Classification and Performance Evaluation of Instruction Buffering Techniques*

Lizyamma Kurian, Paul T. Hulina,
Lee D. Coraor and Dhamir N. Mannai
Department of Electrical and Computer Engineering
The Pennsylvania State University
University Park, PA 16802

January 22, 1996

Abstract

The speed disparity between processor and memory subsystems has been bridged in many existing large-scale scientific computers and microprocessors with the help of instruction buffers or instruction caches. In this paper we classify these buffers into traditional instruction buffers, conventional instruction caches and prefetch queues, detail their prominent features, and evaluate the performance of buffers in several existing systems, using trace driven simulation. We compare these schemes with a recently proposed queue-based instruction cache memory. An implementation independent performance metric is proposed for the various organizations and used for the evaluations. We analyze the simulation results and discuss the effect of various parameters such as prefetch threshold, bus width and buffer size on performance.

1 Introduction

CPU speeds have increased tremendously in the recent past, but memory speeds have barely kept pace, resulting in a speed disparity between CPU and memory. This disparity has to be bridged in some way and the processing bandwidths matched in order to avoid system bottlenecks. In many existing systems, the speed gap has been bridged by employing memory interleaving or by using a fast memory (cache or instruction buffer) between the CPU and the main memory. Instruction buffers and cache memories are typically 5 to 10 times faster than main memory and hence can reduce the effective memory access time if carefully designed and implemented. As observed by Hill and Smith [9], cache is a time-tested mechanism for improving memory system performance by reducing access time and memory traffic through the exploitation of temporal and spatial locality of reference.

Since the characteristics of access patterns for instructions and data are typically different, a split-cache organization that decouples data access from instruction access has been used to yield more efficient access schemes. This is because the designer can now take advantage of the particular characteristics of the access patterns and design the cache memory systems accordingly. Several recent microprocessors have implemented the split-cache approach (MC 68030, MC 68040); while some have adopted the unified cache scheme as well (Intel 80486).

Cache memory design for mainframes and minicomputers has been extensively studied since IBM introduced the first commercial cache in its System 360, Model 85. Hill and Smith [9] carried out a study on on-chip caches with emphasis on sector cache using miss and traffic ratios as performance metrics, but they performed no studies on prefetching. A performance evaluation of on-chip cache organizations is given by Eickemeyer and Patel [6] where they compare different types of caches such as instruction, data, split, unified, stack, and top-of-stack caches. Farrens and Pleszkun [7] present a combination of an instruction cache, instruction queue and instruction queue buffer and evaluate a set of instruction fetch strategies.

In this paper, we study the instruction buffers and instruction caches of several existing computers and CISC and RISC microprocessors. We first present a classification of the different buffering and caching approaches. Then we describe the prominent features of several existing systems and also of a proposed queue-based instruction cache scheme [4]. We then detail the simulation methodology, propose a performance metric and present the simulation results.

*This research was supported in part by the National Science Foundation under grant MIP-8912455.

2 A Classification Scheme

The instruction caches/buffers incorporated in various commercial computers and microprocessors share certain characteristics, but differ in several design aspects. Any classification of these buffering or caching schemes will depend to some extent on subjective distinctions and often tend to be vague and overlapping. These classifications may be subject to varying interpretations and it may not always be clear, to which class a particular scheme belongs. Recognizing this, we now turn our attention to classifying instruction buffers/caches of existing machines based on major design aspects. Very broadly, we classify the instruction buffering schemes into three categories:- (i) Instruction Buffers (ii) Instruction Caches and (iii) Prefetch Queues.

The first category in our classification is instruction buffers. By instruction buffers, we mean the traditional instruction buffers as found in the earlier machines such as the IBM 360/91, CDC STAR-100, and CDC 6600 which typically contain one contiguous segment of the program (or a single locality). A recently proposed queue-based cache scheme [4] also shares several characteristics with these buffers and hence we include it also in this category. These instruction buffers are often designed as a queue or stack. Instructions are prefetched in consecutive order and executed until a successful branch occurs. At the point of a successful branch, if the branch target is not in the buffer, then the prefetch buffer is cleared and refilled with instructions which start at the branch address. In most of the traditional instruction buffers, there is an emphasis on prefetching and thus capturing more of spatial locality. The queue-based organization deviates here. It has more emphasis on retaining whatever has already been fetched and thus temporal locality is captured primarily. Also, in the queue-based cache, prefetching is done only when the bus is available and with a lower priority than operand fetch during instruction execution. There are two major factors affecting the success of such instruction buffers, average size of a locality and scattering of localities. The buffer should be larger than the average size of a locality. This is because if the average loop size is greater than that of the buffer, the buffer loses its effectiveness. Since these buffers can contain only one contiguous segment of the program, scattered locality characteristics exhibited by many programs may also cause loss of effectiveness.

The second category in our classification is instruction caches. By instruction caches, we mean buffers that are managed more like a conventional cache, in either a direct-mapped, set-associative, or fully associative mapping scheme. Several recent CISC and RISC microprocessors have this type of fairly conventional instruction cache. We also include the Cray instruction buffers in this class. They are not exactly conventional caches, but since they are managed more similarly to caches than the traditional instruction buffers and since they can contain more than one contiguous program segment (four in Cray-1, Cray X-MP and Cray Y-MP and eight in Cray-2), it seems more appropriate to include them in this category. At the time the Cray–1 was designed, virtual memory and instruction caches were receiving significant attention. This influence is apparent in the techniques employed in its instruction buffers. Cray–1's instruction buffer design is closer to today's cache memories than it is to the CDC 6600 or the IBM 360/91 buffer designs [16].

The third category in our classification is prefetch queues. The Intel microprocessors 8086, 8088, 80186, 80286 and the Motorola 68000 contain prefetch queues. These prefetch queues are considerably different from the traditional queue-based instruction buffers mentioned before, the major distinction being that these prefetch queues do not capture locality. Instructions are prefetched, whenever the bus is free, and held in the queue, but they are *consumed* by the CPU as they are executed. In other words, the locality is not captured. The MC68010 has an instruction prefetch queue that will support a loop mode of operation, i.e. capture of locality. The queue is two words or four bytes long as in the MC68000, but in the event of a loop that can be contained in the queue, the MC68010 has the ability to enter a loop mode, suppressing all further opcode fetches until an exit loop condition is met. It should be noted that not all instructions are loopable and that only very small loops can be accomodated.

2.1 Instruction Buffers

In this subsection, we describe a few typical implementations of the traditional type of instruction buffers and the recently proposed queue-based instruction cache.

The CDC 6600: The CDC 6600 was the first commercially available computer whose architecture addressed the speed disparity between the central processor, main memory, and I/O devices. One of the techniques used by the CDC 6600 to maximize the rate at which instructions are executed is to employ an *instruction stack* that provides more rapid access to instructions. The instruction stack consists of eight 60-bit registers which hold the instructions most recently executed. As instructions are fetched, they are sent to the stack's *input register*. Immediately upon entering the input register, the first or the leftmost instruction in the 60-bit word is transferred to a series of instruction registers. As this transfer occurs, another instruction fetch is initiated. The condition for this initiation is simply that the left-most instruction is transferred for execution [23]. Instructions continue to be fetched sequentially from central storage until a branch causes transfer of control. When the execution of an instruction causes a branch back to an instruction that is currently in the stack, no refills are allowed after the branch; thereby holding the program loop in the stack. The CDC 7600 instruction buffer is similar to the CDC 6600 buffer, the only difference being that it consists of twelve 60 bit registers instead of the eight in CDC 6600.

The IBM 360 Model 91: The IBM System 360/91 architecture includes an instruction unit that employs an *instruction prefetch* mechanism. The goal of this mechanism is to assure that future instructions are available for processing as required. These instructions are held in an instruction stack that has eight double-word (64 bit) buffers. When the instruction stream is sequential, the instruction fetch mechanism operates as a fetch-ahead unit and attempts to maintain a supply of instructions in the instruction buffer in advance of need. Whenever the number of instructions in the buffer is below three double words beyond the current instruction, a new prefetch is initiated. If storage is not busy with operand fetching, a fetch for a fourth double word is also initiated. If the number of instructions in the buffer is below three double words beyond the current instruction, memory access conflicts are resolved in favor of instruction prefetch; otherwise it favors operand fetch. Thus, generally the buffer holds 24 or 32 bytes beyond the current instruction. The appearance of a branch instruction signals a point in the instruction stream where sequential flow may be altered. A successful branch causes the startup of a new instruction stream sequence; and if the target address is not already in the buffer then the processing of the new stream is delayed until the instruction buffer is replenished. Forward branches with target addresses within the instruction buffer array are satisfied from the instruction buffer. Back branches that are within eight double-words from the current instruction trigger a special processing mode called *loop mode*. In this case, the entire loop can be contained within the instruction buffer array. The instruction buffer array is supplemented by two double-word registers, also called the *branch target buffer*. These two registers are used when a conditional branch is encountered. Both potential instructions to be executed next are thus made available. When the decision is known, one of them is used to supply the instruction and the other is flushed.

The Queue-Based Instruction Cache: The queue-based instruction cache proposed in [4] is based on a RAM queue and a set of registers. The data is loaded from main memory into the RAM queue in a FIFO fashion. However, the CPU may read any of the words currently in the queue and in any order. So the queue approach concerns only the way data is loaded into the cache. Unlike other cache designs, there is no concept of a line (block). The whole RAM is used as one block (or segment) and the address space of the resulting cache is contiguous.

The operation of the cache depends on three load procedures. A *LoadNew* procedure is used to load data into an initially empty cache or to clear an old cache and reload it with new data. To load data sequentially from main memory into a non-empty cache, a *LoadSequential* procedure is used. One major feature of the queue-cache is *preloading*. Preloading means that when the CPU requests information from an address that is not in the queue, but not too far from the last address loaded into the queue, then the referenced data as well as the intermediate addresses are fetched and appended to the queue without clearing the queue. Whenever a forward branch is encountered, the control logic checks to see how far ahead the new branch location is from the last address loaded in the queue. If this distance is within a predefined *preload window*, then the *cache preload procedure* is invoked. Preloading thus prevents unnecessary flushing of the queue, which occurs in the traditional instruction buffers.

2.2 Instruction Caches

In this subsection, we examine the instruction buffers of the Cray computers, the instruction caches in several Motorola processors, and the instruction cache of the Intel RISC processor 80860.

The Cray series: The Cray–1 has four instruction buffers each of which can contain 128 bytes. 128, allowing the buffers to map up to 512 bytes of contiguous memory. Associated with each instruction buffer is a beginning address register (BAR) which holds the address of its first location. When an instruction is referenced, the four BARs are compared with the PC. If there is a match, the next instruction register is loaded from the appropriate instruction buffer. Forward and backward branches are accomodated. There is no storage access delay when instruction sequences are found in the instruction buffers. A two cycle delay ensues when the next instruction is found in a buffer different from the one currently in use. If there is no match, the next instruction must first be fetched from memory and placed in an instruction buffer before instruction processing can proceed. The instruction buffers are used in rotation. The least recently filled instruction buffer is selected for use when the next instruction is not already in a buffer, analogous to a FIFO replacement strategy in caches. The Cray-1 is capable of fetching 128 bytes of data in one memory cycle or 4 processor clock cycles.

Similar to the Cray-1, the Cray X-MP uses four large independent instruction buffers which play the role of an instruction cache. Each buffer contains 32 words of 64 bits, for a total buffer size of 256 bytes. In fact, these buffers are managed like a 1024 byte fully associative cache memory with a FIFO replacement scheme. The operation of these buffers is identical to those of Cray-1, the differences being only in the access times associated with a miss in the buffer. This is due to the difference in the speeds of the memory used in the Cray-1 and the Cray X-MP. The Cray X-MP has a 9.5 ns processor cycle and a 135 ns memory cycle. Hence, 14 clock cycles are required for one memory access whereas 4 cycles would suffice in the Cray–1. If a change of buffer is required, a 2 cycle delay is incurred just as in Cray-1. If the instruction is not in any buffer, then 16 clock cycles are added [15]. The Cray Y-MP also has the same type of buffers as the Cray X-MP. The Cray-2 has 8 instruction buffers, each 128 bytes long.

The Motorola series:

The first 32-bit generation Motorola processor, the MC68020, incorporates a 256-byte on-chip instruction cache memory used to store the instruction stream prefetch accesses from the main memory. It is organized as a direct-mapped cache of 64 long word entries [14]. Each cache entry consists of a tag field, a valid bit, and 32 bits of instruction data.

The MC68030 microprocessor includes a 256-byte on-chip instruction cache organized as a direct mapped cache of 16 lines [14]. This 32–bit processor from Motorola has in addition to this instruction cache, a data cache of equal size. In this study, however, we are interested only in the instruction cache. Each line of the instruction cache consists of four entries and each entry contains four bytes. The tag field for each line contains a valid bit for each entry in the line and each entry is independently replaceable. All four entries in a line have the same tag address. The MC68030 instruction cache operates almost identically to that of the MC68020, except for the cache load/fill procedure. The cache memory can be filled in a single entry mode or as a burst fill, when a miss occurs. In the single entry mode, four longwords are loaded into the cache, one longword at a time. This mode uses asynchronous data transfer requiring four times the time needed to fetch one longword, whereas in the burst mode four long words are transferred in a burst in less than twice the time required to fetch one longword.

The next 32 bit processor from Motorola, the MC68040, contains a 4K-byte on-chip instruction cache configured as a four-way set associative cache of 64 sets of four 16 byte lines [14]. Each cache line contains an address tag, a valid bit and four longwords of instruction data. Since entry validity is provided on per–line basis, an entire line must be loaded from system memory in order for the cache to store an entry. Only burst mode accesses that successfully read four longwords can be cached. Memory devices unable to support bursting forces the processor to complete the access as a sequence of longwords. For prefetch requests that hit in the cache, two longwords are multiplexed onto the internal instruction data bus. When an access misses in the cache, the cache controller requests the memory line containing the required data from memory and places the line in the cache. If all the lines in the set are already valid, a pseudo random replacement technique is used to select one of the four lines and replace the tag and instruction data contents of the line with the new line information. To implement this replacement algorithm, each cache contains a 2-bit counter which is incremented for each access to the cache. The counter is incremented for each half line accessed in the instruction cache. When a miss occurs and all four lines in the set are valid, then the line pointed to by the current counter value is replaced, and the counter is incremented.

Intel 80860:

Intel's powerful 64-bit RISC microprocessor, the 80860 has both an instruction cache and a data cache. The instruction cache is a two-way, set associative memory of 4 Kbytes, with 32-byte blocks [11]. This processor supports accesses up to a maximum of 64-bits per clock cycle.

2.3 Prefetch Queues

On early microprocessor chips, both Intel and Motorola used prefetch queues since semiconductor technology did not permit integration of on-chip caches at that time. The features and operation of these prefetch queues are discussed below.

The Intel 8086, 80186 and 80286 have a 6-byte first-in-first-out (FIFO) prefetch queue that is continually filled whenever the system bus is not needed for some other operation. This look-ahead feature can significantly increase the CPU's throughput because, much of the time, the next instruction would already be in the CPU when the present instruction completes its execution. If a branch is taken, then the instruction queue is flushed and there is no time saving; but on the average this occurs only a small percent of the time. During the execution of computationally intensive instructions, for instance, the multiplication instruction in the 8086, the CPU will not be using the bus and there are ample clock cycles to fill the instruction queue. When there is a branch to an odd address, the 8086 brings in 1 byte and continues with even address words. After five of the six bytes in the queue are full (when just one byte is left empty), the next fetch will not begin until a word is available in the queue [12].

The 8088 has only a 4-byte instruction queue instead of the 6-byte queue in the 8086. The reason for the smaller queue is that the 8088 can fetch only 1 byte at a time (since it has only an 8-bit data bus) and the longer fetch times mean that the processor cannot fully utilize a 6-byte queue [12]. The prefetching algorithm is different from the 8086 in the sense that instead of waiting for a 2-byte space, the 8088 will initiate a fetch even when a single empty byte in the queue becomes available.

The MC68000 uses a two-word tightly–coupled instruction mechanism to enhance performance. When execution of an instruction begins, the operation word and the word following it would have already been fetched, with the operation word in the instruction decoder. In the case of multi-word instructions, as each additional word of the instruction is used internally, a fetch is made to the instruction stream to replace it. The last fetch from the instruction stream is made when the operation word is discarded and decoding is started on the next instruction. If the instruction is a single-word instruction causing a branch, then the second word is not used. Since this word is fetched by the previous instruction, it is impossible to avoid this superfluous fetch. In the case of an interrupt or trace exception, both words are discarded.

3 Performance Study

The performance models developed in this section are used along with the statistics generated by our simulators to quantify the performance enhancement that can be achieved. The fraction of the total references that result in a hit in the buffer/cache reflects the success of the caching scheme, since the processor infrequently waits for slower main memory references. However, it is observed from our simulations as well as from previous reports [1], that many configurations that provide a higher hit in the cache would actually degrade the performance of the CPU because of higher miss penalties (which is often the case with larger block sizes). Hence hit ratio may not be a good performance measure.

Effective access time is a much better metric to indicate performance. But this parameter depends on the access times of the memory used for the cache and the main memory and thus depends on the particular implementation. In this study, we isolate the implementation dependent parameters and arrive at an implementation independent performance index which we denote as the speed-up factor.

The speed–up that can be obtained by a traditional type of instruction buffer is derived as follows : Let T_{CACHE} denote the cache (instruction buffer) cycle time and T_{MM} denote the access time of the RAM used for main memory. Furthermore, let $p(miss)$ denote the probability of a miss and let $p(prefetch)$ denote the probability of initiation of an *expensive prefetch*. Some prefetches are made only when the bus is free and hence they do not constitute any overhead. However, prefetches made with greater priority than operand fetches do impose severe penalties and we call them *expensive prefetches*. Also let Grain1 represent the number of accesses performed for a miss and Grain2 denote the number of accesses done at a prefetch. The effective access time T^{IB}_{EFF} is

$$T^{IB}_{EFF} = T_{CACHE} + [p(miss) * Grain1 + p(prefetch) * Grain2] * T_{MM}. \tag{3.1}$$

For the CDC 6600 buffer, both Grain1 and Grain2 are equal to 1 while for the IBM 360/91, Grain1 is 2 and Grain2 is 1.

Speed–up of any cache organization may be defined as the percent improvement in the effective access time by using the cache. In other words, the speed–up is

$$S = \frac{T_{MM} - T_{EFF}}{T_{MM}} = 1 - \frac{T_{EFF}}{T_{MM}} \tag{3.2}$$

where T_{EFF} is the effective access time of the memory system with a cache. From equations 3.1 and 3.2, the speed–up S_{IB} of the traditional instruction buffer is

$$S_{IB} = 1 - \frac{T_{CACHE}}{T_{MM}} - p(miss) * Grain1 - p(prefetch) * Grain2 \tag{3.3}$$

Now if we let R denote the ratio of the access times of the cache to that of the main memory, then equation 3.3 can be rewritten as

$$S_{IB} = 1 - R - p(miss) * Grain1$$
$$-p(prefetch) * Grain2 \quad (3.4)$$

and

$$S_{IB} = X_{IB} - R \quad (3.5)$$

where

$$X_{IB} = 1 - p(miss) * Grain1$$
$$-p(prefetch) * Grain2 \quad (3.6)$$

In equation 3.5, S_{IB} is the speed–up and X_{IB} is the speed–up factor. The speed-up factor is independent of the implementation dependent parameter R. The probability of a miss and that of a prefetch are among the statistics generated by our simulator and hence we calculate the speed-up factor from this equation.

Now we turn to the queue-cache organization. Again, let T_{CACHE} denote the cache cycle time, T^{QB}_{MAIN} denote the time required to retrieve a word from main memory upon a miss, and T_{MM} denote the access time of the main memory. Furthermore, let $p(loadnew)$ denote the probability of a call to the *loadnew* procedure upon a miss and let $AvePreDis$ denote the average preload distance upon a call to the *preload* procedure. (These terms were explained in Section 2.1 where we detailed the operation of the queue-cache.) Then the effective access time T^{QB}_{EFF} is

$$T^{QB}_{EFF} = T_{CACHE} + (1 - H_{QB})T^{QB}_{MAIN} \quad (3.7)$$

where

$$T^{QB}_{MAIN} = p(loadnew) * T_{MM}$$
$$+(1 - p(loadnew)) * AvePreDis * T_{MM}. \quad (3.8)$$

The speed–up of the queue-based cache is

$$S_{QB} = 1 - \frac{T_{CACHE}}{T_{MM}} - (1 - H_{QB})[p(loadnew)$$
$$+(1 - p(loadnew))AvePreDis]. \quad (3.9)$$

Proceeding as in the case of the traditional buffers,

$$S_{QB} = X_{QB} - R \quad (3.10)$$

where X_{QB}, the speed–up factor of the queue-cache is,

$$X_{QB} = 1 - (1 - H_{QB})[p(loadnew)$$
$$+(1 - p(loadnew))AvePreDis]. \quad (3.11)$$

For the next category, instruction caches of the conventional type, let H_{BLOCK} denote the hit ratio for the block cache and K be the block size. Then the effective access time is

$$T^{BLOCK}_{EFF} = T_{CACHE} + (1 - H_{BLOCK})\ K\ T_{MM} \quad (3.12)$$

The speed–up can be calculated from equation 3.2 and 3.12 as

$$S_{BLOCK} = 1 - \frac{T_{CACHE}}{T_{MM}} - (1 - H_{BLOCK})\ K \quad (3.13)$$

and separating R, the ratio of the cache and main memory access times, the speed-up factor is

$$X_{BLOCK} = 1 - (1 - H_{BLOCK})\ K \quad (3.14)$$

For our analysis, the hit ratio is obtained from the simulator output and the speed–up factor is calculated using equation 3.14.

Finally we turn to the prefetch queues. Let T_{CACHE} denote the prefetch queue cycle time and $p(miss)$ be the probability of a miss. There are no *expensive prefetches* since all prefetches are made when the bus is free and hence they do not constitute any overhead. Intel literature specifically mentions that prefetches are made only when the bus is free, and even for the MC68000, we make the assumption that operand fetches required during execution take precedence over instruction prefetches. (If this is not the case, the prefetch queue might constitute an overall penalty rather than a performance improvement.) In all the prefetch queues we studied, the fetch granularity is whatever is directly supported by the bus-width. The effective access time T^{PFQ}_{EFF} of these prefetch queues is

$$T^{PFQ}_{EFF} = T_{CACHE} + p(miss) * T_{MM}. \quad (3.15)$$

From equations 3.2 and 3.15 we obtain the speed-up as

$$S_{PFQ} = 1 - \frac{T_{CACHE}}{T_{MM}} - p(miss) \quad (3.16)$$

and after isolating the access time ratio as in the above cases, the speed-up factor may be obtained as

$$X_{PFQ} = 1 - p(miss). \quad (3.17)$$

Thus we observe that the hit ratio in prefetch queues is a direct measure of performance improvement obtained by the scheme. Since the access time ratio R is the same for all systems we can simply compute the speed-up factors and compare them.

4 The Simulation

Trace–driven simulation, which has become the standard cache performance evaluation method, was used in this study. Using a trace permits simulation of caches with many strategies and parameter values in reproducible experiments [1]. A trace-driven simulation is guaranteed to be representative of at least one program in execution [9]. We gathered the address traces used in this study from real programs executed on a VAX 11/785 architecture running the UNIX operating system. The traces were obtained by executing the C Compiler (compiling a program of about 1000 lines of code), the Tex type–setter program (compiling a text file of about 41 Kbytes with several equations) and the Whetstone benchmark. The C Compiler trace has 9222 memory references, the tex trace has 88473 references and the whetstone trace has 528024 references. The C Compiler trace and Tex trace are real program traces and are likely to contain genuine embedded correlations that synthetic benchmarks often lack. For synthetic traces, we observe that the results are extremely sensitive to minor system parameter changes, whereas the range of variation is less wide and more regular for real program traces. We developed trace-driven simulators for the various systems under study.

Simulation Aspects: Our study is an evaluation of the instruction buffering mechanisms only, not of the systems in which they are used. Hence we do not attempt to exercise the various features of the processors or computers, but instead, restrict the evaluation to a general environment with the bus width and access granularity features of the original machine. Any evaluation isolating these features would not be realistic, since for each machine, the cache or buffer size, prefetch threshold, block size and other characteristics were determined by its designers with due concern for these features.

The Cray machines achieve a massive memory bandwidth with the help of interleaving. For instance, the Cray-1 is 16 way interleaved, the Cray X-MP/Model24 is 32 way interleaved and each module has a data width of 8 bytes. The bus width and data transfer granularity of the various processors/computers we studied range from 1 byte to 256 bytes. Considering these aspects, comparing the different schemes in the different machines is not a trivial task; only the comparison of systems that operate with an identical bus width would yield meaningful results. Therefore we compare each cache or buffer under study to other caches capable of fetching identical amounts of data in unit access and to a queue-based cache that holds the same amount of data. Thus the IBM 360/91 and CDC 6600 are compared with a queue-cache of size 64 bytes (considering 60 bit words of the CDC as analogous to 8 bytes of others). Similarly, the MC68020 and MC68030 are compared with a 256-byte queue-cache.

Figure 1: Comparison of Hit Ratios

Figures 1 to 6 illustrate the various comparisons. The queue-cycle or cache cycle in these plots denotes the number of processor clock cycles needed to perform one memory access. This parameter is significant only for some traditional types of buffers, prefetch queues and the queue-cache. In these schemes, the combination of this parameter with the execution times of instructions determines the amount of possible "free" prefetching. The CDC 6600 and IBM 360/91 buffers consistently exhibit higher hit ratios than the queue-based cache (Figure 1) , but when the speed-up factor is evaluated (Figure 2) the queue-cache displays better performance because the other two buffers perform a number of superfluous prefetches. The branch target buffer of the IBM 360/91 improves the hit ratio, but it also increases the traffic ratio significantly and often detrimentally affects the system. The Intel 80860 RISC processor instruction cache results also appear in Figure 2 along with those of IBM 360/91 and CDC 6600 buffers because of identical access granularity. But it has to be always kept in mind that the Intel 80860's cache is 64 times larger than the other buffers and has a much more complex control circuitry and a higher directory cost than the others.

We were not able to plot the results of Cray-1 and Cray X-MP on the same graph because of aforementioned data bandwidth differences, and they appear in Figures 3 and 4 along with queue-caches of equiv-

Figure 4: Cray X-MP and the Queue-Cache

Figure 2: IBM 360/91, CDC 6600 and Intel 80860 instruction buffers/caches

Figure 3: Cray-1 and the Queue-Cache

Figure 5: Motorola Processor Caches and the Queue-Cache

Figure 7: Speed-up factor vs Instruction buffer size

Figure 8: Speed-up factor vs Prefetch Threshold

Figure 6: Prefetch Queues of Intel 8086, MC68000 and the Queue-Cache

alent size. Judgements and inferences from Figure 5 should be made keeping in mind the fact that the MC68040 cache is 16 times larger than the MC68020 and MC68030 caches.

The best case for the MC68030 is obtained by assuming that all cache fills are done in burst mode and the worst case assumes all fills in single entry mode. The results of MC68040 are obtained without considering burst-mode filling. We observe from Figures 2 and 5, that an increase in size from 256 bytes to 4Kbytes for the queue–cache increases the performance only very nominally. Results from prefetch queue simulations are illustrated in Figure 6.

We also performed studies that detailed how an increase in the buffer size would affect the performance. Figure 7 displays the results. Here we observe that for traditional instruction buffers of the type we studied, an increase in size beyond 256 bytes is only of marginal improvement. This can be explained on the basis of locality characteristics and average loop sizes of programs. Studies of program behaviour [2] show that 60.7% of all branches are within 256 bytes from current point of execution; only 7.2% more jumps fall within 512 bytes, and just 4.0% more in 1024 bytes. Again, since these buffers contain only one contiguous segment of the program, the scattered locality characteristics exhibited by many programs cannot be captured.

The unfavorable effects of excessive prefetching are highlighted by Figure 8. Simulations were performed by changing the prefetch threshold of an IBM360/91 type instruction buffer. The IBM design has a prefetch threshold of three double words or 24 bytes. We varied this threshold and performed simulations. It was observed that prefetching often increases the hit ratio, but when the overall performance of the system is analyzed, excessive prefetching is often detrimental.

We also performed simulations to study the ef-

Figure 9: Prefetch Queue Performance vs Queue Size

fect of cache size, line size and associativity on conventional instruction caches. The results are in concurrence with published results [18] and hence we do not report them here. Prefetch queues are inherently small in all implementations which is justified from the results in Figure 9 as well. Large prefetch queues tend to be under–utilized. In Figure 9, the performance improvement saturates with fairly small queues, especially for the lower bus-width case. The queues are better utilized with higher bus-width, however, the speed-up factor still saturates sooner or later. It may be noted that in this figure, a queue-size of 4 bytes for the 1-byte bus–width case represents the Intel 8088 microprocessor queue.

5 Conclusions

In this paper, we have presented a classification and performance evaluation of several on-chip and mainframe/minicomputer instruction caches and buffers. We presented a performance metric that does not depend on implementation dependent parameters and used it in our studies. We compared the existing caches with a proposed queue-based caching approach. Among the three categories we studied, prefetch queues are the simplest and fully associative caches are the most complex. In situations with no restrictions on chip area or directory cost, fully associative or set associative caches that offer better performance than equally large traditional instruction buffers/queue-cache may be used. But larger instruction caches do not necessarily provide a significant improvement in performance, a phenomenon reported by [7] also. The queue-based cache performs better than the implemented approaches when small caches are considered, whereas with larger caches the performance of the queue-cache is either comparable or inferior. Hence this approach is suitable for emerging technologies such as Gallium Arsenide which promise higher speed but currently prohibit reaching high circuit densities. We also conclude that the queue-based cache is not suitable for machines like the Cray which fetch large amounts of data in one access. But such methods of reducing the speed disparity are limited to mainframes and are not feasible in a single-chip environment. Multiple queues can capture scattered locality and might offer good performance in comparison to the large set-associative instruction caches and the Cray buffers. This is an area of study that needs to be investigated. Traditional instruction buffers and the queue-based caches have significantly lower directory overhead compared to equal data area set associative or fully associative caches, and hence consume less chip area. Instruction buffers with less prefetching and the queue-cache offer good performance where constraints of chip area exist and only small caches are feasible. Another conclusion is that prefetching can reduce effective memory access time if carefully designed and implemented, however there is a very good chance that the potential gain in performance is heavily diminished by the disproportionate increase in the number of fetches performed.

References

[1] Alpert, D.B. and M.J. Flynn, "Performance Trade–offs for Microprocessor Cache Memories," IEEE Micro, August 1988, pp. 44–53.

[2] Alexander, W.G. and D.B. Wortman, "Static and Dynamic Characteristics of XPL Programs," Computer, Vol. 8, No. 11, November 1975, pp. 41–46.

[3] Anderson, D.W., Sparacio, F.J., Tomasulo, R.M., "Machine Philosophy and Instruction Handling," IBM Journal of Research and Development, 11, No.1. 8-24, January 1967.

[4] Coraor, L.D., P.T. Hulina, and D.N. Mannai, "A Queue–Based Instruction Cache Memory," Proc. of the Int. Symp. on Computer Architecture and Digital Signal Processing, October 1989, Hong Kong.

[5] Ditzel, D.R., "Program Measurements on a High–Level Language Computer," Computer, Vol. 13, No. 8, August 1980, pp. 62–72.

[6] Eickemeyer, R.J. and Patel J.H. , "Performance Evaluation of On-Chip Register and Cache Organizations", Proc. 15th Int. Symp. on Computer Architecture, 1988, pp. 64–72.

[7] Farrens, M.K. and Pleszkun, A.R., "Improving Performance of Small On-Chip Instruction Caches, " Proc. 16th Int. Symp. on Computer Architecture, 1989, pp. 234–241.

[8] Goodman, J.R. and Wei–Chung Hsu, "On the Use of Registers vs. Cache to Minimize Memory Traffic," Proc. 13th Annual Int. Symp. on Computer Architecture, June 1986, pp. 375–383.

[9] Hill, M.D. and A.J. Smith, "Experimental Evaluation of On–Chip Microprocessor Cache Memories," Proc. 12th Annual Int. Symp. on Computer Architecture, June 1985, pp. 55–63.

[10] "i486 Microprocessor Programmer's Reference Manual", Intel Corp., 1989.

[11] "i860 Microprocessor Programmer's Reference Manual", Intel Corp., 1989.

[12] Liu, Y.C and Gibson G.A. , "Microcomputer Systems: The 8086/8088 Family – Architecture, Programming and Design", Prentice Hall, Inc. Englewood Cliffs, N.J. 07632, 1984.

[13] Laha, S., et.al., "Accurate Low–Cost Methods for Performance Evaluation of Cache Memory Systems," IEEE Transactions on Computers, Vol. C–37, No. 11, November 1988, pp. 1325–1336.

[14] MC68xxx Microprocessor User's Manuals, Motorola Inc.

[15] Robbins, K.A. and Robbins, S. , " The Cray X-MP/Model 24," Lecture Notes in Computer Science, no.374, Springer-Verlag, 1989.

[16] Schneck, P.B., "Supercomputer Architecture," Kluwer Academic Publishers, 1987.

[17] Smith, A.J., "Sequential Program Prefetching in Memory Hierarchies," IEEE Computer, December 1978, pp. 7–21.

[18] Smith, A.J., "Cache Memories," ACM Computing Surveys, Vol. 14, No. 3, September 1982, pp. 473–530.

[19] Smith, A.J., "Line (Block) Size Choice for CPU Cache Memories," IEEE Transactions on Computers, Vol. C–36, No. 9, September 1987, pp. 1063–1075.

[20] Smith, A.J., "Cache Memory Design: An Evolving Art," IEEE Spectrum, December 1987, pp. 40–44.

[21] Smith, J.E. and J.R. Goodman, "Instruction Cache Replacement Policies and Organizations," IEEE Transactions on Computers, Vol. C–34, No. 3, March 1985, pp. 234–241.

[22] Strecker, W.D., "Transient Behaviour of Cache Memories," ACM Transactions on Computer Systems, Vol. 1, November 1983, pp. 281–293.

[23] Thornton, J.E.,"Design of a Commputer: The Control Data 6600," Scott, Foresman and Company, Glenview 1970.