CPU Cache Prefetching: Timing Evaluation of Hardware Implementations

John Tse and Alan Jay Smith, *Fellow*, *IEEE*

Abstract—Prefetching into CPU caches has long been known to be effective in reducing the cache miss ratio, but known implementations of prefetching have been unsuccessful in improving CPU performance. The reasons for this are that prefetches interfere with normal cache operations by making cache address and data ports busy, the memory bus busy, the memory banks busy, and by not necessarily being complete by the time that the prefetched data is actually referenced. In this paper, we present extensive quantitative results of a detailed cycle-by-cycle trace-driven simulation of a uniprocessor memory system in which we vary most of the relevant parameters in order to determine when and if hardware prefetching is useful. We find that, in order for prefetching to actually improve performance, the address array needs to be double ported and the data array needs to either be double ported or fully buffered. It is also very helpful for the bus to be very wide (e.g., 16 bytes) for bus transactions to be split and for main memory to be interleaved. Under the best circumstances, i.e., with a significant investment in extra hardware, prefetching can significantly improve performance. For implementations without adequate hardware, prefetching often decreases performance.

Index Terms—Cache memory, prefetching, timing model, cache prefetching, CPU architecture, memory system design, CPU cache memory.

1 Introduction

It is well known that delays in accessing CPU memory are one of the major factors in limiting CPU performance [28]. Cache memories are the principal technique used to improve CPU memory system performance, but cache misses continue to degrade performance. Cache studies have considered various factors, such as line (block) size, associativity, cache size, prefetching algorithms, etc. In this paper, we study the effectiveness of hardware prefetching when it interacts with other cache design parameters.

Prefetching has long been known to significantly decrease cache miss ratios. This was shown in [25], with additional results for various algorithms and workloads presented in [28], [29]. Most of these studies, however, have neglected timing effects, i.e., even though the miss ratio has decreased, does the machine get faster? (Reference [25] did look at the issue of whether the needed block would arrive in time.) Some more recent studies which have considered timing include [2], [3], [13], and [19]. In this paper, we present the results of a detailed cycle-by-cycle trace-driven simulation of a memory system in which we vary all relevant parameters in order to determine when and if prefetching is useful. Our study differs from other related research in that we are concentrating on uniprocessor design, in the detail level of our study, and in the range of architectural parameters studied.

• *J. Tse is with Altera Corp., 1745 Wesley Ave., El Cerrito, CA 94530. E-mail: johnts@altera.com.*
• *A.J. Smith is with the Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720-1776. E-mail: smith@cs.berkeley.edu.*

Manuscript received June 1995; revised 20 May 1997.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106387.

1.1. Previous Research and Prefetch Algorithms

CPU cache prefetching involves fetching a block from main memory into the CPU cache when the block has not been referenced in the expectation that it will be referenced soon. Hardware cache prefetching is specifically concerned with prefetching algorithms implemented solely by dedicated hardware and without software support; we consider only hardware prefetching in this paper. Prefetching algorithms have been concerned with two issues: which block to prefetch and when to prefetch. The simplest candidate to prefetch is the next sequential block after the one most recently referenced. Conceptually, this makes sense because instructions are fetched sequentially (except for branches) and data is often referenced sequentially (when in arrays) or, at least, locally (when the compiler allocates related variables in contiguous locations, as when they are stored together in a stack).

One simple prefetch algorithm is called *always prefetch* [28]. With this algorithm, every time there is a reference to block i, the cache is examined for block i + 1 (i.e., the next sequential block, in terms of ascending memory addresses). If block i + 1 is absent from the cache, it is prefetched. A variation which requires fewer prefetches and prefetch lookups (i.e., look into the cache to see if the block is there) is called *prefetch on misses*, which prefetches the next sequential cache block if and only if the access to the current cache block is a miss. A more complicated scheme, known as *tagged prefetch* [12], [28], keeps the number of prefetch lookups low while issuing more prefetches than prefetch on misses. In this case, each cache block has a single bit, called the tag, which is set to zero whenever the block does not reside in the cache. When a block is referenced by the processor, its tag will be set to one. A block brought into the cache by a prefetch, however, retains its tag of zero. Whenever a tag changes from zero to one, a prefetch is initiated for the next sequential cache block. This is similar to always prefetching, but it avoids repeated cache lookups and does not again prefetch a line which was prefetched and later replaced without having been referenced.

More sophisticated schemes like *threaded prefetching* [18] and *bidirectional prefetching* [35] have also been proposed. In threaded prefetching, cache block i has associated with it a list of pointers known as threads. Each thread points to a cache block which is most likely to be referenced after block i has just been accessed. Suppose that the processor is accessing cache block i in cycle T. If block j is referenced in cycle T + 1, a new thread which contains the address of block j will be attached to block i in cycle T + 1. The new thread can be stored together with block i in the instruction or data cache, or it can be stored in a separate cache. As soon as block i is reaccessed by the processor, all threads associated with block i will trigger the prefetching of block j and other cache blocks the threads point to. Bidirectional prefetching attempts to capture the forward and backward accessing behavior found in data caches. If the current and the previous sequential cache blocks are found in the cache, the next sequential cache block will be prefetched; otherwise, if the current and the next sequential cache blocks are found, the previous sequential cache block will be prefetched.

When a prefetching strategy detects a potential miss, most will issue a prefetch for a fixed number of cache blocks. Since the prefetching efficiency varies during the execution of a program, it may be advantageous to prefetch a variable number of cache blocks. Such an adaptive sequential prefetching scheme for shared memory multiprocessors is proposed in [9], [6]. Each cache block has a counter which records the number of times prefetches on this block are serviced by the main memory. By inspecting these counters, one will be able to measure the effectiveness of prefetching dynamically and prefetch an appropriate number of cache blocks accordingly.

Cache prefetching strategies can also be used in special cache organizations, like vector caches. Data prefetching strategies for vector cache memories are examined in [9]. A stride-prefetch strategy is proposed which takes advantage of the vector stride information specified in a vector instruction that causes a cache miss. If element i in the vector data causes the miss, prefetches for blocks i, i + stride, i + 2 * stride, ... , i + p * stride will be issued, where p is the number of sequential blocks to be prefetched. Results show that stride-prefetch strategy performs better for this workload than the always prefetching strategy and no prefetching.

Two level cache memory systems have been shown to be an effective mean to enhance system performance [1], [24]. Methods to evaluate the performance of cache prefetching in second level caches are studied in [31]. Results show that miss ratios alone are insufficient to evaluate the performance of cache prefetching in higher level caches. By using a detailed timing-based trace-driven simulation model, researchers show that hit-rate-only analysis may be extremely optimistic in predicting the benefits of cache prefetching. Our paper uses a similar approach, but explores a larger design space and in greater detail.

One of the disadvantages of cache prefetching is the unavoidable increase in memory traffic because of prefetches which are never referenced. The limitations of cache prefetching on a bus-based multiprocessor system are investigated in [34]. Results show that, when bus bandwidth is the bottleneck, prefetching will not improve performance, even when it reduces the demand miss ratio.

Other recent work on prefetching can also be found in [4], [14], [21], and [7]. In [14], a prefetching scheme which combines hardware and software features is presented and analyzed. Reference [21] concentrates on prefetching in shared memory multiprocessors running scientific (vector) workloads. Reference [4] likewise considers prefetching in a vector environment with constant stride. Reference [7] looks at prefetching a variable number of blocks (sequentially ahead of the current one) as a function of previous behavior.

Another disadvantage of cache prefetching is that useless prefetches may pollute cache contents (“memory pollution”) by displacing useful cache blocks from the cache and, thus, cause new cache misses which would not have happened had there been no prefetching. This effect is analyzed for disk caches in [27]. Memory pollution is most hazardous to performance when cache sizes are small and block (line) sizes are large.

1.2. Research Issues

Cache prefetching has been implemented in at least two machines, the Amdahl 470V/6 [25] and the Fairchild/Intergraph Clipper [5], [16]. In both cases, performance failed to improve, although, for the latter machine, it was at least in part due to a clear implementation flaw. More generally, however, it is not clear that a decrease in the miss ratio attributable to prefetching will actually lead to an improvement in CPU performance. Even when the miss ratio decreases, prefetching can degrade performance by:

1) making the cache address tag arrays busy due to prefetch lookups;
2) making the cache data arrays busy due to prefetch loads and replacements;
3) making the memory bus busy due to prefetch address transfers and data fetches; and
4) making the memory system busy due to prefetch fetches and replacements.

The issue we consider in this paper is the effect of prefetching on performance, which we evaluate using a detailed cycle by cycle simulator of the CPU memory system. We consider the effect of

1) various prefetch algorithms,
2) while varying various cache parameters, such as cache size, block size, and cache associativity, and
3) also while adding hardware resources, such as double ported cache tags, cache data arrays, a wider memory bus, an interleaved memory system, and buffering for loads into the cache.

The rest of this paper is organized as follows: Section 2 gives a detailed description of our architectural model, while Section 3 describes the methodology we used; Section 4 gives a performance overview of cache prefetching with our baseline system; Section 5 investigates the impacts of

TABLE 1.1
TERMINOLOGY USED IN THIS PAPER

Term	Definition
Cache Miss	An event in which the address referenced by the processor is not found in the cache and the address, if issued by the prefetching unit, has not been sent to the main memory yet.
Partial Cache Miss	An event in which the address referenced by the processor is not found in the cache but the address has been issued by the prefetching unit and sent to the main memory already.
True Miss Ratio	The ratio of the total number of cache misses to the total number of references to the cache made by the processor.
Partial Miss Ratio	The ratio of the total number of partial cache misses to the total number of references to the cache made by the processor.
Total Miss Ratio	Sum of true miss ratio and partial miss ratio.
Issued Prefetch	A prefetch which is sent to the prefetch address buffer by the prefetching unit.
Lifetime of a Prefetch	It starts when a prefetch address is sent to main memory and ends when it is replaced from the cache or referenced by the processor.
Useful Prefetch	An issued prefetch whose address is referenced by the processor during its lifetime.
Useless Prefetch	An issued prefetch whose address is never referenced by the processor during its lifetime.
Aborted Prefetch	An issued prefetch which is discarded in the prefetch address buffer because: (1) the prefetch address buffer runs out of room; or (2) the prefetching address has already been referenced by the processor after the prefetch was issued but the prefetching address has not been sent to the main memory yet.
Prefetch Ratio	The ratio of the total number of issued prefetches to the total number of cache references.
Success Ratio	The ratio of the total number of useful prefetches to the total number of issued prefetches.
Global Success Ratio	The fraction of cache misses which are avoided or partially avoided (i.e. a partial cache miss). See section 4.2 for more details.

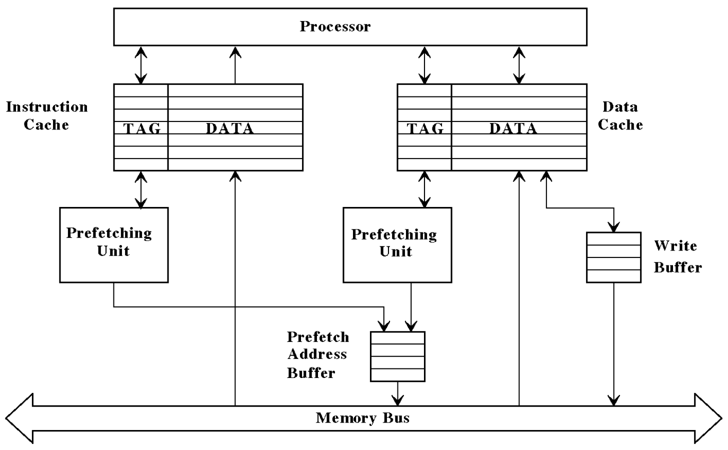


Fig. 2.1. Overview of the architectural model.

several essential system resources on the performance of cache prefetching strategies; Section 6 compares their performance on the best system for prefetching with that on the baseline system; finally, Section 7 concludes the paper. Table 1.1 collects and defines for convenient reference and lookup various terms used throughout this paper.

2 ARCHITECTURAL MODEL

This section gives a detailed description of the architectural model which our cycle-by-cycle simulator is based on. Fig. 2.1 shows an overview of the model. In each case, we indicate the default (baseline, standard) design and the various options.

2.1 Processor

Our CPU model is a RISC type processor similar to the MIPS R3000. There are five stages in the pipeline: instruction fetch, instruction decode, read or write to memory, ALU computation, and register file update. The processor fetches an instruction from the instruction cache every clock cycle. Whenever there is a cache miss, the processor halts and waits for the required cache block to be filled. Execution resumes as soon as the required word, rather than the entire cache block, has arrived. The assumption is that all instructions take one cycle; no special cases (e.g., floating point operations, branches) are considered.

2.2 Caches

We use a split instruction and data cache. Both the instruction and data caches can have either single or double ported tag arrays and data arrays. When an array is double ported, one port is used by the processor and the other by the prefetcher, so that prefetches and processor accesses (to tags, instructions, or data) can proceed in parallel without interference. In one case, we also simulate a buffered data array and compare that with a double ported data array.

In the default (baseline) system, all caches are eight-way set associative and virtually addressed with a total size of 64KB and a block size of 64 bytes, unless otherwise specified. This is to avoid conflict misses and translation issues. The data cache uses a write back and write allocate policy.

A read hit takes one clock cycle, while a write hit takes two (lookup and then modify). The set associativity, cache size and block size are varied in some of our studies.

2.3 Write Buffer

The data cache sends dirty cache blocks back to the main memory via the write buffer. The write buffer is a four entry first-in-first-out (FIFO) queue. It waits until the memory bus is idle before it puts the oldest entry onto the bus. If the write buffer is full, it cannot accept new entries and an attempt to replace a dirty block causes the processor to stall until the write buffer has a free slot. Whenever there is a data cache miss, the write buffer is searched for the missing cache block. If the block is found, it is copied from the write buffer back to the data cache within the same clock cycle.

2.4 Prefetching Units

The two prefetch units, one for each cache, are responsible for issuing new prefetch requests to the main memory. During each clock cycle, each prefetch unit receives information like cache misses, cache hits, instruction types, and branch target addresses from the processor and the caches. Based on this information, it decides whether to issue a new prefetch request or not. If it does, the prefetch address is looked up in the corresponding cache. The request is issued in the next clock cycle if the data is not found in the cache.

Issued requests from both prefetch units are not sent directly to the memory bus, though, but to a prefetch address buffer, each of which is organized as a FIFO queue with 16 entries. The oldest entry is sent to the memory bus only when the bus is free. If the buffer is full when a newly issued request arrives, the oldest entry is discarded from the buffer to make room for the new one. Whenever there is a cache miss, the address of the missing cache block is compared against every entry of the buffer. Any entry which matches the address represents a failed prefetch (because it is issued too late) and is discarded without being issued. (It might be interesting, in future research, to see if LIFO queuing makes any difference.)

2.5 Memory Bus

The memory bus selectively supports both split and nonsplit bus transactions [20]. A bus transaction includes two parts: Sending the address and receiving or sending the data. In a split bus transaction, the memory bus is idle between sending the address and receiving the data and other transactions are free to use it. In a nonsplit bus transaction, however, a transaction holds the memory bus until it receives its data. In the default system, the memory bus operates with nonsplit transactions and is four bytes wide, unless specified otherwise.

Since multiple bus users may compete for the memory bus at the same time, a bus arbitrator is needed to resolve these conflicts. In our model, a fixed priority scheme is used to arbitrate the bus. The priority in descending order is as follows:

1) write back by the write buffer when it is full;
2) a cache miss;
3) returning data from the main memory;
4) write back by the write buffer when it is not full;
5) a new prefetch request.

2.6 Main Memory

The main memory may contain up to 16 banks, each of which is four bytes wide and is interleaved on the low order address bits. Each bank has an input queue to buffer requests while the bank is busy and an output queue to buffer returning data when the bus is busy. Because requests to distinct memory banks can be served out of order, a collating buffer holds the returning data until they can be returned to the caches in order. The default (baseline) memory delay is set to be 16 processor clock cycles, while the default number of memory banks is four. In order to simplify the memory hierarchy, we assume that the main memory always contains all referenced pages, so that there can be no page faults.

Unless otherwise indicated, all tables and figures show results that are the average over all workloads. Breakdowns by individual workloads can be found in most cases in [33].

3 Methodology

3.1 Trace-Driven Simulation

Trace-driven simulation was used for the studies here for the usual well-known reasons; see [32] (among many other papers) for a discussion of the advantages of the technique. We collected address traces on a DECstation 5000, running version 4.2A of the DEC Ultrix operating system. The benchmark programs were compiled using version 2.3.3 of the gcc compiler for C programs and version 2.1 of the f77 compiler for FORTRAN programs, both with the highest level of optimization. The MIPS pixie tool [8] then inserted extra monitoring codes into the compiled binaries so that, when we ran the pixified executables, address traces would be collected in specified files.

We ran our simulations on a DEC 3000, running version 1.2A of the OSF/1 operating system. We developed a trace-driven simulator which read instruction and data addresses from the address trace, simulated the target memory hierarchical model described earlier on a cycle by cycle basis, and collected and reported the simulation results.

3.2 The Workload

In order to have a substantial and, hopefully, representative workload sample, we selected 25 commonly used real programs from five different workload categories: computer-aided design tools (CAD), compiler-related tools (COMP), floating point intensive applications (FP), text processing programs (TEXT), and UNIX utilities (UNIX). We also chose existing inputs of reasonable size and complexity, whenever possible, to run these programs on. Table 3.1 describes these programs and shows the breakdowns of memory references in the final traces.

Each group trace consisted of five different program traces in the same category and they were interleaved according to a round robin based preemptive scheduler. Each address was tagged with a process ID; the cache was not flushed when the address space changed. Whenever a process executed a system call, it was blocked and put into a sleep queue for a random number of cycles which followed a uniform distribution in the range of 25,000 to 50,000 cycles. Control was then switched to the next process in the

TABLE 3.1
CONTENTS OF THE CAD, COMP, FP, TEXT, AND UNIX ADDRESS TRACE WORKLOADS

CAD Traces	Trace Description	Total Ref.	Instruction Fetches	Memory Reads	Memory Writes	Program Size	Memory Touched	Context Switches
eqntott	equations to truth tables	6,709K	5,353K	1,201K	155K	53KB	1,030KB	21
espresso	2 level logic minimizer	8,169K	6,382K	1,350K	438K	246KB	216KB	21
irsim	circuit simulator	8,235K	5,708K	1,601K	927K	679KB	206KB	21
misII	logic synthesizer	8,433K	5,997K	1,556K	880K	2,230KB	832KB	21
spice	circuit simulator	8,464K	6,605K	1,342K	516K	808KB	251KB	21
	Total:	40,010K	30,044K	7,051K	2,915K	4,016KB	2,533KB	106

COMP Traces	Trace Description	Total Ref.	Instruction Fetches	Memory Reads	Memory Writes	Program Size	Memory Touched	Context Switches
as	MIPS assembler	9,651K	7,492K	1,506K	654K	102KB	567KB	26
ccom	front end of C compiler	9,198K	7,099K	1,354K	746K	266KB	1,109KB	25
cpp	C preprocessor	3,940K	3,296K	509K	135K	49KB	833KB	25
fcom	front end of f77 compiler	7,073K	5,503K	977K	593K	398KB	1,071KB	25
ld	MIPS linker	10,147K	7,423K	1,631K	1,093K	188KB	326KB	25
	Total:	40,010K	30,813K	5,977K	3,220K	1,004KB	3,907KB	127

FP Traces	Trace Description	Total Ref.	Instruction Fetches	Memory Reads	Memory Writes	Program Size	Memory Touched	Context Switches
doduc	hydraulic simulator	7,769K	5,577K	1,651K	539K	387KB	538KB	18
linpack	matrix operations	8,005K	5,622K	1,548K	835K	60KB	342KB	18
matrix300	saxpy on matrices	8,135K	5,233K	1,935K	967K	306KB	2,956KB	18
nasa7	7 scientific kernels	6,984K	5,633K	1,210K	140K	185KB	647KB	18
tomcatv	mesh generator	9,118K	5,679K	2,627K	812K	306KB	10,452KB	17
	Total:	40,010K	27,744K	8,972K	3,294K	1,244KB	14,935KB	90

TEXT Traces	Trace Description	Total Ref.	Instruction Fetches	Memory Reads	Memory Writes	Program Size	Memory Touched	Context Switches
enscript	text processor	8,375K	6,203K	1,332K	841K	127KB	121KB	20
jove	text editor	7,735K	5,764K	1,402K	568K	242KB	381KB	21
nroff	text processor	7,524K	6,023K	1,046K	454K	90KB	287KB	20
tbl	formats tables	7,946K	6,144K	1,158K	644K	120KB	377KB	20
vi	text editor	8,431K	6,403K	1,302K	726K	172KB	319KB	21
	Total:	40,010K	30,538K	6,239K	3,232K	751KB	1,485KB	103

UNIX Traces	Trace Description	Total Ref.	Instruction Fetches	Memory Reads	Memory Writes	Program Size	Memory Touched	Context Switches
awk	language processor	8,230K	6,121K	1,288K	820K	111KB	68KB	21
compress	compact files	8,458K	6,639K	1,160K	658K	33KB	1,660KB	21
grep	search for patterns	7,623K	6,229K	9,81K	413K	25KB	63KB	20
sed	text stream editor	7,889K	5,986K	1,219K	684K	45KB	76KB	20
tar	create file archives	7,808K	6,334K	1,012K	462K	372KB	128KB	20
	Total:	40,009K	31,311K	5,661K	3,037K	585KB	1,995KB	103

ready queue. When the sleeping process woke up, it was put on the tail of the ready queue. In addition, no process was allowed to run more than 400,000 cycles continuously.

In order to minimize start-up effects, the first 70 million memory references of each group trace were not included in the final trace used; instead, they were used to warm-start the instruction and data caches. In order to save space and simulation time, we adopted the scheme described in [22]; we kept track of references to unique addresses and the last time they were referenced. The one million most recently used unique addresses were added to the front of the final traces, starting with the least recently used address and ending with the most recently used one. We also distinguished reads from writes to the same addresses so that the correct cache blocks would be dirty at the warm-start boundary.

Each final group trace contained roughly 40 million memory references other than those used to warm-start the caches. These traces were originally stored in an ASCII address trace format known as the *dinero* [15] format. We used a trace compacting scheme similar to *mache* [23], which basically converts each ASCII address into a one to four byte binary integer. We extended *mache* by keeping a counter with each instruction address. The counter informed the simulator of the number of sequential instructions to be executed before the next data read, data write or instruction branch. By keeping track of the program counter, we could now delete trace entries for those sequential instructions. The size of the final traces compacted by our scheme was about two to five times smaller than that processed by mache, and our simulator ran as much as 20 percent faster.

TABLE 4.1.1
DEFAULT SYSTEM CONFIGURATION FOR THE BASELINE SYSTEM

System Parameters	Default Value	System Parameters	Default Value
Cache tag array	double ported	Bus transaction	non-split
Cache data array	single ported	Bus width	4 bytes
Cache size	64K bytes	Memory latency	16 CPU cycles
Cache block size	64 bytes	Number of memory banks	4
Cache associativity	8	Memory bank width	4 bytes
Cache type	split	Prefetch lookahead distance	1 cache block

In our opinion, for the purposes of this study, the trace lengths were more than sufficient. The purpose of this study was not to obtain absolute miss ratios for large caches, but to explore the behavior of prefetching algorithms. Such behavior should not be sensitive to the length of the trace, but should react primarily to the reference patterns over short lengths of the trace.

4 THE BASELINE SYSTEM

4.1 Evaluation Metric

Table 4.1.1 describes the default system settings in the baseline system. These values were chosen according to current technology and common design practice, and/or to yield clear results. (Eight-way associativity is used to minimize conflict misses.) Unless other values are explicitly stated, they will be used throughout the rest of the paper.

The main metric we used in our evaluation is known as "(Variable) Cycles Per Instruction contributed by Memory accesses", or MCPI for short [3]. Since we assume that instruction pipelining is perfect and that the processor is capable of executing one instruction per cycle, memory access penalty becomes the sole variable contributor to CPI. MCPI is given by the following equation:

$$MCPI = \frac{total\ memory\ access\ penalty - processor\ stalls\ due\ to\ cache\ write\ hits}{total\ number\ of\ instructions\ executed}.$$

Our definition of MCPI differs from [Chen93] in that we exclude processor stalls due to cache write hits from our calculation, since write hit stalls are unavoidable and insensitive to cache prefetching. This is a constant additive amount and does not affect our results except to more clearly isolate the parameter of interest.

We prefer MCPI to other evaluation metrics, like cache miss ratios and effective memory access time, for two reasons:

1) MCPI covers every aspect of performance which can be improved or degraded by a cache prefetching strategy. Not only does it reflect the reduction in cache miss ratios, but it also includes the effects of heavier data bus traffic.
2) MCPI excludes aspects of performance which cannot be affected by a cache prefetching strategy, for example, the efficiency of the instruction pipelining.

To explain it another way, we note that <cycles per instruction> is the sum of <execution cycles, including pipeline interlocks, etc.> + <constant memory system delays> + <variable memory system delays>. Constant memory system delays are due to writes—every write takes two cycles, whereas all other instructions take only one. "Variable memory system delays" is the same as MCPI—it includes all memory system delays except the extra cycle for a write. In a perfect memory system (infinite size, one cycle access), "variable memory system delays" would be zero.

Because MCPI is a function not only of the cache management algorithms, including prefetching, but also of the cache and system design parameters, its absolute value cannot be used as an evaluation metric. Specifically, we are interested in whether performance gets better or worse due to prefetching, not in the level of performance in the base system. Accordingly, we use the relative MCPI, which compares the performance of a cache prefetching strategy with that when no prefetching is used in the same system. A cache prefetching strategy improves performance only when its relative MCPI is smaller than one. If it is greater than one, the strategy actually degrades performance. Relative MCPI is given as follows:

$$Relative\ MCPI = \frac{MCPI\ when\ a\ cache\ prefetching\ strategy\ is\ used}{MCPI\ when\ no\ prefetching\ strategy\ is\ used\ in\ the\ same\ system}.$$

4.2 Performance of Cache Prefetching in the Baseline System

In this section, we evaluate the performance of cache prefetching strategies in the baseline system. Table 4.2.1 shows the relative and absolute MCPIs when different prefetching strategies are used. The results look discouraging: Most strategies perform worse than if there were no prefetching at all. The average relative MCPI for all strategies is 1.1836. In other words, prefetching worsens MCPI by more than 18 percent, on average, in the baseline system.

But, all prefetching strategies do improve performance for the address trace fp. fp consists of programs like linpack, matrix300, and tomcatv, which work on data structures representing matrices and meshes. Since these data structures are large in size and highly sequential in nature, a prefetching strategy is able to predict future address references accurately. This explains why aggressive, yet simple, strategies, like always and tag, can reduce MCPI by as much as 12 percent.

The address trace unix, on the other hand, gives the worst results when cache prefetching strategies are used. On average, MCPI increases by more than 46 percent. The unix trace consists of small programs that work on many different data sets. The memory accessing pattern is often nonsequential and even backward, especially in the data cache. Most forward predicting strategies, therefore, cannot

TABLE 4.2.1
RELATIVE AND ABSOLUTE MCPI FOR THE BASELINE SYSTEM

Cache Prefetching Strategies	Relative MCPI						Absolute MCPI					
	Trace Name					Trace avg.	Trace Name					Trace avg.
	cad	comp	fp	text	unix		cad	comp	fp	text	unix	
none	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*0.0863*	*0.0983*	*0.1759*	*0.0306*	*0.1460*	*0.1074*
always	1.5914	1.2924	0.8821	1.1770	1.7009	1.2943	0.1373	0.1271	0.1552	0.0360	0.2484	0.1408
miss	1.2686	1.1505	0.9523	1.1306	1.6159	1.2049	0.1094	0.1131	0.1675	0.0346	0.2360	0.1321
tag	1.3774	1.1561	0.8793	1.1495	1.6695	1.2186	0.1188	0.1137	0.1547	0.0352	0.2438	0.1332
bi-dir	1.4329	1.1979	0.8885	1.1712	1.0818	1.1408	0.1236	0.1178	0.1563	0.0358	0.1580	0.1183
thread	1.0004	0.9968	0.9979	0.9933	1.3515	1.0596	0.0863	0.0980	0.1755	0.0304	0.1974	0.1175
prefetch avg	1.3186	1.1547	0.9188	1.1222	1.4633	1.1836	0.1151	0.1139	0.1618	0.0344	0.2167	0.1284

TABLE 4.2.2
STALL BREAKDOWNS IN PERCENTAGE FOR THE BASELINE SYSTEM

CPU Stall Reasons	Cache Prefetching Strategies						Prefetch Average
	none	always	tag	miss	bidir	thread	
cache misses	*99.27*	51.56	66.88	53.69	61.07	82.74	63.19
prefetch (cache)	*0.00*	33.97	19.68	32.28	30.38	8.23	24.91
prefetch (bus)	*0.00*	13.94	12.86	13.47	7.98	8.38	11.33
prefetch (mem)	*0.00*	0.00	0.00	0.00	0.00	0.00	0.00
write buffer	*0.72*	0.52	0.58	0.56	0.57	0.65	0.58

predict future references correctly. Bi-dir, which is designed to predict both forward and backward accessing patterns therefore performs significantly better than other strategies.

From the drastic difference in performance between fp and unix, one can deduce that a cache prefetching strategy is very sensitive to the type of programs the processor is running. In general, it favors a program which references memory sequentially, but works poorly for one which accesses data nonsequentially.

Table 4.2.2 shows the composition of processor stalls: Altogether, there are five reasons why the processor has to stall:

(1) Cache Misses

The processor is stalled because the memory address it is referencing is not found in the cache and it must wait until the required memory blocks are sent from the main memory. Note that this category does not include stalls due to conflicts with active prefetches. It only includes the raw memory latency of demand misses.

Although all prefetching strategies are able to reduce stalls due to cache misses, they introduce three new categories of stalls, described below, which reduce their effectiveness:

(2) Prefetch (Cache) Stalls

The processor is stalled because it cannot access the data port of a cache which is being used by an active prefetch to load a block into the cache. In our architectural model, a regular memory reference by the processor has higher priority to access the data port of a cache than a prefetch. However, if a prefetch has already acquired the data port, the processor has to wait until the prefetch finishes its access.

In the baseline system, this category represents a significant fraction of total processor stalls. Conflicts over cache data ports between prefetches and regular processor references seriously affect the performance of cache prefetching. We will study this issue in greater detail in Section 5.2.

(3) Prefetch (Bus) Stalls

A demand miss is delayed because it cannot access the data bus. As in the case of cache data ports, a demand miss has higher priority to access the data bus than a prefetch. But if a prefetch is already using the bus, a demand miss has to wait until the bus is free again.

Conflicts over the data bus represent the second biggest drawback of cache prefetching. Because the data bus only supports nonsplit transactions in the baseline system, a prefetch will lock the bus until the prefetched block returns from the main memory. A split transaction data bus would have reduced this contention. We will study this issue more carefully in Section 5.6.

(4) Prefetch (Mem) Stalls

A demand miss is delayed because the memory bank it needs to access is being used by an active prefetch. Here, both demand misses and prefetches have the same priority in accessing the memory banks because the main memory does not distinguish between them.

Since a demand miss or a prefetch will not release the data bus until it finishes accessing the memory bank (in the baseline system), there can only be one access to the main memory at any time. This explains why this category is always zero in Table 4.2.2. When the effect of split transaction bus is considered in Section 5.6, multiple accesses to the main memory become possible and this category of stalls will become significant.

(5) Write Buffer Stalls

The processor is stalled because the write buffer is full. In Table 4.2.2, this represents less than 1 percent of total stalls. This frequency of this kind of stall is insignificant because a four entry deep write buffer is highly effective in buffering dirty cache blocks back to the main memory [26].

TABLE 4.2.3
INSTRUCTION AND DATA CACHE MISS RATIOS

Cache Type	Cache Miss Ratios	Cache Prefetching Strategies						Prefetch Average
		none	always	miss	tag	bi-dir	thread	
Instr. Cache	True Miss Ratio	*0.0011*	0.0008	0.0009	0.0008	0.0008	0.0007	0.0008
	Partial Miss Ratio	*0.0000*	0.0002	0.0001	0.0002	0.0002	0.0002	0.0002
	Total Miss Ratio	*0.0011*	0.0010	0.0010	0.0010	0.0010	0.0009	0.0010
Data Cache	True Miss Ratio	*0.0109*	0.0065	0.0085	0.0065	0.0065	0.0101	0.0076
	Partial Miss Ratio	*0.0000*	0.0002	0.0002	0.0001	0.0001	0.0002	0.0002
	Total Miss Ratio	*0.0109*	0.0067	0.0087	0.0066	0.0066	0.0103	0.0078

TABLE 4.2.4
SUCCESS RATIOS AND GLOBAL SUCCESS RATIOS

Cache Type	Prefetch Success Ratios	Cache Prefetching Strategies					Prefetch Average
		always	miss	tag	bi-dir	thread	
Instr. Cache	Success Ratio	0.6151	0.6638	0.6613	0.6190	0.9752	0.7069
	Global Success Ratio	0.2984	0.2168	0.2649	0.2619	0.2779	0.2640
Data Cache	Success Ratio	0.5962	0.5788	0.6052	0.6605	0.6430	0.6167
	Global Success Ratio	0.4926	0.2826	0.4718	0.4620	0.0969	0.3612

Table 4.2.3 gives the miss ratios found in the instruction and data caches. We note that the miss ratios there, for the nonprefetching case, are comparable to the design target miss ratios (DTMRs) proposed in [29], [30], and to the other miss ratios reported from measured real systems and standard workloads, e.g., [10].

We distinguish three kinds of cache miss ratios when a cache prefetching strategy is used; note that these terms are also defined in Table 1.2.

The *True Miss Ratio* is the ratio of the total number of true cache misses to the total number of processor references to the cache. A true cache miss is defined as an event in which the address referenced by the processor is not found in the cache and the address, if issued by the prefetching unit, has not been sent to the main memory yet.

The *Partial Miss Ratio* is the ratio of the total number of partial cache misses to the total number of processor references to the cache. A partial cache miss is defined as an event in which the address referenced by the processor is not found in the cache but the address has already been issued by the prefetching unit and sent to the main memory.

The *Total Miss Ratio* is the sum of the true miss ratio and the partial miss ratio.

The true miss ratio is a better indicator of the accuracy of a cache prefetching strategy than total miss ratio. It is not fair to count a partial cache miss as a genuine demand miss because the prefetching strategy is already halfway to bringing the required block to the cache. If the memory latency were smaller, the prefetch might have already finished and the partial cache miss might not have been a miss at all.

As shown in Table 4.2.3, cache prefetching strategies do not reduce total cache miss ratios or true cache miss ratios as much as one would have expected, or as much as has been reported earlier in [28], [29]. The reason for this is that the line size used in the base system (64-bytes) is much larger than that used in the earlier studies (16-bytes) and prefetching declines rapidly in effectiveness with larger line sizes [25]. This change in line size reflects changes in typical implementations since the time of the earlier work. (The issue of line (block) size is discussed further in Section 5.4.)

Table 4.2.4 shows the success ratios and global success ratios for the instruction and data caches. The *success ratio* is the ratio of the total number of useful prefetches issued to the total number of prefetches issued. A *useful prefetch* is one which fetches a line that is referenced before it is replaced. Success ratio alone is not sufficient to evaluate the accuracy of a prefetching strategy, because a prefetching strategy may issue only a few prefetches and attain a high success ratio, yet fail to catch most demand misses.

A better metric to evaluate the accuracy of a cache prefetching strategy is the *global success ratio* (GSR), which is the fraction of cache misses which are avoided or partially avoided (i.e., partial cache misses). A GSR of zero implies that a prefetching strategy does not save any cache misses while a GSR of one means that it catches all of them. GSR is defined by the equation:

$$GSR = \frac{\text{total number of correct prefetches}}{\text{total number of correct prefetches} + \text{total number of true cache misses}}.$$

The fallacy of success ratio as a metric is illustrated by *thread*. Note that *thread* achieves a very high success ratio, yet its GSR tells us that it does not perform much better than other prefetching strategies. *Thread* is a very conservative strategy; it will not issue a prefetch unless the current cache block has been accessed at least twice. It attains a high success ratio but it fails to capture many demand misses.

Table 4.2.4 helps to explain the small decrease in miss ratios found in Table 4.2.3. Because the global success ratios are low (26 percent for the instruction cache and 36 percent for the data cache), cache prefetching strategies are not very successful in reducing cache miss ratios in the baseline system. Note again that these results are quite different than those in [28], [29] because the baseline system has the longer line size of 64-bytes.

Fig. 4.2.1 gives the compositions of prefetches for the instruction and data caches. We classify prefetches into three different categories.

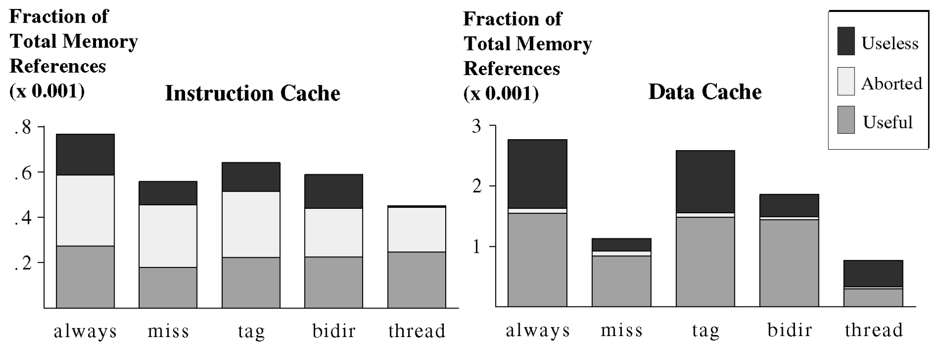


Fig. 4.2.1. Average distribution of prefetches.

(1) Useful Prefetches

They are prefetches whose addresses are referenced by the processor when they are still residing in the caches or being prefetched from the main memory.

(2) Useless Prefetches

They are prefetches whose addresses are not referenced by the processor before they are replaced in the caches. These constitute "memory pollution."

(3) Aborted Prefetches

They are prefetches which are discarded in the prefetch address buffer because:

1) the prefetch address buffer overflows; or
2) the prefetch address has already been referenced by the processor after the prefetch was issued but the prefetch address has not been sent to the main memory yet.

Fig. 4.2.1 explains why the global success ratio is low for the instruction cache. More than 40 percent of all prefetches for the instruction cache are aborted. Most of these aborted prefetches are actually correct prefetches, but the prefetching unit fails to send them to the main memory in time because the data bus is busy and they are not issued early enough. Notice that incorrect prefetches are seldom discarded in the prefetch address buffer because:

1) From the simulation results, the prefetch address buffer seldom overflows; and
2) Incorrect prefetches will not be referenced when they are still in the prefetch address buffer.

As a result, incorrect prefetches are always sent to the main memory successfully, while many correct prefetches are aborted.

Although very few prefetches are aborted in the data cache, many prefetches are useless, as shown in Fig. 4.2.1. They degrade the performance of cache prefetching by increasing the data bus traffic and replacing useful blocks from the data cache.

4.3 An Idealized System

In this section, we will study some of the architectural constraints that limit the performance of a cache prefetching strategy. The major limitations which reduce the effectiveness of cache prefetching are conflicts and delays in accessing the caches, the data bus, and the main memory. By running simulations on a hypothetical system in which these limitations are removed, we will be able to evaluate the extents to which these constraints affect a cache prefetching strategy. We refer to the system in which the constraints are removed as "idealized" or "ideal." Note that the ideal system that we describe below is just that—idealized—some aspects of it are not feasible or even possible for a reasonable implementation. Our ideal system is made so by:

1) The cache is ideal if there is a special access port to the tag array for prefetch lookups, and there is a special access port to the data array for prefetch loads. This port is as wide as the data bus so that there is no need to buffer prefetched blocks coming off the data bus.
2) The data bus is ideal if there is a private bus connecting the prefetching unit and the main memory and the width of this bus is as large as the prefetch block size so that an entire prefetched block takes only one cycle to transfer from the main memory to the destination cache.
3) The main memory is ideal if memory banks are dual ported for regular and prefetch accesses and prefetches take zero time to access these memory banks.

Table 4.3.1 shows the performance of the eight systems in which each component can be either ideal or realistic, where "realistic" corresponds to the baseline system. We identify each system by a three letter word: The first letter can be C or c, meaning that the caches are ideal or realistic, respectively; the second letter can be B or b, meaning that the data bus is ideal or realistic, respectively; and the third letter can be M or m, meaning that the main memory is ideal or realistic, respectively.

In the most ideal system (CBM), cache prefetching reduces MCPI by 46 percent on average. Since there is no penalty in using the cache ports, the data bus, and the main memory, system CBM favors aggressive strategies that issue lots of prefetches. Hence, *always prefetch* and *tag prefetch* perform the best. These results also confirm previous work (e.g., [25], [28], [29]) which suggested that prefetching could yield significant benefits; that work, however, ignored most of the timing and implementation issues considered here.

Cache prefetching, however, increases MCPI by 18 percent on average in the real system (cbm). Because resources are limited, aggressive strategies like always and tag are heavily penalized and yield the lowest performance.

TABLE 4.3.1
RELATIVE MCPI

Prefetching Strategies	CBM	cBM	CbM	CBm	Cbm	cBm	cbM	cbm
none	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*
always	0.3952	1.1724	0.6636	0.4937	0.8169	1.1245	1.1655	1.2943
miss	0.7166	1.1214	0.8547	0.7716	0.9787	1.0719	1.1118	1.2049
tag	0.3952	1.1117	0.6509	0.4835	0.7890	1.0632	1.1039	1.2186
bi-dir	0.4568	1.0336	0.6509	0.5341	0.7555	1.0354	1.0549	1.1408
thread	0.7142	0.9608	0.8810	0.7783	0.9832	0.9424	0.9840	1.0596
prefetch average	0.5356	1.0800	0.7402	0.6122	0.8647	1.0475	1.0840	1.1836

All data are normalized by the MCPI when no prefetching strategies are used.

TABLE 4.3.2
RESULTS FROM THE SIGN TABLE ANALYSIS ON AVERAGE RELATIVE MCPI

Effects or Interactions	Magnitude	Variations Explained (%)
Cache	-0.2053	82.14
Bus	-0.0746	10.86
Memory	-0.0335	2.19
(Cache) (Bus)	-0.0396	3.06
(Bus) (Memory)	0.0225	0.99
(Cache) (Memory)	-0.0168	0.55
(Cache) (Bus) (Memory)	-0.0105	0.22

We analyzed the effects of the three factors, cache, bus, and main memory, by using the technique of a sign table [17]. Table 4.3.2 shows the results of this analysis. The effects due to caches alone account for more than 82 percent of the variations in average relative MCPI. Fortunately, making the caches ideal, or close to ideal, is feasible, but making the data bus and main memory ideal are not.

5 EFFECTS OF SYSTEM RESOURCES ON CACHE PREFETCHING

In this section, we consider each design parameter separately and analyze its impact on the effectiveness of prefetching. Extensive and more detailed quantitative results can be found in the appendix of [33]. In particular, in [33] we show the results for many of our experiments for each individual workload.

5.1 Single vs. Double Ported Cache Tag Arrays

Table 5.1.1 shows the impact of cache tag ports on relative MCPI. We see that the impact of a second cache tag port on performance is enormous when cache prefetching strategies are used. The average relative MCPI when the cache tag arrays are single ported is more than 12 times bigger than that when the arrays are double ported. Conflicts over the cache ports account for an average of 59 percent of total stalls when the tag arrays are single ported. This is consistent with the failure of both the Amdahl 470V/6 [25] and the Fairchild/Intergraph Clipper [5] to gain from prefetching; neither permitted prefetch tag accesses in parallel with the CPU accesses.

When the cache tag arrays change from double to single ported, relative MCPI increases 15-fold for *always* and *thread*, since both strategies look up the cache tag arrays immediately after each memory access made by the processor. The situation for *bi-dir* is even worse: Its relative MCPI is almost twice that of *always* and *thread*, because it looks up the tag arrays twice for every memory access: once for the next cache block and once for the block immediately before. Strategies *miss* and *tag*, on the other hand, are only slightly affected by the type of cache tag ports. Because both strategies look up the tag arrays very rarely, their relative MCPIs increase by only about 0.05 when the tag arrays change from double to single ported.

In general, if a prefetch strategy looks up the cache tag arrays frequently, extra access ports to the tag arrays are vital. If these ports are not available, contention for them will be so damaging to performance that the strategy becomes useless.

5.2 Single vs. Double Ported Cache Data Arrays and Buffering

Table 5.2.1 gives the relative MCPI when the cache data arrays are single and double ported; note that the tag arrays are double ported in this case. The issue of single or double ported cache data arrays is far less important than that of the cache tag arrays, because prefetch strategies access the data arrays far less frequently than they do to the tag arrays. When the data arrays are single ported, however, all prefetching strategies give a relative MCPI greater than one. But, when the data arrays are double ported, all strategies are able to reduce MCPI and give a relative MCPI smaller than one. On average, relative MCPI decreases by about 0.32, a significant amount. Conflicts over cache ports total about 13 percent of all stalls on average when the data

TABLE 5.1.1
RELATIVE MCPI

Prefetching Strategies	Cache Tag Arrays	
	single ported	double ported
none	*1.0000*	*1.0000*
always	15.9536	1.2943
miss	1.2415	1.2049
tag	1.2676	1.2186
bi-dir	29.0202	1.1408
thread	15.6638	1.0596
prefetch average	12.6293	1.1836

All data are normalized by the MCPI when no prefetching strategies are used with the same type of cache tag arrays.

TABLE 5.2.1
RELATIVE MCPI

Prefetching Strategies	Cache Data Arrays		
	single ported	double ported	single ported (buffered)
none	*1.0000*	*1.0000*	*1.0000*
always	1.2943	0.8169	0.9135
miss	1.2049	0.9787	0.9846
tag	1.2186	0.7890	0.8781
bi-dir	1.1408	0.7555	0.8402
thread	1.0596	0.9832	0.9898
prefetch average	1.1836	0.8647	0.9212

All data are normalized by the MCPI when no prefetching strategies are used with the same type of cache data arrays.

arrays are single ported. When they are double ported, these conflicts completely vanish and result in no processor stalls.

Double ported data arrays are no doubt a highly desirable feature for cache prefetching strategies. However, adding an extra access port to a cache data array is very costly. A more practical solution would be to provide some buffering for the data arrays. In this scheme, a prefetched block is first stored in a buffer after it has arrived from the main memory. When the data array becomes free, the contents of the buffer are then loaded into the data array. If the processor needs to access the data array during the load, the prefetch load will be halted to allow access by the processor. After the processor finishes its access, the load will resume. In this way, the processor does not need to wait to access the data arrays and the data bus will be freed once a prefetched block gets off the bus. Note that there is an additional optimization we have not considered in this paper, which is a double ported prefetch buffer. Such a buffer would permit the processor to access any valid data bytes in the prefetch buffer as they arrive and before they are loaded into the data array; we believe that such a double ported prefetch buffer might have a measurable performance impact and we hope to consider it in future research.

Table 5.2.1 shows that such a buffering scheme indeed improves the performance of cache prefetching effectively and at a lower hardware cost. Relative MCPI decreases by 0.26 on average and this is almost as good as when the cache data arrays are double ported.

5.3 Cache Size

Fig. 5.3.1 depicts the relative MCPI for various cache sizes. Performance of a prefetch strategy generally improves as the cache size increases, although, even for a 1MB cache, prefetching still does not improve performance in the baseline system.

When a cache is larger, a prefetched block is more likely to reside in the cache for a longer period of time before it is replaced. Consequently, it has a higher chance of being referenced by the processor while it is still in the cache. Moreover, a useless prefetch is less likely to replace a useful block, i.e., to pollute the cache, if the cache is larger.

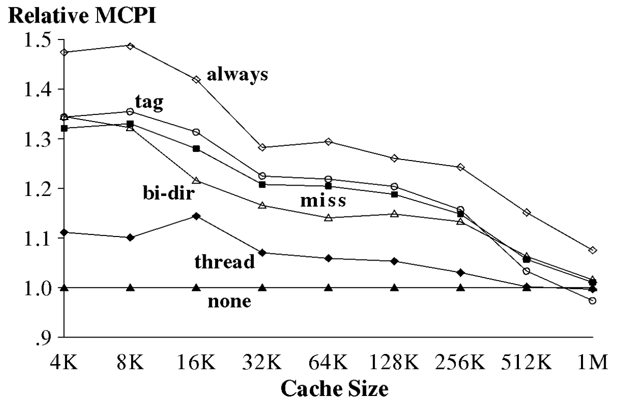


Fig. 5.3.1. Relative MCPI. All data are normalized by the MCPI when no prefetching strategies are used with the same cache size.

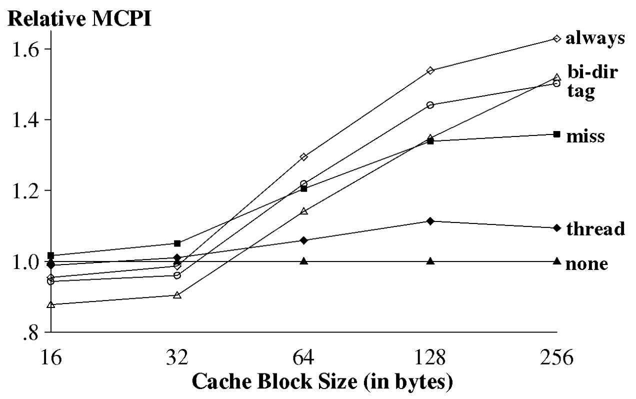


Fig. 5.4.1. Relative MCPI. All data are normalized by the MCPI when no prefetching strategies are used with the same cache block size.

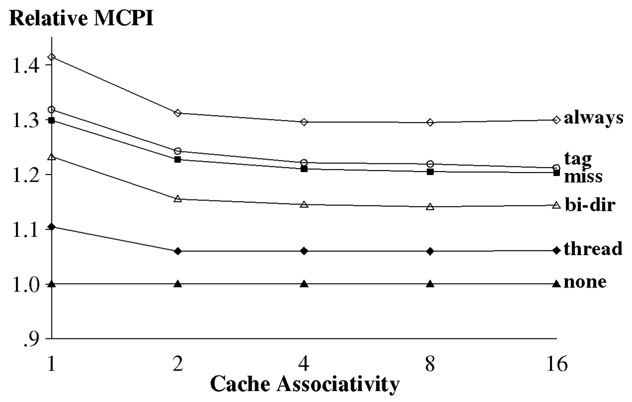


Fig. 5.5.1. Relative MCPI. All data are normalized by the MCPI when no prefetching strategies are used with the same cache associativity.

Another reason for the improvement with cache size is that there is less interference caused by prefetches in large caches. The bigger the caches, the fewer the cache misses; hence, prefetches are less likely to interfere with normal cache operations.

In general, cache prefetching strategies should be used only when there is sufficient space in the caches; otherwise, the effects of interference and cache pollution due to useless prefetches will overshadow any benefits brought about by cache prefetching. The effect of cache pollution was clearly shown in [28], [29], where it can be seen that prefetching only cuts the miss ratio for sufficiently large cache memories.

5.4 Cache Block Size

Fig. 5.4.1 illustrates the effect of cache block size on prefetching. Note that we assume that, in all cases, the block size is the same in the instruction cache, the data cache, and is also the transfer size for both demand and prefetch transfers. Fig. 5.4.1 shows that, when cache blocks are 16 or 32 bytes long, most prefetch strategies perform better than when there is no prefetching. But, as the cache block size increases, relative MCPI rises above one and prefetching hurts performance. This confirms some results in [25], which showed that the miss ratio improvement for prefetching increases with decreasing block size.

The increase in relative MCPI with increasing block size is mainly a result of more cache port conflicts. On average, cache port conflicts account for only 2 percent of total stalls when cache blocks are 16 bytes long. But, when cache blocks are 16 times larger, the percentage rises to 20 percent. Because the port width of the data arrays remains the same but the block size increases, a cache needs more cycles to load a prefetched block into its data array. Consequently, a single prefetch load holds up the data array for more consecutive cycles and creates more chances for port conflicts.

Note also that, as the block size increases, prefetching is less likely to improve the miss ratio. The reason is that the prefetched block is less likely to be useful (it is further away from the current point of reference) and the corresponding replaced block is more likely to be useful (since it is bigger). A similar effect is visible in [30], where we see that increases in block size produce diminishing returns; since fetching a double block is very similar to fetching a block and prefetching the next one, this is essentially the same effect.

5.5 Cache Associativity

Fig. 5.5.1 illustrates the impact of cache associativity on prefetching. When the caches change from direct mapped to two-way mapped, relative MCPI decreases by 0.07 on average, but remains almost constant as cache associativity increases further.

There are more prefetches issued, both useless and useful ones, in a direct mapped cache than in a set associative cache. This is because in a direct mapped cache, a cache block can be placed in only one slot. A prefetched block, therefore, has a higher chance of replacing a useful block from the cache. The prefetch unit detects more potential misses and issues more prefetches. Bus traffic increases and more conflicts arise. In a multiply mapped (set associative) cache, a prefetched block can be placed in more than one slot and, hence, the problem of cache pollution is less serious than in a direct mapped one.

From these results, we conclude that a cache prefetch strategy performs more effectively in set associative caches than in direct mapped caches.

5.6 Split vs. Nonsplit Bus Transaction

Table 5.6.1 shows the relative MCPI for the two types of bus transactions. Relative MCPI decreases by 14 percent on average when bus transactions change from nonsplit to split. For both caches, most aborted prefetches become useful prefetches when the data bus supports split transactions. Recall that an aborted prefetch is one which is canceled in the prefetch address buffer before it can be sent to the main memory. Although most aborted prefetches correctly predict future memory references, they fail to be sent to the main memory because the data bus is too busy. When split transactions are available, a transaction does not have to hold up the bus during its entire access to the main memory. It is, therefore, easier for the prefetching unit to acquire the bus and send prefetch requests to the main memory. Many would-be aborted prefetches now become useful. True miss ratios decrease and the prefetch strategy is more able to improve MCPI.

TABLE 5.6.1
RELATIVE MCPI

Prefetching Strategies	Bus Transaction	
	non-split	split
none	*1.0000*	*1.0000*
always	1.2943	1.0937
miss	1.2049	1.0100
tag	1.2186	1.0335
bi-dir	1.1408	1.0547
thread	1.0596	0.9309
prefetch average	1.1836	1.0246

All data are normalized by the MCPI when no prefetching strategies are used.

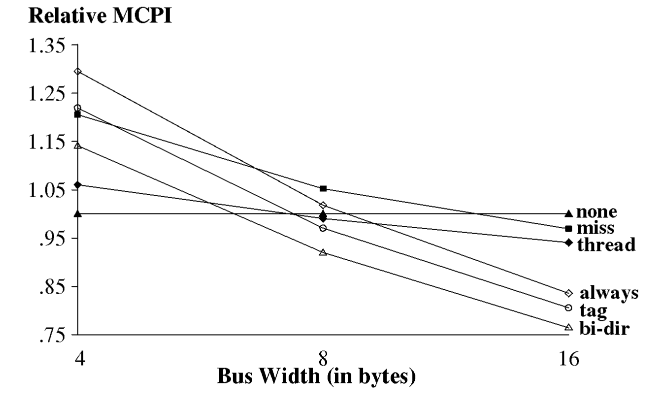


Fig. 5.7.1. Relative MCPI. All data are normalized by the MCPI when no prefetching strategies are used with the same bus width.

The type of bus transaction also affects the impact of other system resources on cache prefetching. For example, the number of memory banks has very little effect on relative MCPI unless the data bus supports split transactions. We will consider this issue later in Section 5.9.

5.7 Bus Width

Fig. 5.7.1 shows the relative MCPI as the bus width varies. In order to take full advantage of a wider bus, we assume that the cache data ports are as wide as the data bus so that, when a prefetched block arrives, a cache can load the block into its data array in a single cycle and hence no buffering is needed. Since a line can be much bigger than the bus width, of course, this does not mean that we have a one cycle block load.

Fig. 5.7.1 shows that, as the bus width increases, relative MCPI begins to fall below one for the base system. The major reason is because there are fewer cache port conflicts. On average, the percentage of total stalls contributed by cache port conflicts drops by 10 percent when the bus width increases from 4 bytes to 16 bytes. Since a cache block is 64 bytes long by default, a cache takes 16 cycles to finish a prefetch loading when the bus is 4 bytes wide. But, when the bus is 16 bytes wide, it only takes four cycles to do so. A wider bus, therefore, relieves the contention for cache data ports.

However, the above argument holds only when the cache data ports are as wide as the data bus. If the ports are smaller in size, a prefetch load will take multiple cycles to finish. In this case, we need some buffering schemes for the data arrays or we need to lock the bus until the prefetch load finishes.

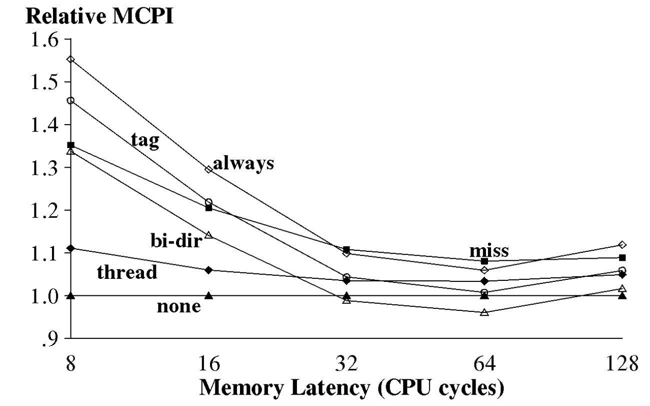


Fig. 5.8.1. Relative MCPI. All data are normalized by the MCPI when no prefetching strategies are used with the same memory latency.

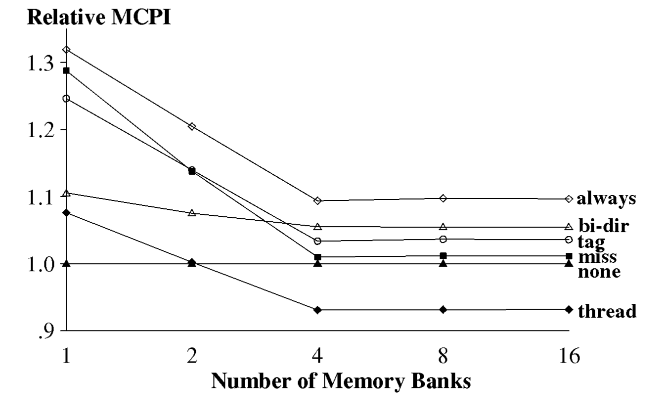


Fig. 5.9.1. Relative MCPI. All data are normalized by the MCPI when no prefetching strategies are used with the same number of memory banks.

5.8 Memory Latency

Fig. 5.8.1 illustrates the performance of prefetch strategies as the memory latency increases. Although relative MCPI decreases as the memory latency increases from eight to 64 processor cycles, it starts to rise as the memory latency increases further. Two reasons account for these U-shaped curves:

1) Fewer cache port conflicts: When the memory latency increases, prefetch loadings happen less frequently in the same period of time. Conflicts over cache ports occur more rarely and the processor is stalled less for that reason. This explains why relative MCPI falls. On average, the percentage of total stalls contributed by these conflicts drops by 13 percent when the memory latency increases from eight to 128 processor cycles.
2) More data bus conflicts: As the memory latency increases, a prefetch has to hold the bus longer and is more likely to conflict with a regular demand miss. Hence, the processor is stalled more. This explains why relative MCPI starts to rise when the memory latency is larger than 64 cycles. The percentage of total stalls contributed by bus conflicts rises by 11 percent on average when the memory latency increases from eight to 128 processor cycles.

If the performance gap between memory and processor speed continues to increase, cache prefetching will become increasingly useful until the memory latency passes a certain latency for that architecture. For our base system, the latency at which prefetching is most useful is around 64 processor cycles.

5.9 Number of Memory Banks

If bus transactions are nonsplit, having more memory banks cannot improve the performance of cache prefetching because a nonsplit bus transaction does not allow multiple prefetches to access the main memory concurrently. Therefore, in this section, we assume that bus transactions are split.

Fig. 5.9.1 depicts the impact of the number of memory banks on cache prefetching. Relative MCPI decreases by 0.18 on average when the number of memory banks increases from one to four. When there are more memory banks, relative MCPI levels off.

One reason for this improvement is because there are fewer conflicts over the memory banks. These conflicts account for 16 percent of total stalls when there is only one memory bank. But, when there are 16 of them, the percentage drops to 2 percent.

Another advantage to multiple banks is that multiple prefetches can occur in parallel when there are more memory banks. As long as the cache doesn't experience a demand miss, any memory reference can initiate a prefetch. However, if there is only one memory bank for a prefetch to access, then the prefetches must be processed sequentially by the memory.

5.10 Bus Traffic

Bus traffic is another aspect of a cache prefetching strategy. Previous measurements [29] show that prefetching increases bus traffic from 20 percent to 40 percent. This additional traffic may seriously affect the performance of the system. In order to study the effect of bus traffic, we added an extra synthetic load on the bus by simulating a direct memory access (DMA) of variable speed made by some imaginary IO devices. The priority of these DMA accesses to acquire the data bus is higher than that of ordinary prefetches but lower than those of demand misses and write backs.

Fig. 5.10.1 illustrates the effect of bus traffic on prefetching. As the amount of bus traffic increases, relative MCPI starts to converge to one. This is because there is less and less bus bandwidth to send prefetch requests to the main memory. When the amount of bus traffic increases, more and more prefetches become aborted until all issued prefetches are aborted. Since it becomes increasingly difficult to access the data bus, very few prefetch requests can be sent to the main memory successfully. When the bus utilization used by DMA reaches 80 percent, almost no prefetches can acquire the data bus and the processor runs as if there is no prefetching.

Fig. 5.10.1 seems to suggest that heavier bus traffic actually improves the relative performance of cache prefetching since relative MCPI decreases. This is true in the baseline system because prefetching does not improve performance. Heavier bus traffic helps to reduce the amount of these undesirable prefetches and, hence, relative MCPI decreases. But, for a system in which prefetching is beneficial, heavier bus traffic will reduce the amount of desirable prefetches and relative MCPI will rise. In any case, relative MCPI converges to one as the data bus becomes more congested.

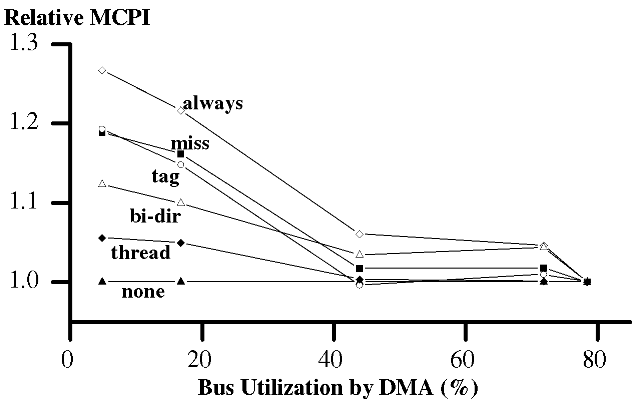


Fig. 5.10.1. Relative MCPI. All data are normalized by the MCPI when no prefetching strategies are used with that DMA bus utilization.

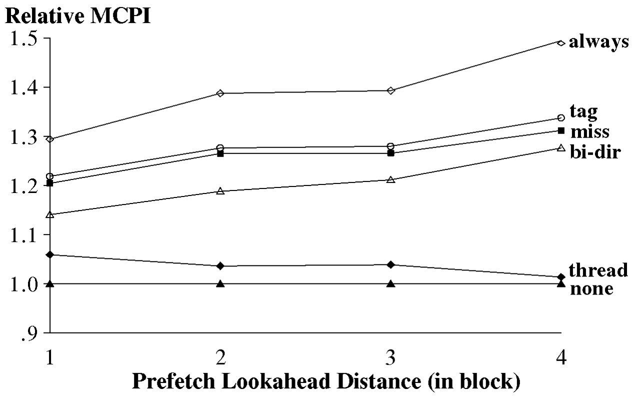


Fig. 5.11.1. Relative MCPI. All data are normalized by the MCPI when no prefetching strategies are used with the same prefetch lookahead distance.

We also note that shared memory multiprocessor systems, particularly those with a shared bus, are typically limited in their throughput by the bus bandwidth [11]. In such a case, prefetching will almost certainly lower overall throughput.

5.11 Prefetch Lookahead Distance

Ideally, we would like to issue a useful prefetch well in advance so that, by the time the prefetch target is referenced by the processor, the block will already be in the cache. This can be accomplished by prefetching block p + LA instead of block p, where p is the original block address requested by the prefetching strategy and LA is the lookahead distance, in blocks.

Fig. 5.11.1 shows the relative MCPI for different prefetch lookahead distances. For all strategies except *thread*, relative MCPI rises with increasing lookahead distance. This is because the chance that block p + LA will be referenced by the processor decreases as LA increases, since the effect of spatial locality is diminishing. As a result, more prefetches are not referenced by the processor before they are replaced from the caches. The number of useless prefetches increases with higher LA. More bus and cache port conflicts occur and performance degrades, even though an issued prefetch is more likely to actually complete before the block is referenced.

Kim et al. suggested a better way to increase the lookahead distance for *thread* algorithm [18]. Recall that *thread* records all blocks brought into the cache by the block currently accessed by the processor. When the current block is

TABLE 6.1.1
THE WORST AND THE BEST VALUES FOR EACH SYSTEM PARAMETER IN TERMS OF THE EFFECTIVENESS OF PREFETCHING

System Parameters	Worst Value	Rel. MCPI	Abs. MCPI	Rel. CPI	Abs. CPI	Best Value	Rel. MCPI	Abs. MCPI	Rel. CPI	Abs. CPI	% improve
Cache tag array	single	12.6293	1.1645	1.5538	1.8815	double	1.1836	0.1091	1.0171	1.2317	90.63
Cache data array	single	1.1836	0.1091	1.0171	1.2317	double	0.8647	0.0797	0.9918	1.2009	26.94
Cache size	8KB	1.3193	0.4960	1.0266	1.6263	1MB	1.0508	0.0034	0.9999	1.1280	20.35
Cache block size	256 bytes	1.4206	0.1610	1.0140	1.3065	16 bytes	0.9562	0.1680	0.9967	1.3096	32.69
Cache associativity	1	1.2733	0.2210	1.0141	1.3275	16	1.1836	0.1076	1.0082	1.2297	7.04
Bus transaction	non-split	1.1836	0.1091	1.0171	1.2317	split	1.0246	0.0938	1.0038	1.2149	13.43
Bus width	4 bytes	1.1836	0.1091	1.0171	1.2317	16 bytes	0.8630	0.0667	0.9974	1.1843	27.09
Memory latency	8 cycles	1.3613	0.0829	1.0170	1.2015	64 cycles	1.0284	0.2881	1.0065	1.4358	24.45
No of memory banks	1	1.2067	0.2092	1.0366	1.3475	16	1.0260	0.0939	1.0040	1.2151	14.97
Lookahead distance	4 blocks	1.2868	0.1186	1.0241	1.2401	1 block	1.1836	0.1091	1.0171	1.2317	8.02

The average data shown here are the same as those reported in Section 5, where only one particular parameter is changed in the baseline system. The last column gives the improvement in average relative MCPI when a particular parameter changes from its worst value to its best.

TABLE 6.1.2
RELATIVE AND ABSOLUTE MCPI FOR THE OPTIPREFETCH SYSTEM

Cache Prefetching Strategies	Relative MCPI						Absolute MCPI					
	Trace Name					Trace avg.	Trace Name					Trace avg.
	cad	comp	fp	text	unix		cad	comp	fp	text	unix	
none	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*0.4292*	*0.6273*	*2.3140*	*0.2090*	*0.4362*	*0.8031*
always	0.5941	0.5133	0.3110	0.4295	0.8686	0.5433	0.2550	0.3220	0.7197	0.0897	0.3789	0.3531
miss	0.6980	0.6313	0.5202	0.6031	0.9186	0.6743	0.2996	0.3960	1.2038	0.1260	0.4007	0.4852
tag	0.5902	0.5121	0.3111	0.4428	0.8678	0.5448	0.2533	0.3212	0.7200	0.0925	0.3786	0.3531
bi-dir	0.8069	0.7308	0.4583	0.6879	0.8912	0.7150	0.3463	0.4584	1.0605	0.1437	0.3887	0.4796
thread	0.7836	0.7076	0.7882	0.7073	0.8941	0.7761	0.3363	0.4438	1.8238	0.1478	0.3900	0.6284
prefetch avg	0.6946	0.6190	0.4778	0.5741	0.8881	0.6507	0.2981	0.3883	1.1056	0.1200	0.3874	0.4599

later reaccessed, the recorded blocks will be prefetched if they are not already in the cache. To increase the lookahead distance, we can keep track of all the blocks brought into the cache by the next block, rather than the current block. We implemented this scheme and found out that relative MCPI decreases with increasing lookahead distance. Our results are consistent with those presented in [18]. Because this scheme is able to maintain a high prefetch accuracy, increasing lookahead distance does not generate more useless prefetches. Since we are prefetching earlier, it is more likely that a prefetched block is already in the cache when the processor references it. Hence, relative MCPI improves.

From our results, increasing lookahead distance simply by prefetching block p + LA instead of block p is generally not a good idea.

6 SYSTEM DESIGNS

6.1 When Prefetching Is Most Effective

Table 6.1.1 gives a summary of the impacts of various system resources on the performance of cache prefetching. We see that, when a system parameter changes from its worst value (the least favorable setting for prefetching) to its best (the most favorable one), the improvement in the performance of cache prefetching ranges from 7 percent to as high as 90 percent. Please note that the use of the word "best" is in relation to when prefetching is most useful, and does not refer to the overall system design or level of performance. In this section, we will focus our attention on a system in which each one of these parameters is chosen favorably (but feasibly) for prefetching. We call it the OptiPrefetch system; the configuration for the OptiPrefetch system is given by the best values shown in Table 6.1.1 except for the cache size. Since the cache size is one of the most important factors in system performance, we chose the instruction and data cache size in the OptiPrefetch system to be the same as that in the baseline system (64K bytes) in order to facilitate comparisons.

Tables 6.1.2 and 6.1.3 show the relative and absolute MCPI and CPI for different address traces in the OptiPrefetch system. The results are much better than those found in the baseline system. All prefetching strategies perform better than when there is no prefetching. The relative MCPI for all strategies averages 0.65, meaning that prefetching reduces MCPI by 35 percent on average relative to the corresponding baseline system.

Prefetching performs the best on the address trace fp, for which MCPI decreases by more than 52 percent on average. *Always* and *tag* even reduce MCPI by as much as 69 percent. Prefetching continues to perform the worst on the address trace unix, for which the reduction in MCPI averages 11 percent.

TABLE 6.1.3
RELATIVE AND ABSOLUTE CPI FOR THE OPTIPREFETCH SYSTEM

Cache Prefetching Strategies	Relative CPI						Absolute CPI					
	Trace Name					Trace avg.	Trace Name					Trace avg.
	cad	comp	fp	text	unix		cad	comp	fp	text	unix	
none	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.0000*	*1.5253*	*1.7301*	*3.4329*	*1.3141*	*1.5319*	*1.9068*
always	0.8861	0.8239	0.5361	0.9097	0.9629	0.8237	1.3515	1.4255	1.8399	1.1955	1.4749	1.4575
miss	0.9153	0.8666	0.6770	0.9371	0.9770	0.8746	1.3961	1.4993	2.3240	1.2315	1.4966	1.5895
tag	0.8851	0.8235	0.5361	0.9117	0.9626	0.8238	1.3499	1.4249	1.8403	1.1982	1.4747	1.4576
bi-dir	0.9460	0.9029	0.6354	0.9508	0.9693	0.8808	1.4428	1.5620	2.1810	1.2494	1.4848	1.5840
thread	0.9391	0.8941	0.8573	0.9536	0.9699	0.9228	1.4325	1.5470	2.9420	1.2531	1.4857	1.7321
prefetch avg	0.9143	0.8622	0.6484	0.9326	0.9684	0.8652	1.3945	1.4917	2.2254	1.2255	1.4833	1.5641

TABLE 6.1.4
STALL BREAKDOWNS IN PERCENTAGE FOR THE OPTIPREFETCH SYSTEM

CPU Stall Reasons	Cache Prefetching Strategies						Prefetch Average
	none	always	miss	tag	bidir	thread	
cache misses	*81.25*	67.78	74.75	67.98	74.65	76.47	72.33
prefetch (cache)	*0.00*	0.00	0.00	0.00	0.00	0.00	0.00
prefetch (bus)	*0.00*	3.95	0.95	3.86	2.43	0.92	2.42
prefetch (mem)	*0.00*	0.01	0.00	0.01	0.00	0.00	0.00
write buffer	*1.01*	0.19	0.50	0.19	0.40	0.91	0.44
write hits	*17.74*	28.09	23.80	27.97	22.51	21.70	24.81

TABLE 6.1.5
INSTRUCTION AND DATA CACHE MISS RATIOS IN THE OPTIPREFETCH SYSTEM

Cache Type	Cache Miss Ratios	Cache Prefetching Strategies						Prefetch Average
		none	always	miss	tag	bi-dir	thread	
Instr. Cache	True Miss Ratio	*0.0024*	0.0004	0.0013	0.0004	0.0007	0.0007	0.0007
	Partial Miss Ratio	*0.0000*	0.0011	0.0000	0.0011	0.0015	0.0009	0.0009
	Total Miss Ratio	*0.0024*	0.0015	0.0013	0.0015	0.0022	0.0016	0.0016
Data Cache	True Miss Ratio	*0.0252*	0.0076	0.0160	0.0076	0.0081	0.0204	0.0119
	Partial Miss Ratio	*0.0000*	0.0060	0.0000	0.0060	0.0095	0.0006	0.0045
	Total Miss Ratio	*0.0252*	0.0136	0.0160	0.0136	0.0176	0.0210	0.0164

TABLE 6.1.6
SUCCESS RATIOS AND GLOBAL SUCCESS RATIOS IN THE OPTIPREFETCH SYSTEM

Cache Type	Prefetch Success Ratios	Cache Prefetching Strategies					Prefetch Average
		always	miss	tag	bi-dir	thread	
Instr. Cache	Success Ratio	0.8509	0.8867	0.8773	0.8333	0.7942	0.8485
	Global Success Ratio	0.8432	0.4514	0.8101	0.7050	0.5768	0.6773
Data Cache	Success Ratio	0.6705	0.6289	0.6769	0.7682	0.7501	0.6989
	Global Success Ratio	0.6039	0.3308	0.5921	0.5553	0.1792	0.4523

The OptiPrefetch system favors aggressive strategies like *always* and *tag*, which issue lots of prefetches. The baseline system, on the other hand, favors conservative strategies like thread and bi-dir, which only issue prefetches with high chances of being referenced. This difference arises from the fact that there are more resources and bandwidth available in the OptiPrefetch system for prefetching than in the baseline system. Strategies which make better use of these resources by aggressively issuing prefetches win in the OptiPrefetch system. But, when resources are limited, these strategies run into too many conflicts. The type of prefetching strategy a system should use is, therefore, highly dependent on the availability of system resources.

Table 6.1.4 shows the stall compositions. There are no conflicts over the cache data ports because the data arrays are dual ported. Contention over the data bus is minimal and it accounts for less than 2.5 percent of total stalls on average. Since there are 16 memory banks available, conflicts over memory banks occur very rarely. Because conflicts are rare, the OptiPrefetch system does not heavily penalize useless prefetches. An aggressive strategy takes advantage of this fact by issuing more prefetches and is therefore better able to improve MCPI.

Table 6.1.5 lists the miss ratios found in the instruction and data caches. Prefetching strategies lower both the total miss ratios and true miss ratios significantly in the OptiPrefetch

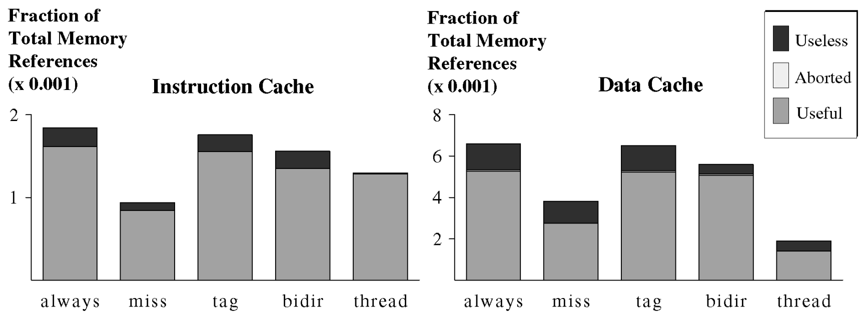


Fig. 6.1.1. Average distribution of prefetches in the OptiPrefetch system.

TABLE 6.2.1
PERFORMANCE OF CACHE PREFETCHING IN SOME POSSIBLE SYSTEMS

System Parameters	A	B	C	D	E	F	G	H	I	J	K	L
Cache tag array	double	double	double	double	double	double	double	double	double	double	double	double
Cache data array	double	double	buffer	double	double	buffer	double	double	double	double	double	double
Cache size	64KB	64KB	64KB	64KB	64KB	64KB	256KB	256KB	256KB	256KB	1MB	1MB
Cache block size	16	16	16	64	64	64	64	64	64	64	64	64
Cache associativity	2	2	2	2	2	2	2	2	2	2	2	2
Bus transaction	non-sp	split	split	non-sp	split	split	non-sp	split	non-sp	split	non-sp	split
Bus width	4	8	16	4	8	16	4	8	16	16	16	16
Memory latency	16	16	16	16	16	16	16	16	16	16	16	16
No. of mem. banks	1	4	4	1	4	4	1	4	4	4	4	4
Lookahead distance	1	1	1	1	1	1	1	1	1	1	1	1
Relative MCPI	0.8978	0.6204	0.6192	1.0923	0.7305	0.7482	1.0669	0.6961	0.7833	0.7094	0.6805	0.6184
Absolute MCPI	0.1773	0.1144	0.1147	0.3156	0.0711	0.0704	0.1338	0.0303	0.0333	0.0303	0.0126	0.0112
Relative CPI	0.9701	0.9298	0.9296	1.0189	0.9763	0.9773	1.0007	0.9814	0.9838	0.9815	0.9871	0.9861
Absolute CPI	1.2814	1.2187	1.2191	1.4202	1.1760	1.1753	1.2386	1.1354	1.1383	1.1353	1.1178	1.1164

system. This is because prefetch accuracy is greatly improved. As shown in Table 6.1.6, both the success ratios and global success ratios are high. The global success ratios in the instruction and data caches average 68 percent and 45 percent, respectively. For comparison, the global success ratios in the baseline system averaged about 26 percent and 36 percent for the instruction and data caches, respectively.

Fig. 6.1.1 shows what happens to the prefetches. Because there is abundant bus bandwidth, most prefetch requests can be sent to the main memory successfully and almost no prefetches are aborted. On average, over 80 percent and 69 percent of all prefetches are useful in the instruction and data caches, respectively.

6.2 Possible System Designs

Table 6.2.1 shows some hypothetical system configurations and the performance of cache prefetching in these systems. Prefetching improves performance in all systems except systems D and G. Prefetching strategies perform poorly in systems D and G because the data bus bandwidth is too small (no split transaction, only one memory bank, and the data bus is only four bytes wide).

7 CONCLUSION

In memory systems with appropriately designed hardware, prefetching can often reduce average memory latency. For a cache to prefetch effectively, it is necessary that the cache tag arrays be double ported, and that the data arrays either be double ported or buffered. The cache should also be at least two way set associative. Prefetching is most effective when the cache is large, the block size small, and the memory bus wide. Using a split transaction bus and interleaving main memory help considerably. Such well-designed systems achieve their highest performance with aggressive prefetch strategies which prefetch blocks frequently.

Conversely, in less well-endowed memory systems, prefetching often degrades the memory system performance. In this case, additional conflicts and bus traffic may overshadow any benefit brought about by prefetching. Such systems function best with conservative strategies which only issue prefetches with high chances of being referenced by the processor. It is important to note, therefore, that for prefetching to be useful, considerable extra hardware needs to be dedicated to making it effective; less complex or expensive designs, such as are commonly found, do not permit prefetching to be helpful.

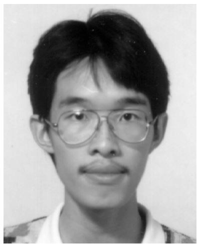
In this paper, we have only considered single level caches, i.e., caches with a processor referencing them on one side and main memory on the other side. We hope in future work to explore two additional issues. One is the use of prefetching in multiple level caches, e.g., on-chip and onboard caches, and the other is the use of prefetching in a multiprocessor system with shared memory. Such systems are often limited by bus bandwidth, in which case prefetching is unlikely to help aggregate system performance.

ACKNOWLEDGMENTS

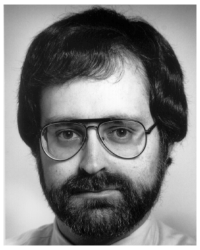
The authors' research is or has been supported in part by the U.S. National Science Foundation under grants MIP-9116578 and CCR-9117028, by NASA under Grant NCC 2-550, by the State of California under the MICRO program, and by Apple Computer Corporation, Intel Corporation, Sun Microsystems, Fujitsu Mircoelectronics, Toshiba Corporation, Sony Research Laboratories, Digital Equipment Corporation, and Philips Laboratories / Signetics.

REFERENCES

[1] J.L. Baer and W.H. Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proc. 15th Int'l Symp. Computer Architecture*, pp. 73-80, June 1988.
[2] Callahan, Kennedy, and Porterfield, "Software Prefetching," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 40-52, Apr. 1991.
[3] T.-F. Chen and J.-L. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," *IEEE Trans. Computers*, vol. 44, no. 5, pp. 609-623, May 1995.
[4] C.-H. Chi, "Compiler Optimization Technique for Data Cache Prefetching Using a Small CAM Array," *Proc. 1994 Int'l Conf. Parallel Processing*, vol. I, pp. 263-266, Aug. 1994.
[5] J. Cho, H. Sachs, and A.J. Smith, "The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER Processor," Technical Report UCB / CSD-86-289, Univ. of California, Berkeley, Mar. 1986.
[6] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors," *Proc. 1993 Int'l Conf. Parallel Processing*, pp. I56-I63, Aug. 1993.
[7] F. Dahlgren and P. Stenstrom, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 7, pp. 733-745, July 1995.
[8] "Pixie," DEC Ultrix manual page, 1991.
[9] J.W.C. Fu and J.H. Patel, "Data Prefetching Strategies for Vector Cache Memories," *Proc. Fifth Int'l Parallel Processing Symp.*, pp. 555-560, May 1991.
[10] J. Gee, M. Hill, D. Penvmatikatos, and A.J. Smith, "Cache Performance of the SPEC Benchmark Suite," *IEEE Micro*, vol. 13, no. 4, pp. 17-27, Aug. 1993.
[11] J. Gee and A.J. Smith, "Evaluation of Cache Consistency Algorithm Performance," *Proc. Mascots '96 (Int'l Workshop Modeling, Analysis, and Simulation of Computer and Telecommunication Systems) Conf.*, pp. 236-249, Feb. 1996.
[12] J.D. Gindele, "Buffer Block Prefetching Method," *IBM Technical Disclosure Bulletin*, vol. 20, no. 2, pp. 696-697, July 1977.
[13] E. Gornish, E. Granston, and A. Veidenbaum, "Compiler Directed Data Prefetching in Multiprocessors with Memory Hierarchies," *Proc. 1990 Int'l Conf. Supercomputing*, pp. 354-368, 1990.
[14] E. Gornish and A. Veidenbaum, "An Integrated Hardware / Software Data Prefetching Scheme for Shared-Memory Multiprocessors," *Proc. 1994 Int'l Conf. Parallel Processing*, vol. II, pp. 281-284, Aug. 1994.
[15] M.D. Hill, manual page on DineroIII, Univ. of California, Berkeley, Oct. 1985.
[16] W. Hollingsworth, H. Sachs, and A.J. Smith, "The Fairchild CLIPPER: Instruction Set Architecture and Processor Implementation," *Comm. ACM*, vol. 32, no. 2, pp. 200-219, Feb. 1989.
[17] R. Jain, *The Art of Computer Systems Performance Analysis*, pp. 283-292. John Wiley & Sons, 1991.
[18] S.B. Kim et al., "Threaded Prefetching: An Adaptive Instruction Prefetch Mechanism," *Microprocessing and Microprogramming*, vol. 39, no. 1, pp. 1-15, Nov. 1993.
[19] L. Kurian, P.T. Hulina, L.D. Coraor, and D.N. Mannai, "Classification and Performance Evaluation of Instruction Buffering Techniques," *Proc. 18th Int'l Symp. Computer Architecture*, pp. 150-159, May 1991.
[20] J.V. Levy, "Buses: The Skeleton of Computer Structures," *Computer Engineering: A DEC View of Hardware Systems Design*. Digital Press, 1978.
[21] D. Poulsen and P.-C. Yew, "Data Prefetching and Data Forwarding in Shared Memory Multiprocessors," *Proc. 1994 Int'l Conf. Parallel Processing*, vol. II, pp. 276-280, Aug. 1994.
[22] S.A. Przybylski, *Cache and Memory Hierarchy Design—A Performance-Directed Approach*, pp. 181-186. Morgan Kaufmann, 1990.
[23] D. Samples, "Mache: No-Loss Trace Compaction," *Performance Evaluation Review*, vol. 17, no. 1, pp. 89-97, May 1989.
[24] R.T. Short and H.M. Levy, "A Simulation Study of Two-Level Caches," *Proc. 15th Int'l Symp. Computer Architecture*, pp. 81-88, June 1988.
[25] A.J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7-21, Dec. 1978.
[26] A.J. Smith, "Characterizing the Storage Process and Its Effects on Main Memory Update," *J. ACM*, vol. 26, no. 1, pp. 6-27, Jan. 1979.
[27] A.J. Smith, "Sequentiality and Prefetching in Data Base Systems," IBM Research Report RJ 1743, 19 Mar. 1976, and *ACM Trans. Data Base Systems*, vol. 3, no. 3, pp. 223-247, Sept 1979.
[28] A.J. Smith, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, pp. 473-530, Sept. 1982.
[29] A.J. Smith, "Cache Evaluation and the Impact of Workload Choice," *Proc. 12th Int'l Symp. Computer Architecture*, pp. 64-75, June 1985.
[30] A.J. Smith, "Line (Block) Size Selection in CPU Cache Memories," *IEEE Trans. Computers*, vol. 36, no. 9, pp. 1,063-1,075, Sept. 1987. (Also see correction *IEEE Trans. Computers*, vol. 38, no. 6, p. 927, June, 1989.)
[31] R.B. Smith, J.K. Archibald, and B.E. Nelson, "Evaluating Performance of Prefetching Second Level Caches," *Performance Evaluation Review*, vol. 20, no. 4, pp. 31-42, May 1993.
[32] A.J. Smith, "Trace-Driven Simulation in Research on Computer Architecture and Operating Systems," *Proc. New Directions in Simulation for Manufacturing and Comm. (SIM94)*, Morito, Sakasegawa, Yoneda, Fushimi, Nakano, eds., pp. 43-49, Tokyo, 1-2 Aug. 1994.
[33] J. Tse and A.J. Smith, "Performance Evaluation of Cache Implementation," Technical Report UCB / CSD-95-877, Univ. of California, Berkeley, June 1995.
[34] D.M. Tullsen and S.J. Eggers, "Limitations of Cache Prefetching on a Bus-Based Multiprocessor," *Proc. 20th Int'l Symp. Computer Architecture*, pp. 278-288, May 1993.
[35] A. Varma and G.K. Sinha, "A Class of Prefetch Schemes for On-Chip Data Caches," technical report, Computer Science Dept., Univ. of California, Santa Cruz, 1992.



John Tse received the BS degree in electrical engineering and computer science in 1992 and the MS degree in computer science in 1995 from the University of California at Berkeley. He is currently a software engineer at Altera Corporation, San Jose. His research interests include computer architecture, partitioning algorithms, and computer music.



Alan Jay Smith received the BS degree in electrical engineering from the Massachusetts Institute of Technology, Cambridge, Massachusetts, and the MS and PhD degrees in computer science from Stanford University, Stanford, California. He was a U.S. National Science Foundation Graduate Fellow.

He is currently a professor in the Computer Science Division of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, where he has been on the faculty since 1974; he was vice chairman of the EECS Department from July 1982 to June 1984. His research interests include the analysis and modeling of computer systems and devices, computer architecture, and operating systems. He has published a large number of research papers, including one which won the IEEE Best Paper Award for the best paper in the *IEEE Transactions on Computers* in 1979. He also consults widely with computer and electronics companies.

Dr. Smith is a fellow of the IEEE and a member of the ACM, IFIP Working Group 7.3, the Computer Measurement Group, Eta Kappa Nu, Tau Beta Pi, and Sigma Xi. He is on the board of directors (1993-1999) of the ACM Special Interest Group on Computer Architecture (SIGARCH), was chairman of SIGARCH 1991-1993, was chairman of SIGOPS 1985-1989, and was program cochair of Hot Chips in 1990, 1994, and 1997.