Memory transfers due to a cache miss are costly. Prefetching all memory references in very fast computers can increase the effective CPU speed by 10 to 25 percent.

Sequential Program Prefetching in Memory Hierarchies

Alan Jay Smith
University of California, Berkeley

Transfers of information between levels of an automatically managed memory hierarchy at the time the program references it (a miss) are usually costly in overhead operations and idle time. The fact that patterns of program execution and data access are largely sequential provides the opportunity to set up some means for predicting which sections (pages) of a program's memory address space are likely to accessed in the near future. By prefetching these pages before they are actually needed, system efficiency can be significantly improved.

The problem is to relate the type of prefetching to the page size and the memory size. We simulated these variables using several kinds of program address traces. It was found that prefetching small page sizes, such as those used in cache memories, is potentially effective, but is critically dependent on the details of how the cache memory is implemented. This led us to investigate the architecture of the cache memories for the IBM 370/168 and Amdahl 470V/6, and we show how prefetching might be properly implemented in these machines.

Memory hierarchies and prefetching

Large modern computer systems frequently employ an automatically managed memory hierarchy such as the one illustrated in Figure 1. Information is usually transferred to higher levels (or "fetched") on a demand basis whereby, when a datum is referenced and is found to be absent from the highest level of the hierarchy, it is copied from the highest level at which it is resident to some or all higher levels. The event of a missing datum is called a "miss"; it is also known, in the case of a datum missing from main memory, as a page fault. The miss ratio (to a given level of the hierarchy) is the fraction of all memory references resulting in a miss (that is, a fetch from a lower level).

Memory reference patterns govern prefetching. Certain patterns are commonly found in the memory referencing behavior of computer programs, and it is possible to use these patterns to attempt to predict which sections of a program's address space will be referenced next. Information that is fetched before it is actually accessed is prefetched. Prefetching is possible between any two levels of the memory hierarchy. Because demand fetches usually involve higher "costs" to the system than successful prefetches, prefetching can improve system efficiency.

The higher costs for demand fetches are associated with both CPU overhead and idle time. Demand transfers between cache and main memory are performed by the hardware while the CPU remains idle, but prefetches can be done in parallel, simultaneously with normal program operation, thus avoiding CPU idle time. With current technology, transfers at lower levels of the hierarchy require software intervention, but scheduling and initiating several of these transfers at once can often reduce the overhead per transfer. Although multiprogramming is usually used to prevent the CPU from becoming idle because of page faults, there is not always a program ready to run, and some idle time frequently occurs.

Prefetching has a system cost. The prefetch ratio is the ratio of the number of prefetch data transfers to the total number of memory references, and the

0018-9162/78/1200-0007$00.75 © 1978 IEEE

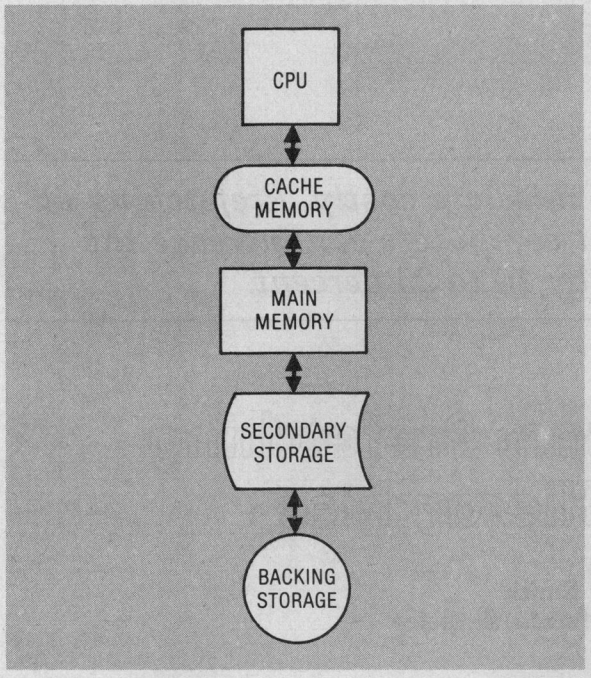


Figure 1. A typical memory hierarchy.

Previous work in prefetching

Input/output files are usually accessed sequentially[4-6] and multiple buffering techniques are standard for managing I/O data streams. Substantial sequentiality can also be seen in data-base systems,[1] and studies indicate that prefetching is very useful there as well. There is less agreement on the utility of sequential prefetching for pages in a virtual memory system.

Joseph[7] experimented with two prefetching algorithms and concluded that prefetching was not generally useful. Baer and Sager[8] used one of Joseph's algorithms and also experimented with two simple methods of nonsequential prefetching of pages (prepaging). They found that their best nonsequential algorithm produced decreases in the page-fault rate of from 2 percent to 14 percent for a page size of 256 words. The increase in the transfer rate was not indicated. Their results seemed only moderately sensitive to the prefetching algorithm. The best algorithm did about 27 percent better (miss ratio) than the worst in the most extreme case, and also seemed somewhat sensitive to the page size. Better results were achieved for small page sizes (256 words) than large ones (512 words).

In the special case of algorithms with highly predicated data reference patterns, such as algorithms that operate on arrays, some researchers have found prepaging to be very helpful.[9-13] Spaniol[14] found prepaging useful, but only, it appeared, when almost the entire program fit in the allocated memory. In this case, prepaging simply reads in the program faster. Pooch[15] suggested dynamically clustering pages that are referenced together and fetching clusters when any page in the cluster is accessed. However, there seems to be some question about the feasibility of this algorithm for normal system operation.

Sequential prefetching can also be applied to cache and other high-speed memories. Bennett and Franaczek[16] discuss an implementation of prefetching for set-associative cache memories.[17-19] Lookahead buffering of instruction fetches is used in many high-speed pipelined computers; Anderson et al.[20] discuss this for the IBM 360/91. Enger[21] is concerned with prefetching for a writable control store.

transfer ratio is the sum of the miss and prefetch ratios. If we assign a cost P to a prefetch and a cost D to a demand fetch, then the utility of prefetching depends on the proper relational operator for Equation 1:

$$D\text{*miss ratio(demand)} >=< D\text{*miss ratio(prefetch)} + P\text{*prefetch ratio} \quad (1)$$

where miss ratio(demand) is the miss ratio when no prefetching is used and miss ratio(prefetch) is the miss ratio when a prefetch algorithm is employed.

Prefetches may be frequent enough that it is not only possible that P*prefetch ratio is large, but also that miss ratio(prefetch) is greater than miss ratio(demand). This latter effect is a consequence of what we call memory pollution.[1] Every prefetch operation involves removing some other, already resident page from memory. If this removed page is referenced sooner than the prefetched one (which may not ever be referenced), this prefetch has increased the number of page faults. This is called memory pollution since memory is polluted with prefetched pages that will not be used and that have displaced other, more useful pages from memory.

Program sequentiality aids prefetching. Two types of behavior have been found to be characteristic of almost all programs: locality and sequentiality. Locality has two aspects: locality by time and locality by space.[2] Locality by time means that information recently referenced by a program is likely to be used again soon. That this type of behavior is observed should be expected from the fact that programs have loops. Locality by space means that portions of the address space near the current (or recent) locus of reference (data or instructions) is likely to be referenced in the near future. This type of behavior is again expected from common knowledge of programs—related data items (variables, arrays) are usually stored together, and instructions are executed sequentially.

Sequentiality is closely related to locality by space: programs will execute code sequentially, and when branches do occur, they are usually over short distances and forward.[3] Locality per se is of little aid in prefetching since it indicates that the most useful pages are those that have been recently referenced and are therefore already in memory. The principle of sequentiality, however, is quite helpful since it indicates that pages (blocks or lines) following the one accessed are likely to be referenced. As we note below, it is possible in special cases to correctly prefetch other than the next sequential block, but we are concerned for most of this article solely with sequential prefetching.

Research to determine utility. Equation 1 showed that the utility of prefetching could be considered by comparing costs with and without prefetching. The cost of a demand fetch D can be estimated for a specific implementation. The cost P for a prefetch varies depending on how and when the prefetch is

initiated and, often, with the load on the system. However, it can also be approximated for a given system.

The miss ratio for a given replacement algorithm can be measured from the program memory-reference trace, as can other statistics. For simple sequential prefetching, the miss and transfer ratios could be calculated from some simple set of program statistics; but for the class of page-replacement algorithms that we consider, this becomes hard to do. Other researchers, such as Joseph[12] and Baer and Sager,[3] have distinguished prefetched pages from demand-accessed pages, for example, by reserving a number of page frames for the exclusive use of such prefetched pages, and thus have been able to estimate analytically the effectiveness of prefetching.

The problem that occurs when prefetched pages are treated just like any other page is memory pollution; it is extremely difficult to estimate how much the miss ratio will be increased, because prefetched pages clutter up memory and require that other pages be removed. Elsewhere I have discussed two methods for approximating the effect of memory pollution, but here I will look at prefetching from a purely experimental and practical point of view.

We are concerned solely with sequential prefetching whereby, if page i is referenced, only page $i + 1$ (which is sequentially adjacent in the address space) is considered for prefetching. Other, more sophisticated algorithms either require prior analysis of the program in question or a greater amount of overhead in performing dynamic analysis. Our experiments suggest that prefetching is useful only for cache memories, where such dynamic analysis is clearly unfeasible. Sequential prefetching does allow considerable latitude, however, as to when to initiate a prefetch operation. To indicate a few possible variants, prefetching can be done on every memory reference, only at fault times, only for instruction fetches, or for data accesses or channel activity.

Experiments with sequential prefetching strategies

Simulations using IBM 360 instruction traces were run to study a variety of sequential prefetching strategies. The four traces used were Watfiv, the execution of the Watfiv compiler; Watex, the execution of a combinatorial search program written in Fortran and compiled by the Watfiv compiler; APL, an APL program that produces plots at a terminal; and FFT, the Fast Fourier Transform algorithm written in AloglW. These simulations were run both for uniprogramming (using a single trace) and multiprogramming. In the latter case, the program (trace) in control of the process was switched every Q time units (usually 10,000), where each memory reference required one time unit and each memory miss required R time units. The replacement algorithm in all cases was LRU, least recently used, and the memory mapping algorithm either full or set associative.

Notation clarifies prefetch strategies. All prefetch accesses in our experiments are made to the page with the next sequential (virtual) address after the currently referenced page. These prefetches can be initiated on all memory accesses or only when the

Definitions of terms

D	Cost of a demand fetch
Demand fetch	Transferring data from a lower level of the memory hierarchy to a higher level after the program has required it
Fault	The event of a datum being absent from the cache or main memory when requested. Often used to refer to main memory, in which case the term would be "page fault" or "segment fault"
Fully associative	A page of main memory may reside anywhere in the cache
LRU	Least recently used
Memory pollution	The situation that occurs when pages are prefetched into the cache or main memory and then not used
Miss	The event of a datum being absent from one of the higher levels of the memory hierarchy, such as the cache, when needed by the program. Also known in main memory as a page fault
Miss ratio	Fraction of memory references (to a given level of the memory hierarchy) resulting in a miss
Miss ratio (demand)	Miss ratio with no prefetching
Miss ratio (prefetch)	Miss ratio with prefetch
P	Cost of a prefetch
Prefetching	Transferring data from a lower level of a memory hierarchy to a higher level before it is needed by the program
Prefetch ratio	Number of prefetch data transfers to total number of memory references
Q	Quantum size or number of time units (usually set at 10,000) between program switches in multiprogramming, where the time unit is taken to be the time required by a reference to memory
S-unit	Storage unit in Amdahl computer, composed of cache memory and its associated registers, buffers, etc.
Sequentiality	Programs generally execute code in the order of address numbers and branches are usually forward a short distance
Set associative	A page of main memory is restricted to a *set* of locations in the cache.
TLB	Translation lookaside buffer. Used to quickly translate from virtual to real addresses
Transfer ratio	Miss ratio plus prefetch ratio

reference by the program causes a page fault. If the prefetching mechanism is implemented entirely in the hardware, it is feasible to consider prefetching on every memory reference, which is currently the case for cache memories but not main memories. For main-memory paging, page faults are currently handled by software, and without hardware support prefetching is feasible only at fault times. In addition, the overhead of prefetching for main memory includes supervisor-state CPU time to do such things as execute the page-replacement algorithm, construct a channel program to accomplish the transfer, switch tasks, and deal with the I/O interrupt when the fetch is completed.

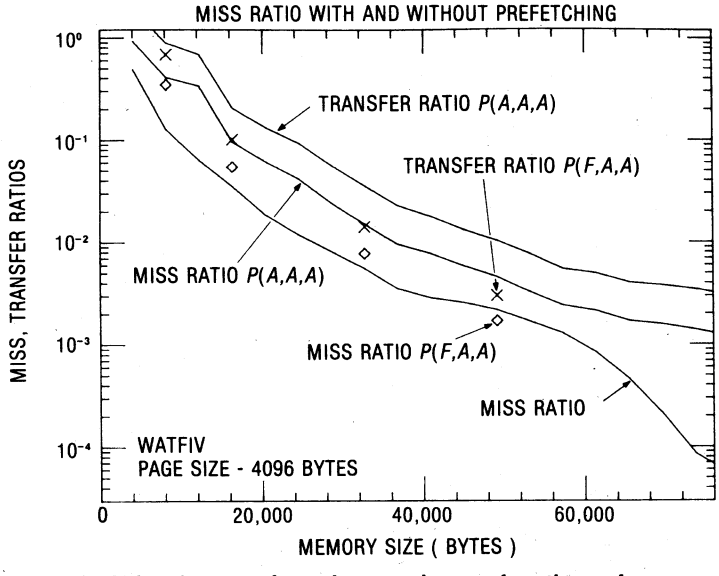


Figure 2. This miss transfer ratios are given as functions of memory size for the Watfiv program trace using a page size of 4096 bytes. Two different prefetch algorithms are used. *P(A,A,A,)* means that a prefetch was initiated on every memory reference (first *A*) and that the prefetched block is always placed at the top of the LRU replacement stack (second *A*). Both instruction fetches and data reads and writes caused prefetch operations (third *A*). *P(F,A,A)* indicates that prefetch operations were initiated only when the memory reference was one which resulted in a page fault (the *F*).

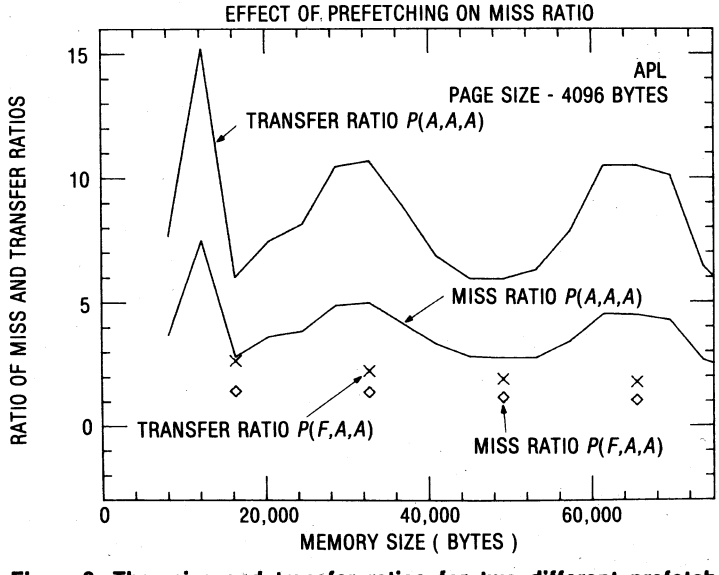


Figure 3. The miss and transfer ratios for two different prefetch algorithms are shown by giving their value as a multiple of (ratio to) the miss ratio for the given memory capacity when using only demand fetching. The APL program trace was used and a page size of 4096 bytes was employed.

The total overhead for transferring N (sequential) pages is only slightly higher than that for transferring one page, so sequential prefetching is a low-cost operation. Conversely, prefetching at other than fault times incurs all of the costs that appear for demand paging except (usually) multiprogramming idle. Therefore, prefetching at fault times is the only feasible strategy for software-controlled paging. It is also possible to limit prefetching to some subset of all memory references or a subset of those that cause page faults. For example, only instruction fetches (or data references) may be allowed to initiate a prefetch. We experimented with some of these possibilities as well.

Some researchers have treated prefetched pages differently from pages that have actually been referenced since they were fetched. While we do not choose to make the significant distinctions they have made, it is still necessary to consider the replacement status of prefetched pages in a special manner. When using LRU replacement, normal memory references always cause the referenced block to be placed at the top of the LRU stack.[22] If a prefetch reference finds the prefetched block already in memory, it can either (1) do nothing or (2) move that block to the top of the LRU stack.

It is necessary to create a notation that distinguishes these two replacement modes as well as the times when prefetches are initiated. This notation is used to specify a prefetch algorithm:

P(x,y,z)

where x indicates that prefetches are initiated on:
- every relevant memory reference (relevant being subject to z) with A being substituted for x and A standing for *always;* or only at *fault* times, with $x = F$;

where y takes on the values of:
- *A:* The prefetched block is *always* moved to the head of the stack, or
- *F:* the prefetched block is moved to the head of the stack only when accessing it causes a *fault*;

where z is used to indicate special conditions on when a prefetch is initiated:
- *A:* for *always,* subject to the value of x;
- *I:* for only *instruction* fetches;
- *D:* for only *data* fetches.

Miss ratio decreases for small page sizes. Our first set of experiments tested the effect of page size on the utility of prefetching. Figure 2 shows the transfer and miss ratios for the Watfiv trace using a page size of 4096 bytes. The miss ratio almost

always increases due to prefetching, and the more frequent the prefetching (for example, $P(A,A,A)$ as compared to $P(F,A,A)$) the higher the miss ratio. The transfer ratio, which takes into account the total increase in memory traffic, is higher still.

This figure, however, does not present the information in its most useful form; it would be more convenient to see directly the ratio of the prefetch miss and transfer ratios to the original miss ratio, since this comparison is difficult to make visually. Plotting the ratio would also improve the efficiency of the graphical plots, since all the curves would no longer run from the upper left to lower right of the frame. For this reason, Figures 3-12 display the ratio of each "ratio" (miss ratio, transfer ratio) to the original nonprefetching miss ratio. Thus, if the miss ratio due to prefetching is half of what it was, the point shown will be at a vertical position of 0.5 on the y axis. If the transfer ratio is thereby doubled, it will appear at 2.0 on the y axis.

Figure 3 shows the effect of prefetching pages of 4096 bytes for the APL trace, and it is clear that prefetching again performs badly. The fact that the miss-ratio curve is above 1.0 means that the miss ratio has increased. It can be seen here, and will also be evident in most other cases, that prefetching is generally a better idea for larger memory sizes. Here, this only means that prefetching degrades performance less for large memory capacities; in other cases it indicates that the improvement is larger. This occurs for two reasons: (1) prefetching when the memory is small means the page that is displaced was probably being used, whereas for a large memory size the page displaced is quite possibly idle, and (2) prefetching to a large memory is often a way of more quickly collecting needed pages without removing a page at all.

Figure 4 shows that, using a page size of 1024 bytes for the APL trace, prefetching performs poorly for small memory sizes but does lead to a slight decrease in the miss ratio for larger memory sizes. The same effect is visible in Figure 5, where a page size of 256 bytes has been used. Prefetching results in a 70 percent decrease in the miss ratio for only a 10 percent to 20 percent increase in the transfer ratio for large memory sizes.

For a page size of 32 bytes, as shown in Figure 6, prefetching seems to be useful independently of the memory size, although increases in memory size do increase the effectiveness of prefetching on performance. For large enough memory sizes, prefetching reduces the miss ratio to about 15 percent of its former level at a cost of less than 10 percent in increased transfer ratio.

In all these cases, the more frequently prefetches take place the bigger the improvement, if any improvement at all is experienced. The increased usefulness of prefetching as page size decreases is easily explained. The probability of needing a word that is k words away from the current locus of reference is generally decreasing with k; thus, for large page sizes, the probability of actually needing a prefetched page would decrease faster than the increased size of the page could include additional words that might be used. Also, the page that is removed to make room for the prefetched page is larger, and therefore presumably has more words in it that might actually be in use by the program.

Figure 7 shows the effect of prefetching pages of 32 bytes for the Watfiv trace; again, the miss ratio decreases substantially with prefetching. The shapes of the curves, however, appear to follow no obvious logic, and the peaks at 12K bytes seem to be artifacts of the behavior of this particular program. In comparing the results to those of Figure 6, we see a much smaller improvement than occurred for the APL trace. Even in the best case, the miss ratio is still about 35 percent of its former level, and

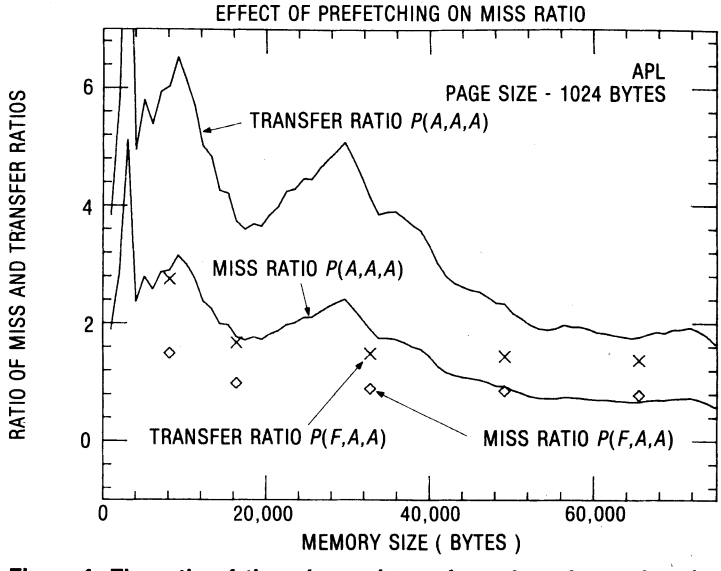


Figure 4. The ratio of the miss and transfer ratios when using the APL trace and a page size of 1024 bytes.

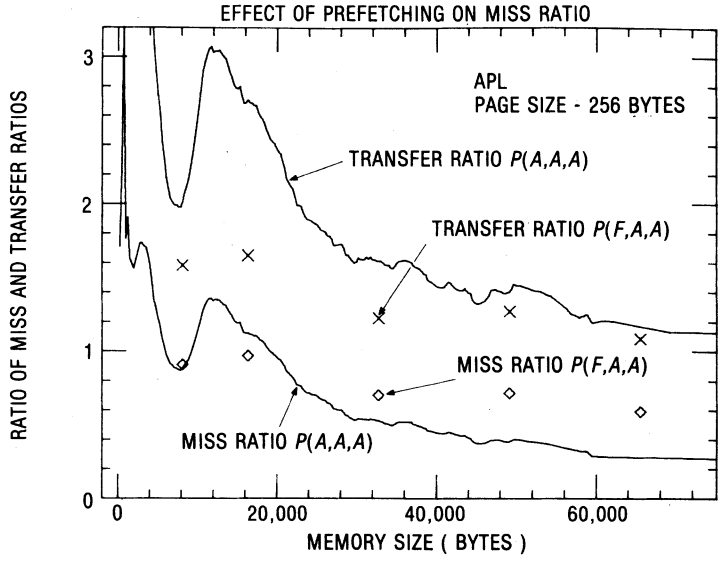


Figure 5. The ratio of the miss and transfer ratios when using the APL trace and a page size of 256 bytes.

Table 1.
Four-way multiprogramming using different page sizes.
(Watfiv-FFT-Watex-APL; quantum size Q = 10,000)

PAGE SIZE (BYTES)	PREFETCH STRATEGY	SETS	RATIO TO NORMAL MISS RATIO FOR DIFFERENT MEMORY SIZES					
			MISS RATIO			TRANSFER RATIO		
			16K	32K	64K	16K	32K	64K
64	$P(A,A,A)$	64	.504	.551	.397	1.461	1.682	1.489
64	$P(F,A,A)$	64	.698	.741	.663	1.255	1.299	1.188
128	$P(A,A,A)$	1	.599	.645	.468	1.551	1.684	1.569
128	$P(F,A,A)$	1	.931	.775	.659	1.698	1.401	1.187
256	$P(A,A,A)$	1	.660	.561	1.098	1.626	1.390	2.902
256	$P(F,A,A)$	1	.785	.692	.895	1.402	1.243	1.586
512	$P(A,A,A)$	1	.795	.875	1.981	1.761	1.942	4.789
512	$P(F,A,A)$	1	2.815	1.194	.963	5.252	3.655	1.697

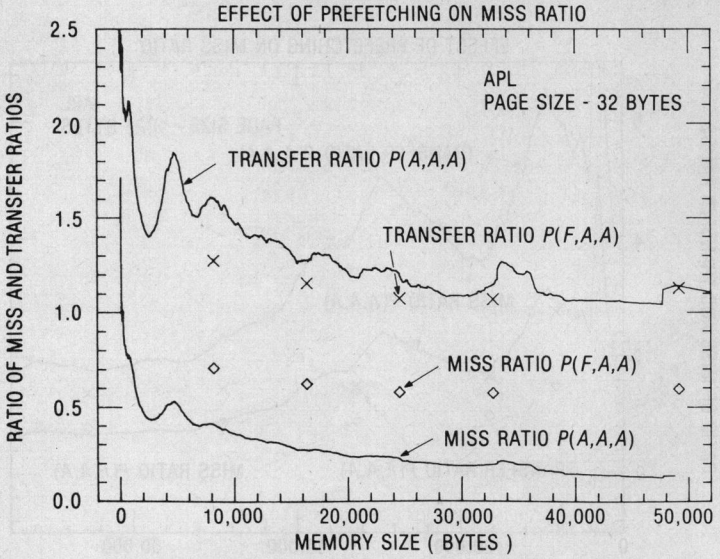


Figure 6. The ratio of the miss and transfer ratios when using the APL trace and a page size of 32 bytes shows prefetching to be useful, independent of memory size.

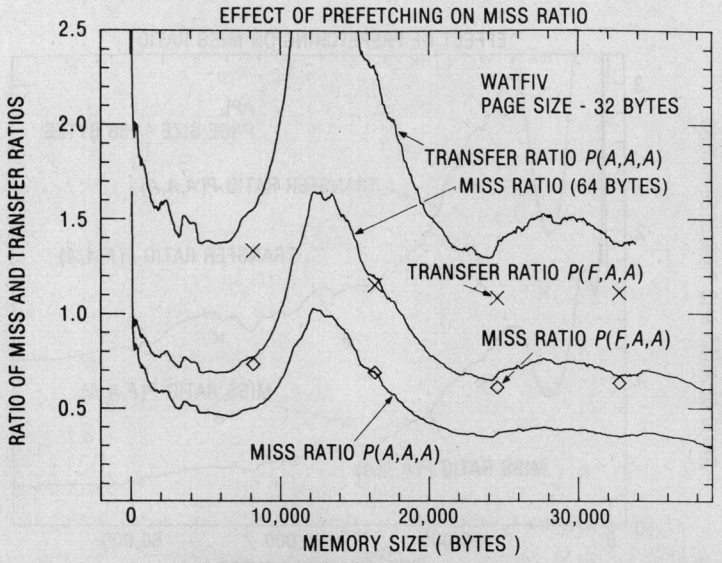


Figure 7. The ratio of the miss and transfer ratios when using a page size of 32 bytes and the Watfiv program trace. The miss ratio decreases with prefetching.

the transfer ratio is 40 percent to 50 percent above the original miss ratio.

It has been suggested that prefetching simply takes advantage of spatial locality and that successful prefetching implies the need for a larger page size. Therefore, we note that in Figure 7 the additional line labeled "miss ratio (64 bytes)" is the ratio of the miss ratio for a page size of 64 bytes to the original 32-byte-page miss ratio. We see that a page size of 64 bytes produces a miss ratio comparable (sometimes higher, sometimes lower) to that for 32 bytes, but never does as well as the *P(A,A,A)* prefetch algorithm. Every miss to a 64-byte page, of course, transfers twice as many bytes as a miss to a 32-byte page, so the transfer ratio is twice as high as the miss ratio for this case.

Simulations using four traces to imitate normal multiprogramming operation can be expected to display results that are less dependent on the individual peculiarities of each program. Figures 8, 9, and 10 show the results of simulations using four traces (Watfiv, Watex, APL, and FFT) for three different values of Q, the quantum size. As before, prefetching achieves a significant reduction in the miss ratio for 32-byte pages with only a small increase in the transfer ratio. The effect of the quantum size is minor, and the improvement using prefetching is consistent over the three values of Q shown. For a very large value of Q, a program gains control of the processor and finds that all of its pages have been removed from the cache. Prefetching is thus very useful in bringing those pages back. Its utility is less pronounced for smaller values of Q, (the probability of removing one's own pages increases), but prefetching is still helpful. Figure 9 also displays the miss ratio for a page size of 64 bytes; the value is again comparable to that for 32-byte pages without prefetching.

Four-way multiprogramming has also been used for a variety of other page sizes. Some selected experimental values appear in Table 1. Clear and significant decreases in the miss ratio occur for page sizes of 64 and 128 bytes, with only minor increases in the transfer ratio. For 256-byte pages, the decrease in the miss ratio is not consistent and the increase in the transfer ratio is greater. The measurements for 512-byte pages show large in-

Table 2.
Miss and transfer ratios for two different prefetch strategies.

PROGRAM	PAGE SIZE (BYTES)	PREFETCH STRATEGY	RATIO TO NORMAL MISS RATIO FOR DIFFERENT MEMORY SIZES							
			MISS RATIO				TRANSFER RATIO			
			4K	8K	12K	16K	4K	8K	12K	16K
WATEX	32	$P(F,A,A)$		.613		.580		1.134		1.079
WATEX	32	$P(A,A,A)$	.503	.428	.335	.382	1.939	1.696	1.435	1.625
			8K	16K	24K	32K	8K	16K	24K	32K
WATFIV	256	$P(F,A,A)$		.913		.716		1.611		1.228
WATFIV	256	$P(A,A,A)$	.858	.712	2.101	.907	2.083	1.791	5.67	2.362
			16K	32K	48K	64K	16K	32K	48K	64K
WATFIV	1024	$P(F,A,A)$	1.158	.923			1.879	1.461		
WATFIV	1024	$P(A,A,A)$	1.515	1.237	5.14	.725	3.171	2.379	11.76	2.206

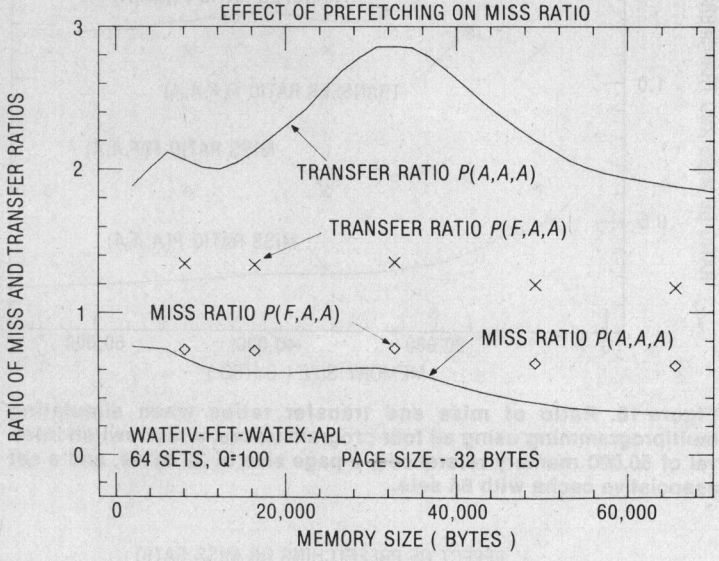
Figure 8 (at right). The ratio of the miss and transfer ratios when using all four (Watfiv, FFT, Watex, and APL) traces. Multiprogramming was simulated, with a task switch every 100 memory references. A page size of 32 bytes was used, as was set associate mapping with 64 sets. Also shown is the ratio of the miss ratio for 64-byte pages to that for 32-byte pages.

creases in the transfer ratio and no general reduction in the miss ratio. As before, we see that the frequency of prefetching changes primarily the magnitude, not the direction, of its effect.

Table 2 presents additional data in line with the above results.

Instruction or data prefetching only is less successful than always prefetching. Instructions are normally executed sequentially, whereas data-reference patterns need follow no particular logic. This suggests that instruction prefetching should work better than data prefetching. Figures 11 and 12 show experiments in which prefetches were made only on instruction references ($P(A,A,I)$), only on data references ($P(A,A,D)$), and on all memory references ($P(A,A,A)$). For small memory sizes, it can be seen that instruction prefetching is better than data prefetching, but that when the amount of available memory increases, data prefetching becomes superior. The better performance for instruction prefetching for small memory sizes is reasonable from our argument above. Conversely, one would expect much less pronounced sequentiality for data access, but it is not unexpected that if a prefetched block can remain in memory for a reasonable period of time, it will be used. Variables that are used together are often stored together, especially in the case of elements of arrays. It is important to note, though, that prefetching on every memory reference is far superior to limiting prefetching to only data or instruction references.

Prefetching reduces overruns. In the Amdahl 470V/6, all memory references, including those made by the I/O channel, pass through the cache (or high-speed buffer, as it is known in IBM/Amdahl



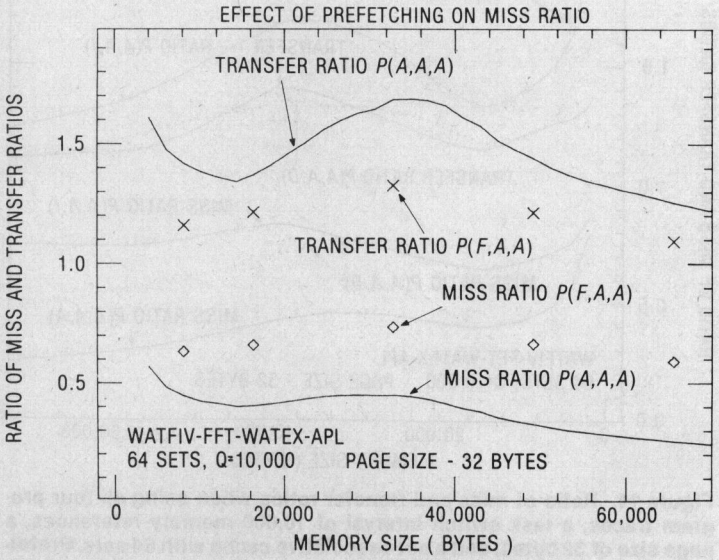


Figure 9. Ratio of miss and transfer ratios when using all four program traces, a task switch interval of 10,000 memory cycles, a page size of 32 bytes, and a set associative cache with 64 sets.

terminology). While sophisticated channel programs can transfer several records in one operation, each single record is usually transferred sequentially. When the cache page size is significantly smaller than the mean record size, it is clear that prefetching would be beneficial in reducing channel misses. Because the channel is constrained to transmit data for most devices at a specific, fixed rate, there may be "overruns." Overruns occur when the buffer cannot supply data to the channel or accept it as necessary from the channel and the channel is forced to abort the I/O operation. Prefetching can greatly reduce the severity of this problem.

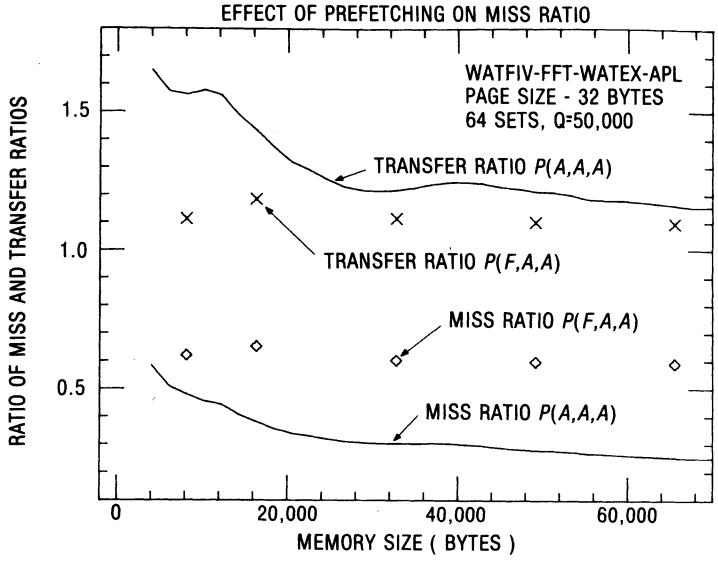


Figure 10. Ratio of miss and transfer ratios when simulating multiprogramming using all four program traces, a task switch interval of 50,000 memory references, a page size of 32 bytes, and a set associative cache with 64 sets.

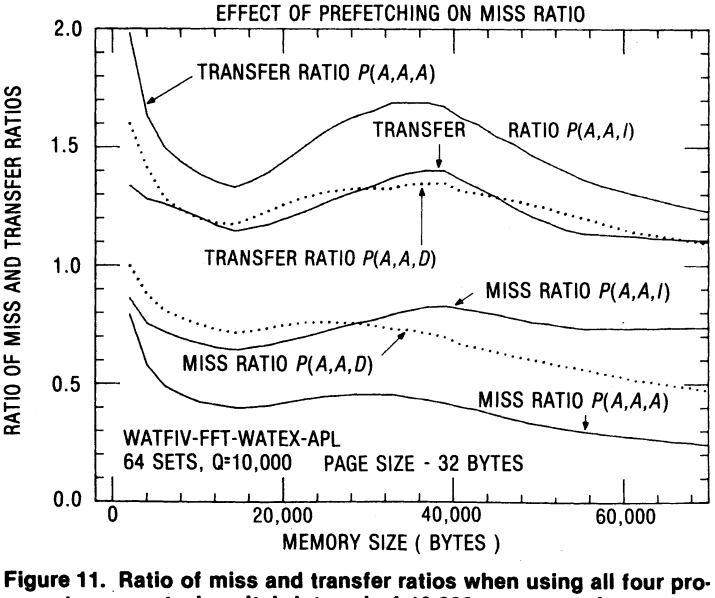


Figure 11. Ratio of miss and transfer ratios when using all four program traces, a task switch interval of 10,000 memory references, a page size of 32 bytes, and a set associative cache with 64 sets. Prefetching only on instruction fetches (*P(A,A,I)*) is compared with prefetching only on data references (*P(A,A,D)*), and prefetching on all memory references (*P(A,A,A)*).

Complicated prefetch strategies not useful. It is possible to prefetch by waiting for the *n*th access to a page rather than prefetching on the first. Alternatively, a prefetch might be initiated only when the latter half (quarter, etc.) of a page is accessed. Still a third possibility is to look for a pattern of sequential references within the page. In each of these cases, the implicit assumption is that prefetches should be initiated only when the reference pattern has been found to be sequential. There are two problems with this approach. If the reference pattern is truly sequential, there is every reason to expect that the prefetch transfer will not be complete by the time the next block is referenced. As we note in the next section, prefetches take many (10-20) machine cycles, and several instructions can be executed in this period. Therefore, the next page may be accessed before the prefetch is complete. Second, our measurements of the usefulness of data prefetching suggest that, although the next sequential page will often be used, it may not be used immediately, nor will the program be observed to be stepping sequentially through memory.

Updating the LRU stack makes little difference. If a prefetch access finds that the prefetched block is already in memory, it is not necessary to adjust the LRU stack to reflect this search. Using our earlier notation, *P(A,A,A)* means that we prefetch on every memory reference and update the LRU stack for the prefetched line. *P(A,F,A)* indicates that this update is performed only for prefetches that cause faults. Table 3 shows comparative data for these two cases for the Watfiv, APL, and four-trace simulations. There is no significant difference in the two cases. This is consistent with results reported by me[1] and by Ragaz and Rodriguez-Rosell,[23] looking at a similar question in the context of data-base data-block prefetching.

Number of sets not significant. Table 4 shows the result of prefetch experiments using our four-program simulation for two different set-associative mapping algorithms. In most cases, there is no significant difference. It is only in the case of 256 sets and 8K bytes of buffer that prefetch does not perform relatively as well as for 64 sets. This is because the 256-set buffer has only one element per set at 8K, whereas the 64-set buffer has eight pages per set. Thus, for the 256-set buffer, the probability that a prefetched page will displace a page in active use is higher and prefetching is less useful.

Prefetching useful in cache memory. From our data, it can be seen that simple sequential prefetching appears to work quite well for very small page sizes such as 32 and 64 bytes. Conversely, it works very poorly for the page sizes used to manage main memory. Since other researchers[8] have noted that

the exact prefetch algorithm makes a minor difference in the improvement or lack of it, this suggests that prefetching is not suitable for main memory use. Cache memories, though, have the potential for greatly improved operation when using properly implemented prefetching. A much lower miss ratio can be expected at only a small cost in additional transfers. These results hold true for several different program address traces, and are largely unaffected by the number of sets or the stack-reordering algorithm.

Two existing cache memory implementations

The benefits of prefetching do not automatically carry over to any arbitrary computer system in which a prefetching algorithm is implemented. First, the implementation must be such that prefetch operations do not impede the normal functioning of the cache. That is, there should be little or no conflict for cache access or main memory access. Second, the additional logic must not slow down the cache cycle time, since in many machines the cache is the unit that already prevents a faster machine cycle. Third, a certain amount of additional logic is required to include prefetching, and if the overall performance of the machine does not improve faster than the gate count, prefetching will not be cost effective. For these reasons we will discuss the detailed design of the cache memories for two large, modern high-speed computers, the Amdahl 470V/6 and the IBM 370/168.

The Amdahl 470V/6. A large high-speed computer that implements the IBM 370 series principles of operation,[24] the Amdahl 470V/6[25] (Figure 13) typically runs at about four million instructions per second, although some measurements[3] indicate speeds more than twice that fast. The cycle time is 32.5 nanoseconds, with the fastest instructions executing in two cycles. The implementation is high-speed ECL LSI.

The cache, or high-speed buffer as it is known, has a capacity of 16 kilobytes (16,384 bytes), a data path to the CPU of 4 bytes and to main memory of 8

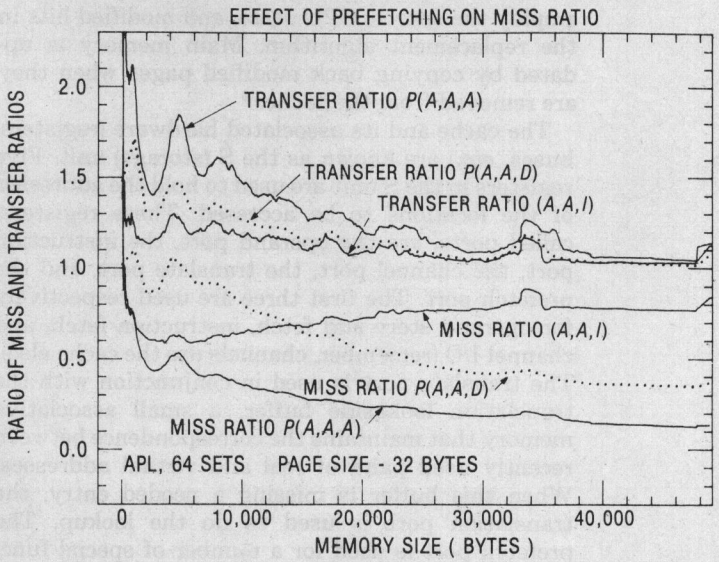


Figure 12. Ratio of miss and transfer ratios, using the APL program trace, a page size of 32 bytes, and a set associative cache of 64 sets. The miss and transfer ratios are compared using prefetching on only data references, only instruction references, and all memory references.

Table 3.
Miss and transfer ratios for different LRU stack updating schemes.

PROGRAM	PAGE SIZE (BYTES)	PREFETCH STRATEGY	RATIO TO NORMAL MISS RATIO FOR DIFFERENT MEMORY SIZES							
			MISS RATIO				TRANSFER RATIO			
			8K	16K	24K	32K	8K	16K	24K	32K
WATFIV	32	*P(A,A,A)*	.518	.565	.515	.362	1.567	1.991	2.034	1.366
WATFIV	32	*P(A,F,A)*		.554		.357		2.161		1.533
APL	32	*P(A,A,A)*	.444	.316	.223	.189	1.468	1.432	1.171	1.149
APL	32	*P(A,F,A)*		.318		.189		1.547		1.205
WATFIV-FFT-WATEX-APL	32	*P(A,A,A)*	.538	.408	.440	.442	1.387	1.374	1.533	1.641
WATFIV-FFT-WATEX-APL	32	*P(A,F,A)*		.411		.421		1.406		1.672

Table 4.
Miss and transfer ratios for two different set sizes.

PROGRAM	PAGE SIZE (BYTES)	PREFETCH STRATEGY	RATIO TO NORMAL MISS RATIO FOR DIFFERENT MEMORY SIZES							
			MISS RATIO				TRANSFER RATIO			
			8K	16K	24K	32K	8K	16K	24K	32K
WATFIV-FFT-WATEX-APL	32 (64 SETS)	*P(A,A,A)*	.538	.408	.440	.442	1.387	1.374	1.533	1.641
WATFIV-FFT-WATEX-APL	32 (256 SETS)	*P(A,A,A)*	.720	.426	.418	.427	4.195	1.425	1.448	1.537

bytes, and is arranged in 32-byte pages (or lines as Amdahl and IBM call them). It is set associative[17] with 256 sets and two pages per set. Main memory is up to four-way interleaved, using memory modules with data paths 32 bytes wide. Associated with each page in the cache are five status bits. These bits identify the source of the last access to that page (CPU, channel), the state (problem/supervisor) of the CPU if/when it did the fetch, the state of the page (modified/unmodified/empty), and the LRU status of the page (most recently/least recently used). It is possible (but not currently done) to employ the source, CPU state, and modified bits in the replacement algorithm. Main memory is updated by copying back modified pages when they are removed from the cache.[26]

The cache and its associated hardware (registers, buses, etc.) are known as the S (storage) unit. Five registers in the S unit are used to hold the addresses of the locations to be accessed. These registers, called 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 (remember, channels use the cache also). The translate port is used in conjunction with the translation lookaside buffer, a small associative memory that maintains the correspondence between recently used pairs of real and virtual addresses. When this buffer is missing a needed entry, the translation port is used to do the lookup. The prefetch port is used for a number of special functions (such as setting the storage key or purging the lookaside buffer) and for prefetch operations. Transfers to and from the S unit pass through one of three data registers, known as the instruction word register, the operand word register, and the channel word register.

A complete read of the cache requires four cycles, known as the P, B1, B2, and R cycles. 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 (more than one of the ports may be occupied and be demanding service). The B1 and B2 cycles are used to actually access the cache and the translation lookaside buffer, to select the appropriate page from the cache, check to make sure that the contents of the page are valid, and shift to get at the correct byte location out of the two-word (8-byte) segment fetched from the page.

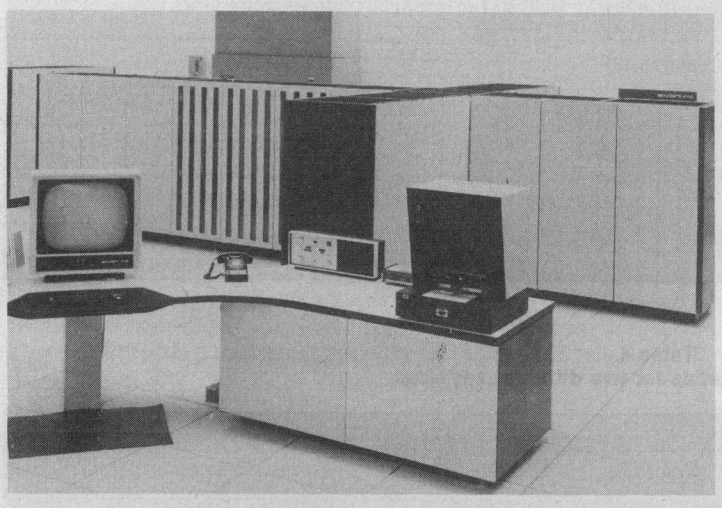


Figure 13. The Amdahl 470V/6, the first of the IBM-compatible central processors, implements the IBM series 370 instruction set at a speed of about four million instructions per second. Its cache memory has a capacity of 16K bytes, arranged in 32-byte pages.

Set-associative mapping is used in the cache, which means that some of the high-order real-address bits of the page are used to select a "set" in the cache that can then be searched associatively to see if the desired page is actually in the buffer. All but two of the necessary bits are available initially from the virtual address, so it is possible to limit the search of the buffer to four sets of two pages each. These eight pages are read out, while in parallel the translation lookaside buffer is accessed to get the real address corresponding to the virtual address. Associated with each page in the cache is a set of tag bits specifying the real memory address from which this page is taken. The real address, after it is obtained from the TLB or main memory, is immediately used to narrow the possible page locations for the desired page to two from the eight that were possible using the virtual address. The tag bits from these two pages are then compared with the real address; if a match is found, then the page is in the associated page position in the cache. A read takes four cycles (three excluding the priority cycle). The time required by a store is longer since it is essentially a read followed by a modify and a write-back; it takes six cycles. The operations in the cache are overlapped (pipelined) in such a way that a read can be performed once for every cycle or a write once for every other cycle.

The timing in the event a cache access is unsuccessful—that is, there is a miss—is given in Figure 14.[27] The buffer access is shown as beginning with a B1 cycle (cycle 1). If a move-out is required (write-back to memory of a modified page), four buffer cycles (5,6,7,8) are used to copy the modified page to a special holding register, while at the same time a main-memory busy cycle starts. The actual transfer of the fetched page (called a move-in) also requires four cycles (17,18,19,20). The original fetch operation is restarted in cycle 21; thus, each cache miss results in a loss of 20 machine cycles. Particularly important is the lower part of this diagram, which shows when the cache is busy and when the main memory is busy. We shall be especially concerned with prefetch operations and the extent to which they interfere with normal cache or main-memory accesses.

An optional prefetch algorithm is built into the 470V/6 cache but has not been used. It permits a prefetch to be initiated on either the first or second access to the page, provided that the access is to the nth (n = 1,2,3, or 4) or succeeding 8-byte (quarter-page) section. Prefetching can be selectively enabled

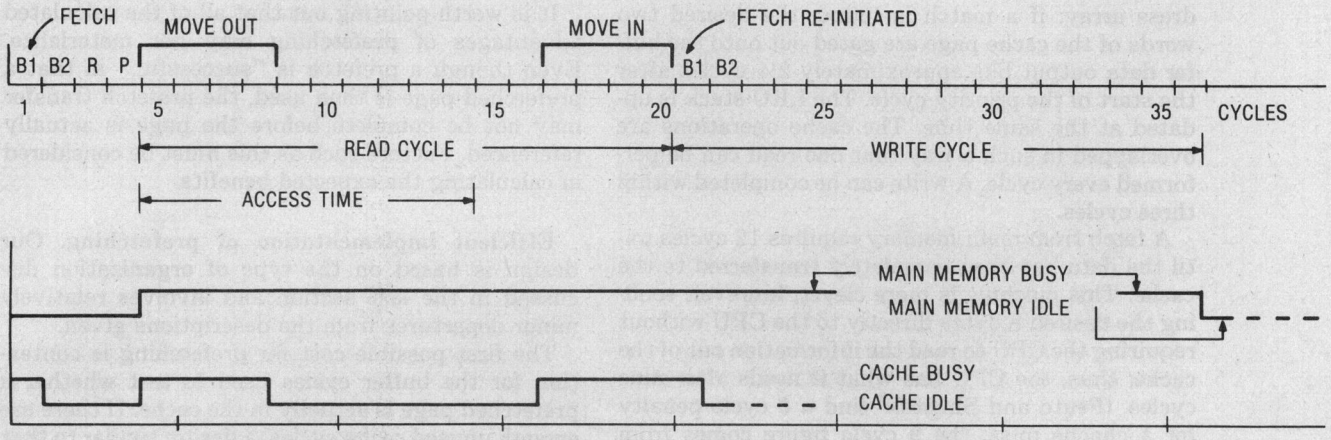


Figure 14. Cache miss timing, Amdahl 470V/6.

for instruction fetch, operand access, or channel I/O. Prefetch operations are initiated by placing the prefetch address in the prefetch port; these prefetch operations are tagged as such and have the lowest priority for cache cycles. In addition, any later request for access to the prefetch port can (optionally) overwrite a previous but still unexecuted prefetch request.

The IBM 370/168-3. The IBM 370/168-3[29,24] (Figure 15) is the fastest IBM computer with virtual memory capability actually delivered to customer sites. (At this writing, the model 3033 processor has not been delivered, and little documentation is available for it.) The 370/168 is approximately 70 percent as fast as the Amdahl 470V/6 and from a software point of view appears to be identical. The Model 3 (168-3), a slight modification of earlier 370/168 models, is slightly faster in some instructions and has a larger high-speed buffer. The machine cycle time is 80 nanoseconds, and the fastest instructions execute in one cycle.

The cache of the 370/168-3 is 32K bytes, arranged in 128 sets of eight 32-byte pages. The data-path width to both main memory and the CPU is 8 bytes. Main memory is interleaved up to four way in modules 8 bytes wide. Each set in the cache contains eight pages arranged in two groups of four. Replacement is a two-step process that is "LRU like." The eight pages in a set are associated in four pairs. The LRU pair is selected, and then the LRU page of that pair is the one chosen for replacement.

All storage operations by the CPU are directed to main memory. Only if a page to be updated is also in the cache is a write operation made to the cache. The channels in this machine talk directly to main memory without using the cache. The buffer invalidation stack (BI)maintains consistency between main memory and the cache. Modifications to main memory from other than the CPU (for example, from the channel) are entered in the BI stack, and if the accessed page is found in the cache as well, it is removed from the cache. (Because of the store-through operation, main memory is always valid.)

Figure 15. The IBM 370/168 is the largest CPU in the IBM 370 line of machines (as of 1977). The cache memory of the 168-3 model has a capacity of 32K bytes, arranged in 128 sets of eight 32-byte pages.

This same mechanism permits IBM's multiprocessor systems (luckily, I believe) to share memory conveniently.

The timing of the 168 is similar to that of the 470V/6. A priority cycle is used to establish priority. The tag bits for each page, giving its real memory address, are stored (unlike the 470) separately from the page in an address array. The output of the TLB is compared with the appropriate entries in the ad-

dress array; if a match is found, the desired two words of the cache page are gated out onto the buffer data output bus approximately 2½ cycles after the start of the priority cycle. The LRU stack is updated at the same time. The cache operations are overlapped in such a way that one read can be performed every cycle. A write can be completed within three cycles.

A fetch from main memory requires 12 cycles until the data has been completely transferred to the cache. This machine is more clever, however, sending the desired 8 bytes directly to the CPU without requiring the CPU to read the information out of the cache; thus, the CPU has what it needs after nine cycles. (Peuto and Shustek[3] find a 6 cycle penalty for a chache miss; the 9 cycle figure comes from documentation published by IBM[29].)Other accesses to main memory are inhibited for the period from 2½ to 11½ cycles after the start of the priority cycle for the fetch.

A good prefetch algorithm must be well implemented

There are two issues in the design of the cache for a high-speed computer system: the abstract prefetching algorithm and the detailed implementation. It is very important to make this distinction and to pursue each of these issues. Obviously, a good implementation cannot save a bad algorithm, but a bad implementation can ruin a good algorithm.

Analysis of these IBM and Amdahl computers indicates that certain design changes could implement prefetching effectively.

On the basis of the discussion in the previous section, it is simple to choose both a prefetching algorithm and some cache design parameters: prefetch the next sequential page on every memory reference, use a set size of at least two, have fairly small pages (say, 32 bytes), and omit updating the LRU status of prefetched pages (since it does not appear to matter.)

The design of the prefetching implementation is a matter of recognizing the possible costs in prefetching and then creating mechanisms to avoid or minimize these costs. To do a prefetch, it is necessary to access the cache, but this access must not interfere with regular use of the cache by the program. The transfer of a page from main memory to cache, and possibly the reverse, means that the main memory module(s) involved will be busy for a read and possibly a write cycle. The actual move-in and move-out from the cache require cache cycles and thus may interfere with normal buffer access. Finally, the TLB is also a possible point of contention, and prefetch access to it must not unduly interfere with regular use.

It is worth pointing out that all of the calculated advantages of prefetching may not materialize. Even though a prefetch is "successful," in that a prefetched page is then used, the prefetch transfer may not be complete before the page is actually referenced. Factors such as this must be considered in calculating the expected benefits.

Efficient implementation of prefetching. Our design is based on the type of organization discussed in the last section and involves relatively minor departures from the descriptions given.

The first possible cost for prefetching is contention for the buffer cycles used to test whether a prefetched page is actually in the cache. If there are enough unused cache cycles, a design similar to that used in the 470V/6 would work. That is, prefetch cycles would be normal cache cycles, but would be requested at very low priority so as not to interfere with normal operation. It is not reasonable to assume that there are spare cache cycles, however, and therefore we make the following observation: the prefetched page is never in the same set as the page being accessed. If the circuitry that reads out and examines tag bits and selects the right page were duplicated, then it would be possible to do prefetch accesses in parallel with the normal lookup, and with no memory interference. Actually, as explained below, it may be necessary to duplicate only the readout circuitry.

Cache lookups are done by comparing the real address of the desired page against the tag bits from the appropriate set of the cache. The real address of the prefetched page is required for this lookup just as the real address of the accessed page is. The real address of the prefetched cache page, however, can almost always be computed without the use of the TLB since the prefetched cache page is almost always in the same main memory page and therefore has the same high-order real-address bits; only when a page boundary is crossed would a TLB lookup be needed. (This case is so rare that prefetches could be omitted.) For example, 32-byte cache pages and 4096-byte main memory pages imply that only one time in 128 (0.78 percent) would a main-memory page boundary be crossed. We also note that, since the prefetch page address is computed from the normal page address, it is therefore available slightly later than the information for the regular access. If this delay is adjusted properly and the circuitry is fast enough, it may be possible to use the same set of comparators to search the tag bits. (The prefetch usually is not urgent and can easily be delayed a fraction of a cycle or more.)

If a prefetch access finds the desired page missing from the cache, it is necessary to (1) schedule a move-in of the prefetched page and (2) a move-out, if any is needed, of the page to be expelled from the cache. To minimize interference, we create two buffers, the move-in buffer and the move-out buffer. The move-out buffer is used to hold moved-out data until the appropriate main memory module is ready to accept the data. If this is the same main memory

module as was affected by the move-in, the move-in takes precedence for access to main memory. The move-in buffer is used to hold a prefetched page until cache cycles are available to receive it. It is not needed for demand move-ins.

A priority is associated with each access request to the cache, and we likewise assign priorities to prefetch move-in and move-out requests. Move-out cycles are issued at a low priority (compared to most cache accesses) until a timer indicates that a demand move-in is imminent, at which time the priority changes to high. Move-in requests are also issued at two levels of priority, low and high. Low priority is used for a prefetch request, and the page to be moved in is kept in the move-in buffer temporarily. Move-ins due to genuine misses always occur at high priority and bypass the move-in buffer. A demand fetch also causes a move-out—from the set into which the incoming page will be placed—to proceed at high priority so that there is somewhere to put the fetched page. Both the move-in and the move-out buffers can be designed to hold more than one cache page, but more detailed studies than this one are required to determine whether this is useful; it appears unlikely to me. It may happen that a line that is accessed is actually in the process of being either moved in or moved out and is therefore resident in the move-in or move-out buffer. It must be possible to recognize this possibility and deal with it correctly. This is a good reason to keep the move-in and move-out buffers small (say, of capacity 1) and to avoid significant additional hardware complexity.

Prefetch transfers that are initiated while a previous prefetch is in progress can either be queued (up to some limited number) or discarded. Again, more detailed simulations, using actual operation-times, are required.

CPU efficiency increased. We identified four major costs in doing prefetching: prefetch lookup, move-in and move-out cycles, busy main memory, and TLB contention. Prefetch lookup is done in parallel, TLB accesses are rarely needed for prefetch, and move-ins and move-outs for prefetching have been buffered, so the cost of prefetching consists primarily of main-memory module-access conflicts and occasional unbuffered move-in or move-out conflicts. In addition, we must consider access to prefetched but unarrived pages. We consider each of these by examining the Amdahl machine.

A prefetch could possibly degrade system operation by busying main memory modules that would otherwise be able to more quickly satisfy a demand fetch. The effect of this factor can be estimated by considering the timing diagram (Figure 14) and some operating statistics for the 470V/6. The machine runs without prefetching at about a 5 percent miss ratio to cache, makes about two memory references per instruction, and has slightly over 30 million buffer cycles per second. Therefore, approximately one-fourth of all buffer cycles are used (at 4 mips), and a miss occurs about once every 80 cycles.

The main memory cycle time is 16 cycles; therefore, the probability that a miss occurs while a main memory module is busy servicing a prefetch is approximately 18 percent. If we assume that this miss is equally likely to occur at any time during those 16 cycles (a pessimistic assumption), this is an expected penalty of 0.18 x 8 x 0.25 = 0.36 cycles (since main memory is interleaved four ways) or an increase of 1.8 percent in the delay for a miss.

As we observed above, it is possible that a prefetched page will be needed before its fetch is complete. In Figure 16 we show the time (in memory references) between the time a (useful) prefetch is begun (using *P(A,F,A)*, *P(A,F,I)* and *P(A,F,D)* prefetching and a cache size of 16K) and the time the prefetched page is first accessed, broken down by instruction and data references. Using the calculation that one memory reference occurs every four machine cycles, only 10 percent to 15 percent of all successful prefetches would occasion a delay. Again we assume that the prefetch transfer is half complete at the time the page is needed. Then 10 percent to 15 percent of the time, half of the value of the prefetch is lost, which is a minor problem.

The effective instruction execution time for the Amdahl 470V/6 is on the order of five to six machine cycles.[3] For a miss ratio of 5 percent and two memory references per instruction, this is a penalty of two cycles on top of the mean instruction time, which is a slowdown of 25 percent to 30 percent. Our experiments earlier suggest that the miss ratio can be cut at least in half by prefetching, and we further found that most of this cut in the miss ratio can be reflected in improved performance. Therefore, we could expect improvements in CPU performance of 10 to 25 percent for sequential cache prefetching.

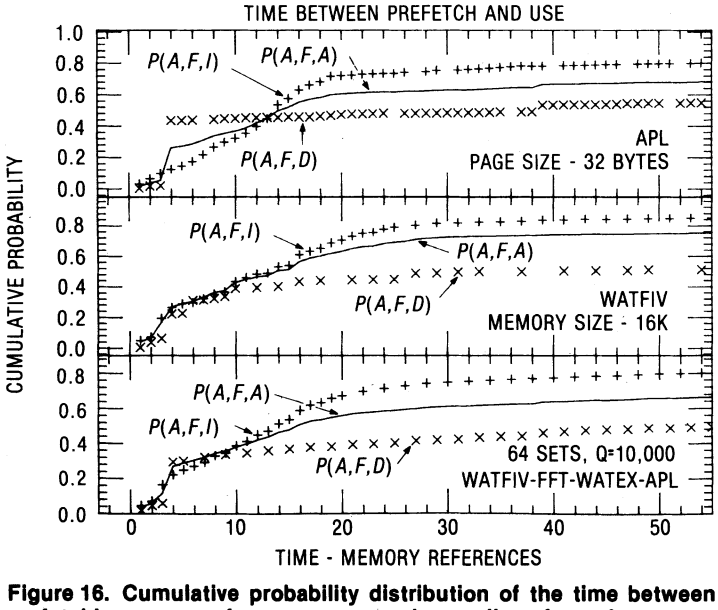


Figure 16. Cumulative probability distribution of the time between prefetching a page from memory and actually referencing a prefetched page. Results are shown for the APL program trace, the Watfiv program trace, and multiprogramming using all four program traces.

Prefetching improves performance at low cost

Prefetching pages of information stored in a cache memory can improve the performance of large computer systems significantly if the prefetching strategy is properly implemented. Our experiments indicate that if prefetching is used with very small page sizes, such as 32 or 64 bytes, it can be effective, whereas prefetching large pages (e.g., 1024-4096 bytes) is more likely to degrade performance than enhance it. Since pages of that small size are typically used in cache memories, it appears that the cache is the level within the memory hierarchy at which to implement prefetching. Although the existing cache-memory implementations we studied would have to be modified to accommodate an efficient prefetching strategy, the potential improvements in CPU speed—on the order of 10 percent to 25 percent—would seem to be worth the expense of modification. ■

Acknowledgment

This research was partially supported by the National Science Foundation under Grant MCS75-06768. Computer time was provided by the Energy Research and Development Administration under Contract E(04-3)515 and by the Department of Energy under Contract EY-76-C-03-0515.

References

1. A. J. Smith, *Sequentiality and Prefetching in Data Base Systems,* IBM Research Report RJ 1743, Mar. 1976. Also in *ACM Transactions on Data Base Systems,* Sept. 1978, pp. 223-247.
2. P. J. Denning, "On Modeling Program Behavior," *AFIPS Conf. Proc.,* Vol. 40, 1972 SJCC, pp. 937-944.
3. B. L. Peuto and L. J. Shustek, "An Instruction Timing Model of CPU Performance," *Proc. Fourth Annual Symp. Computer Arch.,* College Park, Md., Mar. 1977, pp. 165-178.
4. A. J. Smith, "A Locality Model for Disk Reference Patterns," *Digest of Papers, COMPCON 75 Spring,* San Francisco, Calif., pp. 109-113.
5. A. J. Smith, "Analysis of a Locality Model for Disk Reference Paterns," *Proc. Second Conference on Information Sciences and Systems,* The Johns Hopkins University, Baltimore, Md., Apr. 1976, pp. 593-601.
6. A. J. Smith, "On the Effectiveness of Multiple Arm and Buffered Disks," *Proc. Fifth Annual Symp. Computer Arch.,* Palo Alto, Calif., Apr. 1978, pp. 242-247.
7. M. Joseph, "An Analysis of Paging and Program Behavior," *The Computer Journal,* Vol. 13, No. 1, Feb. 1970, pp. 48-54.
8. J. L. Baer and G. R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems," *IEEE Trans. Soft. Eng.,* Vol. SE-2, No. 1, Mar. 1976, pp. 54-62.

9. D. E. Gold and D. J. Kuck, "A Model for Masking Rotational Latency by Dynamic Disk Allocation," *CACM,* Vol. 17, No. 5, May 1974, pp. 278-288.

10. K. S. Trivedi, *Prepaging and Applications to the STAR-100 Computer,* Report 76-28, ICASE, NASA Langley Research Center, Hampton, Va., Aug. 1976, republished in *Proc. Symp. High Performance Computer and Algorithm Organization,* Champaign, Ill., Apr. 1977.

11. K. S. Trivedi, "Prepaging and Applications to Array Algorithms," *IEEE Trans. Computers,* Vol. C-26, No. 9, Sept. 1976, pp. 915-921.

12. K. S. Trivedi, *An Analysis of Prepaging,* Computer Science Report CS-1977-7, Duke University, Durham, N.C., Aug. 1977.

13. K. S. Trivedi, "On the Paging Performance of Array Algorithms," *IEEE Trans. Computers,* Vol. C-27, No. 10, Oct. 1977, pp. 938-947.

14. O. Spaniol, "Demand Prepaging Algorithms Based on a Model of Locality of Programs," *Workshop on Computer Architectures and Networks,* IRIA, Rocquencourt, France, Aug. 1974.

15. U. W. Pooch, "A Dynamic Clustering Strategy in a Demand Paging Environment," *Proc. Symp. Simulation of Computer Systems,* National Bureau of Standards, Boulder, Colo., Aug. 1976, pp. 11-22.

16. B. T. Bennett and P. A. Franaczek, "Cache Memory with Prefetching of Data by Priority," *IBM Technical Disclosure Bulletin,* Vol. 18, No. 12, May 1976, pp. 4231-4232.

17. C. J. Conti, "Concepts for Buffer Storage," *IEEE Computer Group News,* Mar. 1969, pp. 9-13.

18. R. M. Meade, "Design Approaches for Cache Memory Control," *Computer Design,* Vol. 10, No. 1, Jan. 1971, pp. 87-93.

19. A. J. Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Trans. Soft. Eng.,* Vol. SE-4, No. 2, Mar. 1978, pp. 121-130.

20. D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM/360 Model 91: Machine Philosophy and Instruction Handling," *IBM J. Res. and Devel.,* Jan. 1967, pp. 8-24.

21. T. A. Enger, "Paged Control Store Prefetch Mechanism," *IBM Technical Disclosure Bulletin,* Vol. 16, No. 7, Dec. 1973, pp. 2140-2141.

22. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, " Evaluation Techniques for Storage Hierarchies," *IBM Sys. J.,* Vol. 9, No. 2, pp. 78-117.

23. Niklaus Ragaz and Juan Rodriguez-Rosell, *Empirical Studies of Storage Management in a Data Base System,* IBM Research Report RJ 1834, Oct. 1976.

24. *System/370 Principles of Operation,* GA22-7000, IBM Corp., Armonk, N.Y., 1975.

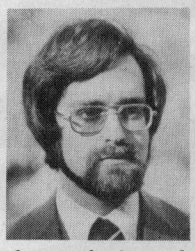
25. *470V/6 Machine Reference Manual,* Amdahl Corp., 1976.

26. A. J. Smith, "Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write-Through," to appear, *JACM,* Jan. 1979.

27. R. Tobias, Amdahl Corporation, private communication, 1977.

28. *IBM System/360 and System/370 Model 195 Functional Characteristics,* GA22-6943-2, IBM Systems Development Division, Poughkeepsie, N.Y., 1971.

29. *System/370 Model 168 Theory of Operation/Diagrams Manual,* Vol. 1, Introduction (SY22-6931-1), Vol. 4, Processor Storage Control Function (SY22-6934-1), IBM Systems Products Division, Poughkeepsie, N.Y., 1975.



Alan Jay Smith is an assistant professor in the Computer Science Division of the Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California, Berkeley, a position he has held since 1974. He also holds a joint appointment at the Lawrence Berkeley Laboratory. His research interests include the analysis and modeling of computer systems and devices, operating systems, computer architecture, and data compression.

He received the BSEE from the Massachusetts Institute of Technology and the MS and PhD in computer science from Stanford University.

Smith is a member of the IEEE, the ACM, the Society for Industrial and Applied Mathematics, Eta Kappa Nu, Tau Beta Pi, and Sigma Xi.