1967), 34-53

ARMS81 ARMSTRONG, R. A Applying CAD to gate arrays speeds 32 bit minicomputer design. *Electronics* (Jan. 31, 1981), 167-173.

AROR72 ARORA, S. R., AND WU, F. L. "Statistical quantification of instruction and operand traces," in *Statistical Computer Performance Evaluation*, Freiberger (ed.), pp. 227-239, Academic Press, New York, N Y., 1972

BADE79 BADEL, M., AND LEROUDIER, J. "Performance evaluation of a cache memory for a minicomputer," in *Proc. 4th Int. Symp on Modelling and Performance Evaluation of Computer Systems* (Vienna, Austria, Feb. 1979).

BALZ81a BALZEN, D., GETZLAFF, K. J., HADJU, J., AND KNAUFT, G. Accelerating store in cache operations. *IBM Tech. Disclosure Bull.* **23,** 12 (May 1981), 5428-5429.

BALZ81b BALZEN, D., HADJU, J., AND KNAUFT, G. Preventive cast out operations in cache hierarchies. *IBM Tech Disclosure Bull.* **23,** 12 (May 1981), 5426-5427

BARS72 BARSAMIAN, H., AND DECEGAMA, A. "System design considerations of cache memories," in *Proc IEEE Computer Society Conference* (1972), IEEE, New York, pp 107-110

BEAN79 BEAN, B M., LANGSTON, K , PARTRIDGE, R., SY, K.-B Bias filter memory for filtering out unnecessary interrogations of cache directories in a multiprocessor system United States Patent 4,-142,234, Feb. 17, 1979.

BEDE79 BEDERMAN, S. Cache management system using virtual and real tags in the cache directory. *IBM Tech. Disclosure Bull.* **21,** 11 (April 1979), 4541.

BELA66 BELADY, L A A study of replacement algorithms for a virtual storage computer. *IBM Syst. J.* **5,** 2 (1966), 78-101.

BELL74 BELL, J , CASASENT, D., AND BELL, C. G. An investigation of alternative cache organizations. *IEEE Trans Comput.* **TC-23,** 4 (April 1974), 346-351.

BENN76 BENNETT, B. T., AND FRANACZEK, P A. Cache memory with prefetching of data by priority *IBM Tech. Disclosure Bull.* **18,** 12 (May 1976), 4231-4232.

BENN82 BENNETT, B. T , POMERENE, J. H , PUZAK, T. R., AND RECHTSCHAFFEN, R. N. Prefetching in a multilevel memory hierarchy. *IBM Tech. Disclosure Bull.* **25,** 1 (June 1982), 88

BERG76 BERG, H. S., AND SUMMERFIELD, A. R. CPU busy not all productive utilization. *Share Computer Measurement and Evaluation Newsletter,* no 39 (Sept. 1976), 95-97.

BERG78 BERGEY, A. L , JR. Increased computer throughput by conditioned memory data prefetching. *IBM Tech. Disclosure Bull* **20,** 10 (March 1978), 4103.

BLAZ80 BLAZEJEWSKI, T. J., DOBRZYNSKI, S. M , AND WATSON, W. D. Instruction buffer with simultaneous storage and fetch operations. *IBM Tech. Disclosure Bull.* **23,** 2 (July 1980), 670-672.

BLOU80 BLOUNT, F T., BULLIONS, R. J., MARTIN, D B., MCGILVRAY, B L., AND ROBINSON, J. R. Deferred cache storing method. *IBM Tech. Disclosure Bull* **23,** 1 (June 1980), 262-263

BOLA67 BOLAND, L J., GRANITO, G. D., MARCOTTE, A V., MESSINA, B. U , AND SMITH, J W. The IBM System/360 Model 91 Storage system. *IBM J Res Dev.* **11,** 1 (Jan 1967), 54-68.

BORG79 BORGERSON, B. R., GODFREY, M. D , HAGERTY, P. E , RYKKEN, T R. "The architecture of the Sperry Univac 1100 series systems," in *Proc. 6th Annual Symp Computer Architecture* (April 23-25, 1979), ACM, New York, N.Y., pp. 137-146.

CAMP76 CAMPBELL, J. E., STROHM, W. G., AND TEMPLE, J. L Most recent class used

search algorithm for a memory cache. *IBM Tech. Disclosure Bull.* **18,** 10 (March 1976), 3307–3308.

CDC74 CONTROL DATA CORP Control Data 6000 Series Computer Systems Reference Manual Arden Hills, Minn., 1974.

CENS78 CENSIER, L., AND FEAUTRIER, P. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.* **TC-27,** 12 (Dec. 1978), 1112–1118.

CHIA75 CHIA, D K. Optimum implementation of LRU hardware for a 4-way set-associative memory. *IBM Tech. Disclosure Bull* **17,** 11 (April 1975), 3161–3163.

CHOW75 CHOW, C. K. Determining the optimum capacity of a cache memory. *IBM Tech Disclosure Bull.* **17,** 10 (March 1975), 3163–3166.

CHOW76 CHOW, C. K. Determination of cache's capacity and its matching storage hierarchy. *IEEE Trans. Comput.* **TC-25,** 2 (Feb. 1976), 157–164.

CHU76 CHU, W. W., AND OPDERBECK, H Program behavior and the page fault frequency replacement algorithm. *Computer* **9,** 11 (Nov. 1976), 29–38

CLAR81 CLARK, D W, LAMPSON, B. W, PIER, K. A. The memory system of a high performance personal computer. *IEEE Trans Comput.* **TC-30,** 10 (Oct. 1981), 715–733

CLAR82 CLARK, D. W Cache performance in the VAX-11/780. To appear in *ACM Trans Comp. Syst. 1,* 1 (Feb. 1983).

COFF73 COFFMAN, E. G., AND DENNING, P J. *Operating Systems Theory.* Prentice-Hall, Englewood Cliffs, N.J, 1973.

CONT68 CONTI, C. J., GIBSON, D. H., AND PITKOWSKY, S. H. Structural aspects of the system/360 Model 85 *IBM Syst. J.* **7,** 1 (1968), 2–21

CONT69 CONTI, C. J. Concepts for buffer storage *IEEE Computer Group News* **2,** 8 (March 1969), 9–13

COSC81 COSCARELLA, A S., AND SELLERS, F F. System for purging TLB. *IBM Tech. Disclosure Bull* **24,** 2 (July 1981), 910–911.

CRAY76 CRAY RESEARCH, INC. Cray-1 Computer System Reference Manual. Bloomington, Minn., 1976.

DEC78 DIGITAL EQUIPMENT CORP. "TB/Cache/SBI Control Technical Description—Vax-11/780 Implementation," Document No. EK-MM780-TD-001, First Edition (April 1978), Digital Equipment Corp., Maynard, Mass., 1978.

DENN68 DENNING, P. J. The working set model for program behavior. *Commun. ACM* **11,** 5 (May 1968), 323–333.

DENN72 DENNING, P. J. "On modeling program behavior," in *Proc Spring Joint Computer Conference,* vol. 40, AFIPS Press, Arlington, Va., 1972, pp. 937–944.

DIET74 DIETHELM, M. A. "Level 66 cache memory," Tech. Info. Notepad I-114, Honeywell, Phoenix, Ariz., April, 1974.

DITZ82 DITZEL, D. R. "Register allocation for free: The C machine stack cache," in *Proc Symp on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., March 1–3, 1982), ACM, New York, N.Y., 1982.

DRIM81a DRIMAK, E. G., DUTTON, P. F., HICKS, G L., AND SITLER, W. R Multiprocessor locking with a bypass for channel references. *IBM Tech. Disclosure Bull.* **23,** 12 (May 1981), 5329–5331.

DRIM81b DRIMAK, E. G., DUTTON, P. F., AND SITLER, W. R. Attached processor simultaneous data searching and transfer via main storage controls and intercache transfer controls. *IBM Tech. Disclosure Bull.* **24,** 1A (June 1981), 26–27.

DRIS80 DRISCOLL, G C., MATICK, R. E, PUZAK, T. R., AND SHEDLETSKY, J. J. Split cache with variable interleave boundary. *IBM Tech Disclosure Bull.* **22,** 11 (April 1980), 5183–5186.

DUBO82 DUBOIS, M., AND BRIGGS, F. A. "Effects of cache concurrency in multiprocessors," in *Proc. 9th Annual Symp. Computer Architecture* (Austin, Texas, April, 1982), ACM, New York, N.Y, 1982, pp. 292–308.

EAST75 EASTON, M. C., AND FAGIN, R "Cold-start vs. warm-start miss ratios and multiprogramming performance," IBM Res. Rep RC 5715, Nov., 1975.

EAST78 EASTON, M C. Computation of cold start miss ratios. *IEEE Trans. Comput* **TC-27,** 5 (May 1978), 404–408.

ELEC76 ELECTRONICS MAGAZINE Altering computer architecture is way to raise throughput, suggests IBM researchers. Dec. 23, 1976, 30–31.

ELEC81 ELECTRONICS New TI 16-bit machine has on-chip memory. Nov 3, 1981, 57.

ENGE73 ENGER, T. A. Paged control store prefetch mechanism. *IBM Tech Disclosure Bull.* **16,** 7 (Dec. 1973), 2140–2141.

FAVR78 FAVRE, P., AND KUHNE, R. Fast memory organization. *IBM Tech. Disclosure Bull.* **21,** 2 (July 1978), 649–650.

FUKU77 FUKUNAGA, K., AND KASAI, T. "The efficient use of buffer storage," in *Proc. ACM 1977 Annual Conference* (Seattle, Wa., Oct. 16–19, 1977), ACM, New York, N.Y., pp. 399–403.

FURN78 FURNEY, R. W. Selection of least recently used slot with bad entry and locked slots involved. *IBM Tech Disclosure Bull.* **21,** 6 (Nov. 1978), 290.

GECS74 GECSEI, J. Determining hit ratios for multilevel hierarchies. *IBM J. Res. Dev.* **18,** 4 (July 1974), 316–327.

GIBS67 GIBSON, D H. "Consideration in block-oriented systems design," in *Proc Spring Jt Computer Conf,* vol 30,

Thompson Books, Washington, D.C., 1967, pp. 75–80

GIND77 GINDELE, J. D. Buffer block prefetching method. *IBM Tech Disclosure Bull.* **20,** 2 (July 1977), 696–697

GREE74 GREENBERG, B. S. "An experimental analysis of program reference patterns in the multics virtual memory," Project MAC Tech Rep. MAC-TR-127, 1974.

GUST82 GUSTAFSON, R. N., AND SPARACIO, F. J. IBM 3081 processor unit: Design considerations and design process. *IBM J Res. Dev.* **26,** 1 (Jan. 1982), 12–21.

HAIL79 HAILPERN, B., AND HITSON, B. "S-1 architecture manual," Tech. Rep No 161, Computer Systems Laboratory, Stanford Univ., Stanford, Calif., Jan, 1979.

HARD75 HARDING, W. J. "Hardware Controlled Memory Hierarchies and Their Performance." Ph.D. dissertation, Arizona State Univ., Dec., 1975.

HARD80 HARDING, W. J , MACDOUGALL, M. H., RAYMOND, W. J. "Empirical estimation of cache miss ratios as a function of cache size," Tech. Rep. PN 820-420-700A (Sept. 26, 1980), Amdahl Corp.

HOEV81a HOEVEL, L W., AND VOLDMAN, J. Mechanism for cache replacement and prefetching driven by heuristic estimation of operating system behavior. *IBM Tech Disclosure Bull.* **23,** 8 (Jan. 1981), 3923.

HOEV81b HOEVEL, L W., AND VOLDMAN, J Cache line reclamation and cast out avoidance under operating system control. *IBM Tech. Disclosure Bull.* **23,** 8 (Jan. 1981), 3912.

IBBE72 IBBET, R. The MU5 instruction pipeline. *Comput. J* **15,** 1 (Jan. 1972), 42–50.

IBBE77 IBBET, R. N., AND HUSBAND, M. A. The MU5 name store *Comput. J.* **20,** 3 (Aug. 1977), 227–231.

IBM71 IBM "IBM system/360 and System/370 Model 195 Functional characteristics," Form GA22-6943-2 (Nov. 1971), IBM, Armonk, N.Y.

IBM75 IBM "IBM System/370 Model 168 Theory of Operation/Diagrams Manual—Processor Storage Control Function (PSCF)," vol. 4, IBM, Poughkeepsie, N.Y., 1975.

IBM78 IBM "3033 Processor Complex, Theory of Operation/Diagrams Manual—Processor Storage Control Function (PSCF)," vol. 4, IBM, Poughkeepsie, N.Y., 1978.

IBM82 IBM "IBM 3081 Functional characteristics," Form GA22-7076, IBM, Poughkeepsie, N.Y., 1982.

JOHN81 JOHNSON, R. C. Microsystems exploit mainframe methods. *Electronics,* Aug. 11, 1981, 119–127

JONE76 JONES, J. D., JUNOD, D. M , PARTRIDGE, R. L., AND SHAWLEY, B. L Updating cache data arrays with data stored by other CPUs. *IBM Tech. Disclosure Bull.* **19,** 2 (July 1976), 594–596.

JONE77a JONES, J. D , AND JUNOD, D M. Cache address directory invalidation scheme for multiprocessing system. *IBM Tech. Disclosure Bull.* **20,** 1 (June 1977), 295–296.

JONE77b JONES, J D., AND JUNOD, D. M. Pretest lookaside buffer. *IBM Tech Disclosure Bull.* **20,** 1 (June 1977), 297–298.

KAPL73 KAPLAN, K. R , AND WINDER, R O. Cache-based computer systems. *IEEE Computer* **6,** 3 (March 1973), 30–36

KOBA80 KOBAYASHI, M. "An algorithm to measure the buffer growth function," Tech. Rep. PN 820413-700A (Aug. 8, 1980), Amdahl Corp

KONE80 KONEN, D. H., MARTIN, D. B., MCGILVRAY, B. L., AND TOMASULO, R. M. Demand driven instruction fetching inhibit mechanism. *IBM Tech Disclosure Bull.* **23,** 2 (July 1980), 716–717

KROF81 KROFT, D. "Lockup-free instruction fetch/prefetch cache organization," in *Proc. 8th Annual Symp. Computer Architecture* (Minneapolis, Minn., May 12–14, 1981), ACM, New York, N Y., pp. 81–87.

KUMA79 KUMAR, B. "A model of spatial locality and its application to cache design," Tech Rep. (unpubl.), Computer Systems Laboratory, Stanford Univ., Stanford, Calif., 1979.

LAFF81 LAFFITTE, D. S., AND GUTTAG, K M. Fast on-chip memory extends 16 bit family's reach. *Electronics,* Feb. 24, 1981, 157–161.

LAMP80 LAMPSON, B W , AND PIER, K. A. "A processor for a high-performance personal computer," in *Proc. 7th Annual Symp. Computer Architecture* (May 6–8, 1980), ACM, New York, N Y , pp 146–160.

LEE69 LEE, F. F. Study of "look-aside" memory. *IEEE Trans. Comput.* **TC-18,** 11 (Nov 1969), 1062–1064

LEE80 LEE, J M., AND WEINBERGER, A. A solution to the synonym problem. *IBM Tech. Disclosure Bull.* **22,** 8A (Jan 1980), 3331–3333.

LEE82 LEE, J. K. F., AND SMITH, A. J. "Analysis of branch prediction strategies and branch target buffer design," Tech. Rep., Univ. of Calif., Berkeley, Calif, 1982.

LEHM78 LEHMAN, A., AND SCHMID, D. "The performance of small cache memories in minicomputer systems with several processors," in *Digital Memory and Storage.* Springer-Verlag, New York, 1978, pp. 391–407.

LEHM80 LEHMANN, A. Performance evaluation and prediction of storage hierarchies. Source unknown, 1980, pp. 43–54.

LEWI71 LEWIS, P. A. W., AND YUE, P. C. "Statistical analysis of program refer-

ence patterns in a paging environment," in *Proc IEEE Computer Society Conference*, IEEE, New York, N Y., 1971.

LEWI73 LEWIS, P. A. W., AND SHEDLER, G. S. Empirically derived micro models for sequences of page exceptions. *IBM J. Res. Dev.* **17,** 2 (March 1973), 86–100.

LIND81 LINDSAY, D C. Cache memories for microprocessors. *Computer Architecture News* **9,** 5 (Aug. 1981), 6–13.

LIPT68 LIPTAY, J. S Structural aspects of the System/360 Model 85, II the cache. *IBM Syst J.* **7,** 1 (1968), 15–21

LIU82 LIU, L. Cache-splitting with information of XI-sensitivity in tightly coupled multiprocessing systems *IBM Tech. Disclosure Bull.* **25,** 1 (June 1982), 54–55

LOSQ82 LOSQ, J. J., PARKS, L S., SACHAR, H. E, AND YAMOUR, J. Conditional cache miss facility for handling short/long cache requests. *IBM Tech. Disclosure Bull.* **25,** 1 (June 1982), 110–111

LUDL77 LUDLOW, D M., AND MOORE, B. B. Channel *DAT* with pin bits. *IBM Tech. Disclosure Bull* **20,** 2 (July 1977), 683.

MACD79 MACDOUGALL, M. H. "The stack growth function model," Tech. Rep. 820228-700A (April 1979), Amdahl Corp.

MARU75 MARUYAMA, K mLRU page replacement algorithm in terms of the reference matrix. *IBM Tech Disclosure Bull* **17,** 10 (March 1975), 3101–3103.

MARU76 MARUYAMA, K. Implementation of the stack operation circuit for the LRU algorithm. *IBM Tech Disclosure Bull.* **19,** 1 (June 1976), 321–325

MATT71 MATTSON, R. L. Evaluation of multilevel memories. *IEEE Trans Magnetics* **MAG-7,** 4 (Dec. 1971), 814–819.

MATT70 MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L Evaluation techniques for storage hierarchies. *IBM Syst J.* **9,** 2 (1970), 78–117.

MAZA77 MAZARE, G "A few examples of how to use a symmetrical multi-micro-processor," in *Proc 4th Annual Symp. Computer Architecture* (March 1977), *ACM*, New York, N.Y., pp. 57–62.

MCWI77 MCWILLIAMS, T., WIDDOES, L C., AND WOOD, L "Advanced digital processor technology base development for navy applications: The S-1 Processor," Tech rep. UCIO-17705, Lawrence Livermore Laboratory, Sept , 1977.

MEAD70 MEADE, R. M. "On memory system design," in *Proc. Fall Joint Computer Conference*, vol. 37, AFIPS Press, Arlington, Va., 1970, pp. 33–43

MILA75 MILANDRE, G., AND MIKKOR, R. "VS2-R2 experience at the University of Toronto Computer Centre," in *Share 44 Proc.* (Los Angeles, Calif , March, 1975), pp. 1887–1895.

MORR79 MORRIS, D., AND IBBETT, R. N. *The MU5 Computer System.* Springer-Verlag, New York, 1979.

NGAI81 NGAI, C. H., AND WASSEL, E R. Shadow directory for attached processor system. *IBM Tech. Disclosure Bull.* **23,** 8 (Jan. 1981), 3667–3668.

NGAI82 NGAI, C. H., AND WASSEL, E. R. Two-level DLAT hierarchy. *IBM Tech. Disclosure Bull.* **24,** 9 (Feb. 1982), 4714–4715.

OHNO77 OHNO, N., AND HAKOZAKI, K. Pseudo random access memory system with CCD-SR and MOS RAM on a chip. 1977

OLBE79 OLBERT, A. G. Fast DLAT load for V = R translations. *IBM Tech. Disclosure Bull* **22,** 4 (Sept. 1979), 1434

PERK80 PERKINS, D. R. "The Design and Management of Predictive Caches." Ph D. dissertation, Univ of Calif., San Diego, Calif , 1980.

PEUT77 PEUTO, B. L., AND SHUSTEK, L. J. "An instruction timing model of CPU performance," in *Proc. 4th Annual Symp. Computer Architecture* (March 1977), ACM, New York, N.Y., pp. 165–178.

POHM73 POHM, A. V., AGRAWAL, O P., CHENG, C.-W., AND SHIMP, A. C. An efficient flexible buffered memory system. *IEEE Trans. Magnetics* **MAG-9,** 3 (Sept. 1973), 173–179.

POHM75 POHM, A. V., AGRAWAL, O. P., AND MONROE, R. N. "The cost and performance tradeoffs of buffered memories," in *Proc IEEE* **63,** 8 (Aug. 1975), pp. 1129–1135.

POHM82 POHM, A. V., AND AGRAWAL, O. P. "A cache technique for bus oriented multiprocessor systems," in *Proc. Compcon82* (San Francisco, Calif., Feb. 1982), IEEE, New York, pp. 62–66.

POME80a POMERENE, J. H., AND RECHTSCHAFFEN, R Reducing cache misses in a branch history table machine. *IBM Tech Disclosure Bull.* **23,** 2 (July 1980), 853.

POME80b POMERENE, J. H., AND RECHTSCHAFFEN, R N. Base/displacement lookahead buffer. *IBM Tech. Disclosure Bull.* **22,** 11 (April 1980), 5182.

POWE77 POWELL, M. L. "The DEMOS File system," in *Proc 6th Symp. on Operating Systems Principles* (West LaFayette, Ind., Nov. 16–18, 1977), ACM, New York, N.Y., pp. 33–42.

RADI82 RADIN, G. M. "The 801 minicomputer," in *Proc Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., March 1–3, 1982), ACM, New York, N.Y., pp 39–47.

RAMA81 RAMAMOHANARAO, K., AND SACKS-DAVIS, R. Hardware address translation for machines with a large virtual memory. *Inf. Process. Lett.* **13,** 1 (Oct. 1981), 23–29.

RAU76 RAU, B. R. "Sequential prefetch strategies for instructions and data," Digital

systems laboratory tech. rep. 131 (1976), Stanford Univ., Stanford, Calif

REIL82 REILLY, J., SUTTON, A., NASSER, R , AND GRISCOM, R. Processor controller for the IBM 3081. *IBM J. Res Dev.* 26, 1 (Jan. 1982), 22-29.

RIS77 RIS, F. N., AND WARREN, H S , JR. Read-constant control line to cache. *IBM Tech Disclosure Bull.* **20,** 6 (Nov. 1977), 2509-2510

ROSS79 ROSSMAN, G. Private communication. Palyn Associates, San Jose, Calif., 1979.

SALT74 SALTZER, J. H. "A simple linear model of demand paging performance, *Commun. ACM* **17,** 4 (April 1974), 181-186.

SATY81 SATYANARAYANAN, M., AND BHANDARKAR, D. Design trade-offs in VAX-11 translation buffer organization. *IEEE Computer* (Dec. 1981), 103-111.

SCHR71 SCHROEDER, M. D. "Performance of the GE-645 associative memory while multics is in operation," in *Proc. 1971 Conference on Computer Performance Evaluation* (Harvard Univ , Cambridge, Mass.), pp. 227-245.

SHED76 SHEDLER, G. S., AND SLUTZ, D. R. Derivation of miss ratios for merged access streams. *IBM J Res. Dev.* **20,** 5 (Sept. 1976), 505-517.

SLUT72 SLUTZ, D. R , AND TRAIGER, I. L. "Evaluation techniques for cache memory hierarchies," IBM Res. Rep. RJ 1045, May, 1972.

SMIT78a SMITH, A. J A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Trans. Softw. Eng* **SE-4,** 2 (March 1978), 121-130

SMIT78b SMITH, A. J. Sequential program prefetching in memory hierarchies. *IEEE Computer* **11,** 12 (Dec. 1978), 7-21

SMIT78c SMITH, A J Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.* **3,** 3 (Sept 1978), 223-247.

SMIT78d SMITH, A J. Bibliography on paging and related topics. *Operating Systems Review* **12,** 4 (Oct. 1978), 39-56.

SMIT79 SMITH, A J. Characterizing the storage process and its effect on the update of main memory by write-through *J. ACM* **26,** 1 (Jan. 1979), 6-27.

SNOW78 SNOW, E. A., AND SIEWIOREK, D. P. "Impact of implementation design tradeoffs on performance The PDP-11, A case study," Dep. of Computer Science Report (Feb. 1978), Carnegie-Mellon University, Pittsburgh, Pa

SPAR78 SPARACIO, F J. Data processing system with second level cache. *IBM Tech Disclosure Bull.* **21,** 6 (Nov. 1978), 2468-2469.

STEV81 STEVENSON, D. "Virtual memory on the Z8003," in *Proc IEEE Compcon* (San Francisco, Calif , Feb. 1981), pp 355-357

STRE76 STRECKER, W. D. "Cache memories for PDP-11 family computers," in *Proc. 3rd Annual Symp. Computer Architecture* (Jan. 19-21, 1976), ACM. New York, N.Y , pp. 155-158.

TANG76 TANG, C K. "Cache system design in the tightly coupled multiprocessor system," in *Proc. AFIPS National Computer Conference* (New York City, New York, June 7-10, 1976), vol. 45, AFIPS Press, Arlington, Va , pp. 749-753.

THAK78 THAKKAR, S. S. "Investigation of Buffer Store Organization." Master's of science thesis, Victoria University of Manchester, England, October, 1978.

TOMA67 TOMASULO, R. M An efficient algorithm for exploiting multiple arithmetic units. *IBM J Res. Dev* **11,** 1 (Jan. 1967), 25-33.

WILK71 WILKES, M. V. Slave memories and segmentation *IEEE Trans Comput* (June 1971), 674-675.

WIND73 WINDER, R. O. A data base for computer performance evaluation. *IEEE Computer* **6,** 3 (March 1973), 25-29.

YAMO80 YAMOUR, J. Odd/even interleave cache with optimal hardware array cost, cycle time and variable data part width. *IBM Tech. Disclosure Bull* **23,** 7B (Dec. 1980), 3461-3463.

YEN81 YEN, W. C., AND FU, K. S. "Analysis of multiprocessor cache organizations with alternative main memory update policies," in *Proc 8th Annual Symp. Computer Architecture* (Minneapolis, Minn., May 12-14, 1981), ACM, New York, N Y., pp. 89-105.

YUVA75 YUVAL, A. "165/HSB analysis," Share Inc., Computer Measurement and Evaluation, Selected Papers from the Share Project, vol III, pp. 595-606, 1975.

ZOLN81 ZOLNOWSKY, J. "Philosophy of the MC68451 memory management unit," in *Proc. IEEE Compcon* (San Francisco, Calif , Feb. 1981), IEEE, New York, pp. 358-361.

BIBLIOGRAPHY

ACKL75 ACKLAND, B. D., AND PUCKNELL, D. A. Studies of cache store behavior in a real time minicomputer environment. *Electronics Letters* **11,** 24 (Nov. 1975), 588-590.

ACKL79 ACKLAND, B. D. "A bit-slice cache controller," in *Proc. 6th Annual Symp. Computer Architecture* (April 23-25, 1979), ACM, New York, N.Y., pp 75-82.

AGRA77b AGRAWAL, O. P., ZINGG, R. J., POHM, A. V. "Applicability of 'cache' memories to dedicated multiprocessor systems," in *Proc. IEEE Computer Society Confer-*

ence (San Francisco, Calif., Spring 1977), IEEE, New York, pp 74–76

AMDA76 AMDAHL CORP. 470V/6 Machine Reference Manual. 1976.

BELL71 BELL, C G., AND CASASENT, D. Implementation of a buffer memory in minicomputers. *Comput Des* **10,** (Nov. 1971), 83–89.

BLOO61 BLOOM, L., COHEN, M., AND PORTER, S. "Considerations in the design of a computer with high logic-to-memory speed ratio," in *Proc Gigacycle Computing Systems* (Jan. 1962), AIEE Special Publication S-136, pp. 53–63.

BORG81 BORGWARDT, P A. "Cache structures based on the execution stack for high level languages," Tech Rep. 81-08-04, Dep. of Computer Science, Univ. of Washington, Seattle, Wa., 1981

BRIG81 BRIGGS, F. A., AND DUBOIS, M. "Performance of cache-based multiprocessors," in *Proc ACM/SIGMETRICS Conf. on Measurement and Modeling of Computer Systems* (Las Vegas, Nev., Sept. 14–16, 1981), ACM, New York, N.Y., 181–190.

CANN81 CANNON, J. W., GRIMES, D. W., AND HERMANN, B. D Storage protect operations. *IBM Tech Disclosure Bull.* **24,** 2 (July 1981), 1184–1186

CAPO81 CAPOWSKI, R. S., DEVEER, J A., HELLER, A. R., AND MESCHI, J. W. Dynamic address translator for I/O channels. *IBM Tech. Disclosure Bull.* **23,** 12 (May 1981), 5503–5508.

CASP79 CASPERS, P. G., FAIX, M., GOETZE, V., AND ULLAND, H. Cache resident processor registers *IBM Tech Disclosure Bull* **22,** 6 (Nov. 1979), 2317–2318.

CORD81 CORDERO, H, AND CHAMBERS, J B. Second group of IBM 4341 machines outdoes the first. *Electronics* (April 7, 1981), 149–152.

ELLE79 ELLER, J. E., III "Cache design and the X-tree high speed memory buffers." Master's of science project report, Computer Science Division, EECS Department, Univ of Calif., Berkeley, Calif., Sept, 1979.

FARI80 FARIS, S M, HENKELS, W H, VALSAMAKIS, E. A., AND ZAPPE, H. H. Basic design of a Josephson technology cache memory. *IBM J Res. Dev* **24,** 2 (March 1980), 143–154.

FARM81 FARMER, D. Comparing the 4341 and M80/42. *Computerworld*, Feb. 9, 1981.

FERE79 FERETICH, R. A., AND SACHAR, H. E. Interleaved multiple speed memory controls with high speed buffer *IBM Tech. Disc. Bull.* **22,** 5 (Octo. 1979), 1999–2000.

GARC78 GARCIA, L. C. Instruction buffer design. *IBM Tech. Disclosure Bull.* **20,** 11b (April 1978), 4832–4833.

HEST78 HESTER, R. L., AND MOYER, J. T. Split cycle for a shared data buffer array. *IBM Tech. Disclosure Bull.* **21,** 6 (Nov. 1978), 2293–2294.

HOFF81 HOFFMAN, R. L., MITCHELL, G. R., AND SOLTIS, F. G. Reference and change bit recording *IBM Tech. Disclosure Bull.* **23,** 12 (May 1981), 5516–5519.

HRUS81 HRUSTICH, J., AND SITLER, W. R. Cache reconfiguration. *IBM Tech. Disclosure Bull.* **23,** 9 (Feb. 1981), 4117–4118

IBM70 IBM "IBM field engineering theory of operation, System/360, Model 195 storage control unit buffer storage," first edition (Aug 1970), IBM, Poughkeepsie, N.Y.

IIZU73 IIZUKA, H., AND TERU, T. Cache memory simulation by a new method of address pattern generation. *J. IPSJ* **14,** 9 (1973), 669–676.

KNEP79 KNEPPER, R. W. Cache bit selection circuit. *IBM Tech. Disclosure Bull.* **22,** 1 (June 1979), 142–143.

KNOK69 KNOKE, P. "An analysis of buffered memories," in *Proc. 2nd Hawaii Int Conf. on System Sciences* (Jan. 1969), pp. 397–400.

KOTO76 KOTOK, A. "Lecture notes for CS252," course notes (Spring 1976), Univ. of Calif, Berkeley, Calif., 1976.

LANG75 LANGE, R. E, DIETHELM, M. A., ISHMAEL, P C. Cache memory store in a processor of a data processing system United States Patent 3,896,419, July, 1975.

LARN80 LARNER, R. A., LASSETTRE, E R., MOORE, B B., AND STRICKLAND, J. P. Channel DAT and page pinning for block unit transfers *IBM Tech Disclosure Bull.* **23,** 2 (July 1980), 704–705.

LEE80 LEE, P. A., GHANI, N., AND HERON, K. A recovery cache for the PDP-11. *IEEE Trans. Comput* **TC-29,** 6 (June 1980), 546–549.

LORI80 LORIN, H., AND GOLDSTEIN, B. "An inversion of the memory hierarchy," IBM Res. Rep. RC 8171, March, 1980.

MADD81 MADDOCK, R. F., MARKS, B. L., MINSHULL, J F., AND PINNELL, M. C. Hardware address relocation for variable length segments. *IBM Tech. Disclosure Bull* **23,** 11 (April 1981), 5186–5187

MATH81 MATHIS, J. R, MAYFIELD, M. J., AND ROWLAND, R. E Reference associative cache mapping. *IBM Tech. Disclosure Bull* **23,** 9 (Feb. 1981), 3969–3971

MEAD71 MEADE, R. M Design approaches for cache memory control. *Comput Des.* **10,** 1 (Jan 1971), 87–93

MERR74 MERRIL, B 370/168 cache memory performance. *Share Computer Measurement and Evaluation Newsletter,* no. 26 (July 1974), 98–101.

MOOR80 MOORE, B B, RODELL, J. T., SUTTON, A. J., AND VOWELL, J. D. Vary storage physical on/off line in a non-store-through cache system. *IBM Tech. Disclosure Bull.* **23,** 7B (Dec. 1980), 3329

NAKA74 NAKAMURA, T., HAGIWARA, H., KITAGAWE, H., AND KANAZAWE, M. Simulation of a computer system with buffer memory. *J IPSJ* **15,** 1 (1974), 26–33.

NGAI80 NGAI, C. H., AND SITLER, W. R. Two-bit DLAT LRU algorithm. *IBM Tech Disclosure Bull* **22,** 10 (March 1980), 4488–4490.

RAO75 RAO, G. S. "Performance analysis of cache memories," Digital Systems Laboratory Tech. Rep. 110 (Aug 1975), Stanford Univ., Stanford, Calif

RECH80 RECHTSCHAFFEN, R. N. Using a branch history table to prefetch cache lines. *IBM Tech Disclosure Bull* **22,** 12 (May 1980), 5539.

SCHU78 SCHUENEMANN, C. Fast address translation in systems using virtual addresses and a cache memory. *IBM Tech Disclosure Bull.* **21,** 2 (Jan. 1978), 663–664.

THRE82 THREEWITT, B A VLSI approach to cache memory. *Comput. Des.* (Jan. 1982), 169–173.

TRED77 TREDENNICK, H. L, AND WELCH, T. A. "High-speed buffering for variable length aperands," in *Proc. 4th Annual Symp. Computer Architecture* (March 1977), ACM, New York, N.Y., pp 205–210.

VOLD81a VOLDMAN, J., AND HOEVEL, L. W. "The fourier cache connection," in *Proc. IEEE Compcon* (San Francisco, Calif., Feb. 1981), IEEE, New York, pp. 344–354

VOLD81b VOLDMAN, J, AND HOEVEL, L. W. The software-cache connection. *IBM J Res. Dev.* **25,** 6 (Nov. 1981), 877–893.

WILK65 WILKES, M. V. Slave memories and dynamic storage allocation. *IEEE Trans. Comput.J* **TC-14,** 2 (April 1965), 270–271.

Received December 1979, final revision accepted January 1982