Improving Performance of Small On-Chip Instruction Caches

Matthew K. Farrens

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

Andrew R. Pleszkun

Department of Electrical and
Computer Engineering
University of Colorado-Boulder
Boulder, CO 80309-0425

Abstract

Most current single-chip processors employ an on-chip instruction cache to improve performance. A miss in this instruction cache will cause an external memory reference which must compete with data references for access to the external memory, thus affecting the overall performance of the processor. One common way to reduce the number of off-chip instruction requests is to increase the size of the on-chip cache. An alternative approach is presented in this paper, in which a combination of an instruction cache, instruction queue and instruction queue buffer is used to achieve the same effect with a much smaller instruction cache size. Such an approach is significant for emerging technologies where high circuit densities are initially difficult to achieve yet a high level of performance is desired, or for more mature technologies where chip area can be used to provide more functionality. The viability of this approach is demonstrated by its implementation in an existing single-chip processor.

1. Introduction

In recent years, advances in VLSI technology have significantly increased the speed at which a single-chip processor (SCP) can be run. As was the case with the first mainframes, raw processing speed is now much greater than memory speed. Therefore, it behooves us to examine what the designers of the first high-performance computers did to minimize the negative impact of memory latency on processor performance and attempt to incorporate some of their methods with ours.

One important difference between the problems faced by mainframe designers and architects of single-chip processors is the adverse effect of off-chip bandwidth and pin limitations on SCP performance. Due to these limitations, certain mainframe approaches to reducing memory latency, such as a massive increase in the memory bandwidth, may not be available in the SCP environment. Other techniques, however, such as the incorporation of more pipelining in the processor, the use of queues between the processor and memory, and making wide use of caches, are still applicable.

Programs generate two different types of memory requests, requests for instructions (I-Fetches) and requests for data (D-Fetches). Both types of requests are competing for the same resource - memory. Mainframe designers developed techniques to reduce the impact of this competition, such as supplying separate data and instruction caches, and allowing multiple outstanding memory requests. Due to physical space limitations in the SCP realm, it is not practical to supply separate *on*-chip data and instruction caches. Furthermore, supplying separate *off*-chip data and instruction caches would require extra I/O pins that may not be available. Therefore, given the inherent spatial and temporal locality exhibited by instructions, SCP on-chip caches are generally instruction caches, used to service I-Fetches. This permits the available off-chip bandwidth to be utilized for servicing D-Fetches. The use of an on-chip instruction cache has been suggested by others [PaSe80,SmGo85,Smit82], and has been already incorporated in several designs [ACHA87,BCDF87,KMOM87].

While a simple on-chip instruction cache will provide a significant increase in performance, some competition for external memory between I-Fetches and D-Fetches will still exist. This is because, even in mature technologies, physical limitations prevent extremely large on-chip caches. In addition, for new and emerging technologies that promise increased speed, the high densities needed to support even moderately sized caches are not available. In this paper we are interested in approaches that minimize the impact of this competition for external memory and provide performance equivalent to that provided by much larger instruction caches.

The remainder of this paper is divided into 6 sections. The next section presents a short discussion of I-fetch and its interaction with D-fetch. Section 3 gives a brief description of the PIPE processor which is used as a basis for our simulations. Next, we contrast the PIPE approach to I-fetch with a conventional instruction cache approach. In Section 5, simulation details are provided and in Section 6, a discussion of our simulation results is presented. Finally, Section 7 presents our summary and conclusions.

2. Instruction and Data Fetch Strategies

Most current processor designs assume the presence of a relatively large external cache, and that accessing that cache can be done quickly. An external cache access is typically associated with a stage of the pipeline and involves broadcasting a request off-chip and then latching the data item (in the case of read) when it is returned by the cache. Cleary, the off-chip communications and cache memory access activities associated with such a strategy can have a dramatic, limiting impact on a processor's performance. If a processor's design is too closely tied to the performance of the external cache, implementing the processor in a faster technology will not necessarily result in a faster system. Techniques to minimize this interdependence between processor and memory will now be presented.

2.1. Instruction Fetch

In the mid 1970's, Rau and Rossman [RaRo77] studied the instruction fetch strategies used by the IBM System/370 series, the

CDC 6600, and the Manchester University MU5. This study examined the use of Prefetch Buffers in conjunction with an Instruction Buffer (an instruction cache). In their model of instruction fetch, the decode logic takes instructions directly out of the Prefetch Buffers, which are loaded with as many sequential instructions as possible given the size of the buffers, the size of the instruction cache, and the speed of external memory. Their results showed that a reduction of up to 50% in average I-Fetch delay can be achieved by the use of these buffers. While the results indicated that, within certain bounds, better performance can be achieved by using more buffers, the results also indicated that increasing the number of Prefetch Buffers increases memory traffic. Since the penalty for going off-chip in an SCP environment is higher than in a mainframe, a balance must be struck between the number of Prefetch Buffers and the amount of off-chip accessing these buffers generate. A similar study by Grohoski and Patel [GrPa82] included the effect of operand accessing on program performance and found similar results.

The use of a Target Instruction Buffer (TIB) was also examined in both of the above studies, as well as one by Hill [Hill87]. A TIB can be used in place of or in addition to an instruction cache, and contains the n sequential instructions stored at a branch target address. (n is a function of the TIB size.) When a branch is taken, the n instructions are taken out of the TIB while the I-Fetch control logic issues requests for the instructions sequential to the ones in the TIB. If there are more instructions in the TIB than the number of clock cycles it takes to access external memory, the instruction stream will have no gaps in it. The AMD29000 [Adva87] uses such a TIB instead of an instruction cache. While the results of the studies indicate that a small TIB can provide better performance than a simple small instruction cache, the use of a TIB implies large amounts of off-chip accessing, which again can be a problem in SCP design.

2.2. Data Fetch

There are several ways to reduce the effective access time of data references. One technique is to treat the external memory as a functional unit, and schedule arrivals from memory. Some Load/Store architectures [HCSS87] employ a version of this technique by providing a delay slot after a load that can be filled with an instruction that will execute while the load is completing (in essence treating the external memory as a functional unit with access time of 2 clock cycles). The obvious drawback to this method is that the architecture is tied directly to external factors such as memory speed.

Another technique is to provide queues, either explicitly architectural or transparent to the user, that allow the machine to continue executing instructions while waiting for the memory request to be serviced. This method has the advantage of making the architecture independent of memory speed. The IBM 801 [Radi82], for example, provides what is in effect a single element transparent queue that allows instructions after a load that do not use the requested data to continue to issue. This machine only blocks issue when an instruction needs to use data that has not yet been returned from memory. However, since only a single element queue is used, even if memory is pipelined only one memory request can be outstanding at a time.

Making the queues part of the architecture and visible to the programmer permits the easy overlap of memory activities with program execution. If the memory is pipelined, several memory requests can be outstanding at the same time. In addition, the use of architectural queues allows requests generated by the instruction fetch unit to take precedence over data requests with a limited impact on performance. In a processor without queues, a data request is issued very near the time the data is required. If an instruction request interferes with this data request, the processor will lose cycles waiting for both the instruction request and the data request to finish. In processor designs incorporating queues, it is assumed a data request has been issued some time before it is actually required, allowing an instruction request to interfere without necessarily causing the processor to block.

While both caches and queues are used to reduce the impact of memory latency on processor performance, there is a fundamental difference between these two strategies. Caches attempt to *eliminate* memory latency, while queues allow the processor to *tolerate* it. The proper combination of these two techniques can lead to significant increases in performance by eliminating the majority of the memory latency and allowing the processor to tolerate what remains. The PIPE architecture, described in the next section, achieves a high level of performance by combining the use of data and instruction queues with a relatively small on-chip instruction cache.

3. The PIPE Processor

The PIPE processor is a pipelined single-chip processor designed at the University of Wisconsin and is an outgrowth of the PIPE project. A more detailed description of the PIPE project is available elsewhere (GHLP85).

The PIPE processor features a simplified load/store instruction set, five stages of pipelining, an on-chip instruction cache with queues, both input and output data queues, and an extended version of a delayed branch. A block diagram of the processor is shown in Figure 1. The five pipeline stages consist of Instruction Fetch, Instruction Decode, Instruction Issue, ALU 1/Logical and ALU2. The use of queues throughout the processor is evident in the diagram. The following is a brief description of the architecture and the features most relevant to the instruction fetch studies to be presented later.

3.1. The PIPE Architecture

The PIPE architecture is a register to register type, and has much in common with the Cray and CDC architectures. The PIPE processor is a 32-bit processor with a 32-bit wide internal bus. PIPE uses sixteen 32-bit data registers, divided into a set of 8 foreground and 8 background registers to improve the speed of subroutine calling. PIPE is 16-bit word-addressable, has separate input and output busses, and a single-level interrupt. A barrel shifter is used to perform shifts and a standard ALU performs adds and subtracts as well as logic functions. There are also 8 Branch Registers, whose use will be described later.

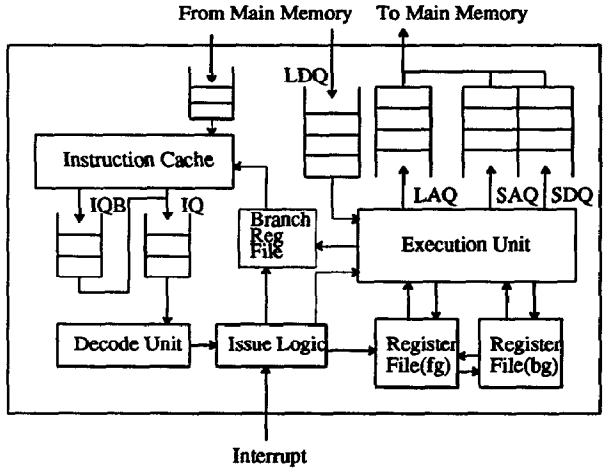
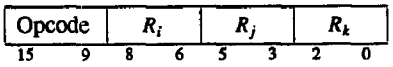


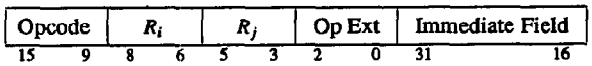
Figure 1. Block Diagram of the PIPE Processor

3.1.1. Instruction Set

PIPE instructions come in 2 forms, single parcel and two parcel, where a parcel is a 16-bit quantity (see Fig. 2). The position of the register fields is the same for all instructions, greatly simplifying the decode logic. The PIPE instruction set supports a basic repertoire of 3 operand instructions; addition, subtraction, logical operations, and shifts in their various forms are all provided. Due to the limited physical space available, PIPE does not support floating point numbers, nor is there any hardware support for multiplication or division.



(a) 16-bit format



(b) 32-bit format

Figure 2. Instruction Format

3.1.2. Architectural Queues

As with the PIPE architecture, the PIPE processor provides both input and output queues which act as insulating buffers between external memory and the internal processing elements of the chip. These queues can be seen in Figure 1. This arrangement allows the on-chip clock rate to be determined solely by the timing delays through the processing elements that comprise the chip. The speed of the external memory has no effect on the processor's internal clock rate.

Since PIPE is a register-to-register type machine, all memory interactions occur through Load and Store instructions. A Load instruction generates a memory address and places it on the tail of the Load Address Queue (LAQ). Items in the LAQ are then sent to the memory system, which sometime later responds with the data item. The data item is not placed directly in a register, but on the *load queue* (LDQ) which acts as a buffer for data. The head of this input queue is visible to the programmer as a register (R7). By making this queue explicit in the architectural definition, a program can have multiple outstanding memory requests without forcing the issue logic to reserve a path into the register file for each request. By employing well known compiler optimization techniques, the load instructions are moved as far ahead of the instruction requiring the data as possible.

The writing of data items to memory occurs in a similar way. A store address is generated and placed on the tail of the Store Address Queue (SAQ). Data items are placed on the tail of the Store Data Queue (SDQ) by specifying register 7 as the destination operand. The items at the top of the SAQ and SDQ are sent as a pair to the memory.

3.1.3. Prepare to Branch

Branch instructions are notorious for causing performance degradation in heavily pipelined machines due to the difficulty in keeping the pipeline full of useful instructions while the branch condition is being evaluated. This problem has been extensively studied [DeLe87,McHe86,Smit81], and a number of methods for minimizing the impact of branches have been developed. The method used in the PIPE architecture is a generalized form of the delayed branch [HJBG82,Radi82].

In the delayed branch scheme, there are a fixed number of *delay slots* following a branch that are filled with instructions that are guaranteed to execute. Ideally the number of delay slots should be as large as possible to guarantee that the branch condition will have been evaluated by the time the instructions complete, thus keeping the pipeline full. Studies of the delayed branch indicate that for many benchmark programs it is difficult to fill more than two delay slots, however. This means that the compiler has to either place null operations into the slots it is unable to fill, or the processor must have the ability to conditionally execute these instructions so that the compiler can place instructions into the delay slots that are *likely* to be executed.

Our experience with many scientific programs is slightly different. We have found that a compiler can easily generate code with an average of 4 instructions that can be unconditionally executed after a branch [YoGo84]. Therefore, PIPE uses an instruction called the prepare-to-branch (PBR) instruction which allows the compiler to specify the number of delay slots (between 0 and 7). Providing the ability to specify how many instructions are to be executed after a PBR instruction allows the PIPE architecture to *always* do as well as the more restrictive delayed branch scheme, while in some cases significantly out-performing it.

In order to support the PBR instruction, 8 Branch Registers were added to the architecture. These registers are not part of the general purpose registers, but a separate set of registers used to store branch target addresses. This allows the PBR instruction to be a single parcel instruction, and allows the compiler to load several branch target address at the beginning of a basic block.

3.2. The PIPE Cache

The PIPE instruction cache is direct mapped and composed of sixteen 4-word lines totaling 64 words, or 128 bytes. This is between 32 and 64 instructions, depending on the distribution of 1 and 2 parcel instructions. In addition to the cache there is an 8-byte Instruction Queue (IQ) and and an 8-byte Instruction Queue Buffer (IQB) that lie between the instruction cache and the decode logic. The only time the cache makes a request for a line from off-chip memory is if the line is guaranteed to contain at least one unconditionally executable instruction. In this paper, when we refer to the PIPE cache, we are actually referring to the physical cache, the IQ, the IQB, and the strategy we use to manage them. The PIPE cache has been explained in detail elsewhere [PIFa86].

This instruction cache, while relatively small, proves sufficient for our purposes. It allows us to verify the design of the control logic, and demonstrate that an I-Fetch strategy such as this need not adversely affect the clock rate. In addition, our simulation results indicate that if the IQ and IQB are used properly, larger instruction caches do not necessarily provide a significant improvement in performance.

3.3. The PIPE Chip

The PIPE chip was fabricated by the MOSIS fabrication facility in 3 micron NMOS with one layer of metallization, and contains just over 37,000 transistors. The chips have been tested and perform at a peak rate of slightly over 6 MIPS. While this number is not as impressive as the performance numbers quoted by several other existing processors, it is important to remember that PIPE was fabricated in a very restrictive technology. SPICE simulations of the PIPE processor using 2 micron NMOS parameters with low resist polysilicon interconnect indicate that if this less restrictive NMOS fabrication process were available to us, the PIPE performance rate would be over 18 MIPS. The availability of a second level of metallization would improve the performance even more. While such fabrication processes are currently being used by such companies as Intel, MOSIS does not support such processes in NMOS. (They do in CMOS, however.) A better comparison for PIPE would be with either RISC-II [Henn84] or MIPS [HJPR83], since both these processors were also manufactured in NMOS. PIPE is 2-3 times faster than either of these machines.

4. Contrasting PIPE and other Instruction Fetch Strategies

When an on-chip instruction cache is all that an SCP design uses to reduce off-chip memory traffic, choosing the correct cache prefetch strategy becomes critical. Hill [Hill87] used trace-driven simulations to compare a wide range of instruction cache configurations and instruction prefetch strategies. Among the many prefetch strategies he modeled were the ones used by the Berkeley SPUR processor, the MIPS-X processor, and a strategy he refers to as *Always*-prefetch. Throughout his study, the always-prefetch strategy consistently provided the best performance. We refer to a cache using this always-prefetch strategy as a *Conventional* cache. The following sections provide a detailed description of the PIPE approach and the approach used in the conventional cache.

4.1. The Conventional Cache

In the model used by Hill, a cache line is composed of a number of sub-blocks, each block with its own individual valid bit. A PC is presented to the cache at the beginning of each clock cycle and a tag lookup and cache array lookup of that PC can both be completed before the end of that cycle. The always-prefetch strategy prefetches an instruction from the next sequential location on each instruction reference, even if this address maps into the next cache line. Memory requests are made for only one instruction at a time, and a new one cannot begin until the previous one finishes. Data fetches have priority over both instruction fetches and prefetches, while instruction fetches have priority over prefetches.

As Hill points out, there are certain implementation problems with the always-prefetch scheme, such as the fact that two reads from the tag array per clock may be necessary. In spite of these problems, this is a very useful model since it consistently provided better performance than any other prefetch strategy studied by Hill.

4.2. The PIPE Instruction Fetch Logic

In the PIPE instruction fetch logic, there are two queues that lie between the instruction cache and the instruction register, the IQ and the IQB (see Fig. 1). Both these queues are the size of a cache line. The IQ, if not empty, is guaranteed to always contain at least one instruction to be executed. No such guarantee is made for the contents of the IQB.

When the IQ becomes empty, an attempt is made to fill it with the data contained in the IQB. If the IQB cannot provide the IQ with valid data, a cache lookup for a new line must be performed. If the line is not in the cache, then a request to memory is initiated.

When the IQB becomes empty, the next sequential line past the one in the IQ is prefetched from the on-chip cache. If that line is not in the cache, the off-chip request is blocked until the control logic can ensure that some portion of the requested line will be executed. The control logic determines whether to make an off-chip memory request based on a number of factors. Due to the encoding of the PIPE instruction set, the existence of a branch instruction is determined by a single bit of the opcode. This allows the cache control logic to easily scan the instructions in the IQ and determine if any are PBR instructions. If there are none, the next sequential line is guaranteed to contain at least one unconditionally executable instruction, and the control logic can initiate the appropriate fetch.

In addition, if there is a PBR instruction in execution, the control logic knows that a certain number of instructions past the PBR instruction will be unconditionally executed. (This number is provided in a 3-bit field of the PBR instruction.) Thus, the control logic can initiate cache lookups or fetches for these instructions to be unconditionally executed while the PBR instruction is being evaluated. Once all the instructions guaranteed to execute pass into the IQ and the result of the PBR instruction has been returned to the cache, the control logic can then start filling the IQB with the instructions stored at the branch target address. If the branch target address is in the cache, there will be no interruption in the supply of instructions and no wasted cycles. If the line is not on chip, having an IQB allows the processor to begin fetching from external memory some number of clocks early. The implication of this is that if the number of delay slots can be made large enough no specific branch prediction strategies are necessary.

5. Simulation Details

For our benchmark programs we chose to use the first 14 Lawrence Livermore loops as defined in [McMa84]. There were two main reasons for this decision. First, we are assuming a scientific workstation environment and the Lawrence Livermore Loops are a well-established benchmark for numeric workloads. Secondly, and perhaps more importantly, we are interested in the interaction between instruction and data requests, and the Livermore Loops do generate a large number of data requests per inner loop, especially when an off-chip floating point unit is assumed. This high data request rate allows us to monitor the ability of the different prefetch schemes to interact with data requests, especially when the cache is small and large numbers of instruction requests are being generated.

The loops were compiled by the original PIPE compiler, and then revised by hand to reflect the slightly different architecture of the PIPE processor. No hand-optimization was done - the loops are not "tuned" to increase performance. The 14 loops were compiled as one large program, so that each loop would run until finished and then fall through to the next loop. This has the effect of flushing the cache every few thousand cycles, since it is guaranteed that at the beginning of each new loop no part of it will be in the cache. The sizes of the inner loops are listed in Table I. A total of 150,575 instructions are executed in a single run through the benchmark program.

Lawrence Livermore Loop sizes in bytes			
Loop #	Inner Loop Size	Loop #	Inner Loop Size
1	116	8	732
2	204	9	272
3	64	10	260
4	80	11	56
5	76	12	56
6	72	13	328
7	288	14	224

Table I. Inner Loops sizes

Memory is modeled as a large external cache that services both instruction and data requests. This cache is connected to the processor chip by a pair of busses, an input bus and an output bus (see Fig. 3). Only the cache deals directly with the large external main memory. The cache is assumed to be large enough to achieve a 100% hit rate in our simulations. The processor does not have an on-chip multiply unit, making an external floating point chip necessary. The floating point unit is addressed as a memory location, so that a pair of data stores to the appropriate locations will cause a

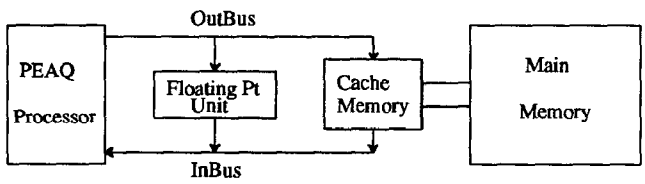


Figure 3. Simulation setup

multiply to occur. The number of clocks necessary to perform a floating point multiply is kept a constant, and is set to 4 clock cycles. Since both the cache and the floating point unit must share the return bus, some bus arbitration is necessary. The simulation model gives precedence to data and instruction loads and stores, followed by multiply results, with instruction prefetches having lowest priority.

Two versions of the PIPE simulator were created, one that used the PIPE cache and one that used the conventional cache. Simulation runs of the benchmark program were performed using both setups, and in the simulation runs performed, the following parameters were varied:

(1) instruction format
(2) instruction cache size
(3) the cache line size
(4) the speed of external memory
(5) the width of the input bus
(6) permitting a pipelined external memory
(7) the instruction queue (IQ) size
(8) the instruction queue buffer (IQB) size

The first parameter compares a fixed 32-bit instruction format to the 16 and 32-bit instruction formats used by PIPE. The second and third parameters are typically associated with cache studies and need no explanation. The next three parameters all deal with variations in effective memory speed. Parameters 4 and 5 specifically reflect technological variations. As a given design is built in a more aggressive technology, the processor may run at a faster speed than the memory. Varying the external memory speed will indicate which instruction fetch strategy can tolerate a relatively slower memory. The input bus width is related to the effective external memory speed since a wider bus will more easily allow prefetch of instructions and thereby make memory appear faster. The 6th parameter, pipelined memory, essentially permits multiple outstanding memory requests. If the memory is pipelined, it is assumed the memory system can accept a new request each clock cycle. The final 2 parameters are specific to the PIPE processor. The simulator was also able to select whether data or instructions have priority at the memory interface (since the PIPE processor uses queues).

Given the large number of parameters listed, it is impractical to include an exhaustive listing of all simulation results. We will therefore concentrate on the significant results and indicate when trends hold for other sets of parameters.

6. Discussion of Simulation Results

The goal of these simulation studies is to compare a conventional cache using the always-prefetch strategy with the PIPE instruction fetch strategy that is based on using an instruction cache, an instruction queue, and an instruction queue buffer. As described earlier, the IQ and IQB permit a type of *lookahead* into the instruction stream and are critical to an effective strategy for fetching off-chip instructions. The early detection of branches and instructions to be unconditionally executed relies on the existence and size of these buffers. In addition, the presence of the IQ makes the the instruction cache available for prefetch activities.

To begin, we compared the conventional cache against versions of the PIPE system with the parameter values listed in Table II. These parameter values were selected as representing a reasonable range of values that could be easily implemented. Notice that the IQ and IQB size track the line size of the cache, except in the case of the 32-byte line size. For this line size we simulated 2 different IQ sizes. For all the simulation results presented here, a fixed 32-bit instruction format was chosen in order to make comparisons to other machines that only have one instruction format more realistic. In addition, instructions requests are given priority over data requests at the memory interface.

Configuration	Line size	IQ size	IQB size
8-8	8 bytes	8 bytes	8 bytes
16-16	16 bytes	16 bytes	16 bytes
16-32	32 bytes	16 bytes	32 bytes
32-32	32 bytes	32 bytes	32 bytes

Table II. Simulated IQ and IQB configurations

Our simulation results indicate that one part of the I-Fetch strategy used by PIPE is non-optimal. As stated earlier, the PIPE processor is an outgrowth of the PIPE project which involved a tightly-coupled pair of processors sharing the same memory. In this environment, it was important to limit memory traffic as much as possible, and so the I-Fetch logic guarantees that some part of every line requested from off-chip will be executed. Our simulation results indicate that, for a single-chip processor on its own, a certain performance penalty is paid by using this strategy and not allowing true prefetch from off-chip. All simulation results presented in this paper assume that true prefetching from off-chip can be done. (The I-Fetch logic does not guarantee that some part of the line requested from off-chip will be executed.)

Our performance metric for these results is the total number of cycles needed to execute the 150,575 instructions. Our choice of this metric is due to the difficulty in computing an effective instruction access time when queues are involved and because the effective access time does not necessarily indicate the impact of the interaction of data and instruction requests upon performance. In the following graphs we plot the total number of cycles used on the vertical axis of our figures and cache size in bytes on the horizontal axis.

Figure 4 shows the performance of our 4 basic configurations and of the conventional cache in the case of a non-pipelined main memory with an access time of 1 clock cycle. (For a memory access time of 1 clock cycle, having a pipelined memory makes no difference.) This would correspond to a machine with a large, fast external cache. Figure 4a is for an input bus width of 4 bytes while in Figure 4b the bus width is 8 bytes. It is clear that the bus width can have a dramatic impact on performance for cache sizes less than 128 bytes. This effect is not unexpected if one considers that the underlying architecture can issue one instruction per cycle and that an instruction is 4 bytes long in these simulations. A bus only 4 bytes wide has difficulty supplying the processor with instructions faster than they are consumed, and cannot get ahead of the issue logic. A bus width of 8 bytes, however, allows the instructions to arrive from memory at twice the rate they are being consumed, thereby permitting the prefetch logic to stockpile instructions.

Looking at the curves in both graphs, an initial large performance improvement followed by a flattening of the curves is evident. The knee of the curve corresponds to the size of most of the inner loops of the benchmark program. In Table I, we see that half of the inner loops fit within a 128 byte cache. The performance improvement seen for an input bus width of 8 bytes is seen across all parameters and becomes more dramatic as the memory access time increases.

There are a couple of other interesting things to note in Figure 4. First, we see in Figure 4b that configurations 8-8 and 16-16 perform uniformly well for varying cache sizes with a bus width of 8 bytes. Thus, using a 16 or 32 byte cache with an IQ and IQB one can achieve close to the performance of a 512 byte cache. Second, as seen in Figure 4a, a memory access time of 1 clock cycle and a bus width of 4 bytes is the only case for which the conventional cache performs better than some PIPE configuration. For a memory access time larger than 1 clock cycle, *all* PIPE configurations *always* perform better than the conventional cache.

In Figure 5 we see the performance results for a bus width of 4 bytes and a bus width of 8 bytes when main memory has an access

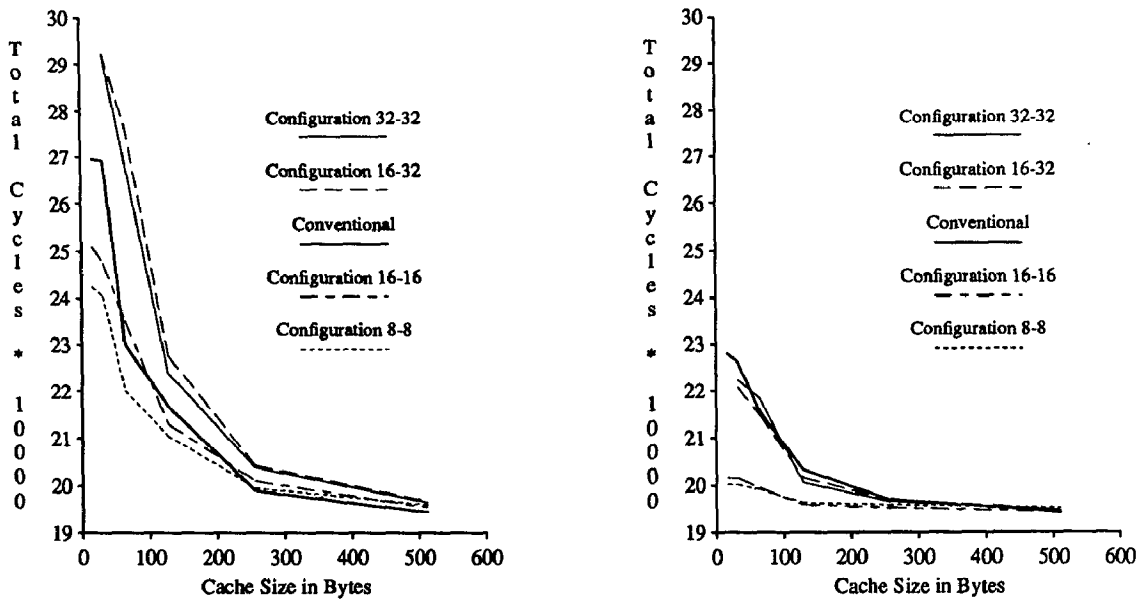


Figure 4. Total execution time with varying cache sizes for a non-pipelined memory with a 1 cycle memory access time.
(a) bandwidth = 4 bytes. (b) bandwidth = 8 bytes.

time of 6 clock cycles and is not pipelined. (Notice that the scale for the total execution time axis has changed from that in Figure 4.) From Figure 5 we see that for small cache sizes and higher memory access times, the PIPE configurations are less sensitive to bus width than the conventional cache. (once the cache size has grown to 256 bytes, the bus width does not make a significant difference.) Simulations with memory access times of 2 and 3 clock cycles showed similar results. Thus, if one is forced to use a bus width of 4 bytes due to technological constraints and memory access time is greater than 1 clock cycle, the PIPE strategy will significantly outperform the conventional cache approach.

Figure 6 shows performance results for a system with a bus width of 8 bytes and a memory access time of 6 clock cycles. Figure 6a (which is the same as Figure 5b with a different scale) represents a non-pipelined memory and 6b represents a pipelined memory. Comparing Figures 6a and 6b, we see that while the shape of the curves is nearly the same, in Figure 6b the curves have shifted down and some compression has taken place. We again see that the

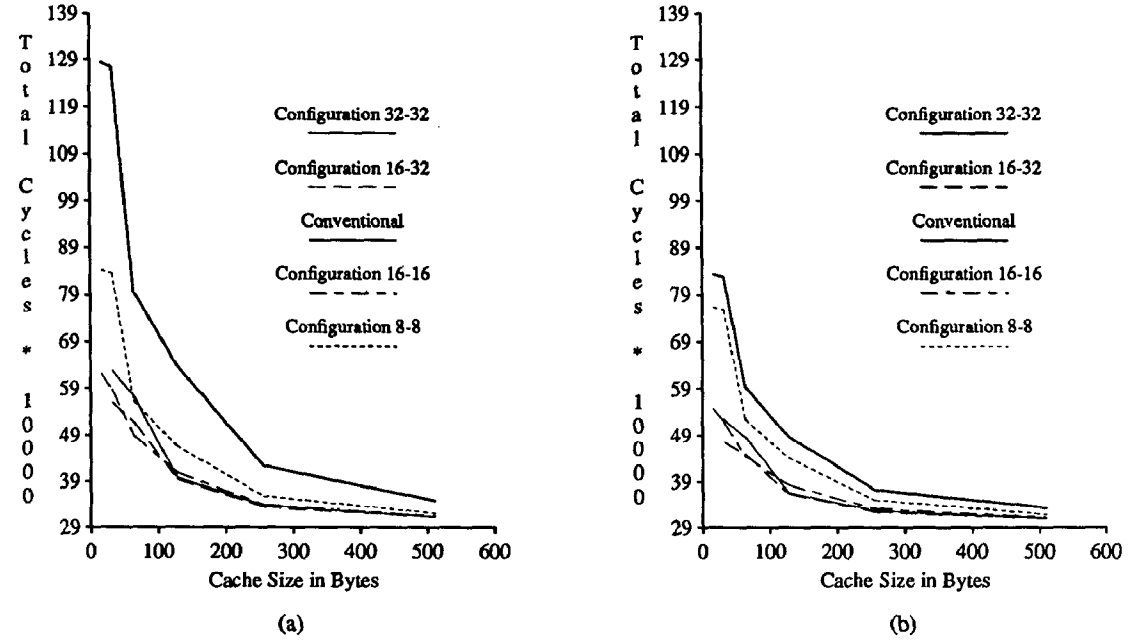


Figure 5. Total execution time with varying cache sizes for a non-pipelined memory with a 6 cycle memory access time.
(a) bandwidth = 4 bytes. (b) bandwidth = 8 bytes.

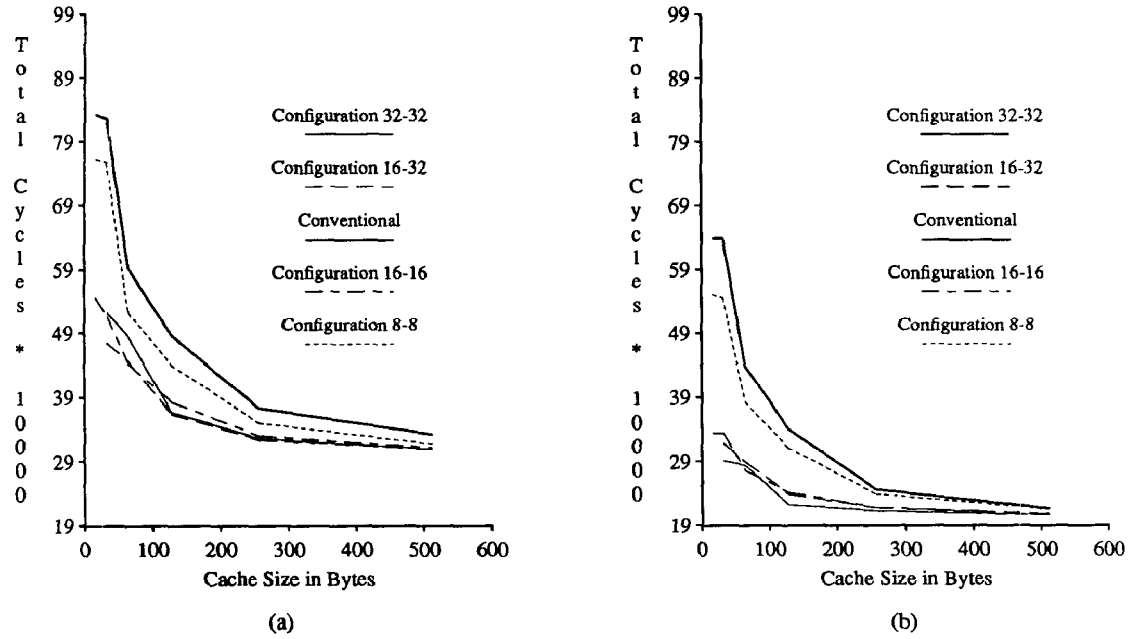


Figure 6. Total execution time with varying cache sizes for a bus width of 8 bytes and a 6 cycle memory access time.
(a) non-pipelined memory. (b) pipelined memory.

PIPE configurations always do better than the conventional cache configuration. In the case of a pipelined memory, some of the PIPE configurations perform substantially better. Figure 6 shows that the PIPE configurations providing the best performance have line sizes of 16 or 32 bytes, which is the reverse of the case shown in Figure 4. In Figure 4, a line size of 8 gave the best performance. Such a result is not completely unexpected and is confirmed by other cache studies [Smit82]. One thing to notice in these figures is the relative flatness of the configurations providing the best performance. In Figure 4b it was configurations 8-8 and 16-16, while in Figures 5b and 6b configurations 16-16, 16-32 and 32-32 were best. Although the performance of the conventional cache and the various PIPE configurations converge as as cache size increases, the PIPE configurations that perform best display a much more uniform performance across all cache sizes. If we compare Figures 4, 5 and 6 we see the PIPE configuration that has a cache line size of 16 (configuration 16-16) performs extremely well for a broad range of cache sizes and memory access times.

From an implementation point of view, in which technology is quickly changing, the results presented here are significant. Using a combination of a relatively small cache, an IQ, and an IQB, excellent performance can be achieved with a limited number of transistors. This is important when designing instruction fetch logic in newer, substantially faster technology that does not permit high density circuitry. In addition, the same design will maintain a high level of performance as main memory speeds improve, or as the technology matures and higher densities become possible. The higher densities achieved in the mature technology can be used to expand the on-chip cache to include data or to provide more on-chip functionality such as multiply hardware.

It is important to remember that the I-Fetch strategy presented here is for a subset of the actual strategy implemented on the PIPE processor. As pointed out earlier, the actual PIPE processor has both a 16-bit and a 32-bit instruction size, which further complicates the I-Fetch logic. Nonetheless, the actual implementation of this approach demonstrates its viability.

7. Summary and Conclusions

In this paper we have presented an evaluation of a set of instruction fetch strategies appropriate for single-chip processors. In most of today's approaches to instruction fetch, the single-chip processor is provided with a simple on-chip instruction cache. On a cache miss, a request is made to the external memory for instructions and these off-chip instruction requests compete with data references for access to the external memory. One way to reduce the number of off-chip instruction requests is to simply increase the size of the on-chip cache. Increasing the instruction cache size is not always possible, however, especially in emerging technologies where high circuit densities are difficult to achieve or where chip area is needed to provide a certain level of functionality. We have suggested an approach to fetching instructions that combines the use of an instruction cache, an instruction queue, and an instruction queue buffer, and briefly described an existing processor that employs this approach. This approach was compared to a simple on-chip instruction cache approach that uses an instruction prefetch strategy that Hill [Hill87] has determined to provide the best performance when used with a conventional cache. Our simulation results indicate that using our approach the processor performs up to twice as fast as a processor using the conventional cache-only approach with a small cache size and can in fact provide performance comparable to larger caches. The viability of this approach is demonstrated by its implementation in the PIPE processor chip.

8. Acknowledgements

This work was supported by National Science Foundation Grants DCR-8604224 and CCR-8706722. We would also like to thank both Mark Hill and Rob Joersz for their valuable assistance.

9. References

[ACHA87] A. Agarwal, P. Chow, M. Horowitz, J. Acken, A. Salz, and J. Hennessy, "On-chip Instruction Caches for High Performance Processors," *Proceedings of the Conference on Advanced Research in VLSI*, Stanford, pp. 1-24, March 1987.

[Adva87] "Advanced Micro Devices," AM29000 User's Manual (1987).

[BCDF87] A. D. Berenbaum, B. W. Colbry, D. R. Ditzel, R. D. Freeman, H. R. McLellan, and K. J. O'Conner, "CRISP: A Pipelined 32-bit Microprocessor with 13k-bit of Cache Memory," *IEEE Journal of Solid State Circuits*, vol. SC-22, pp. 776-782, October 1987.

[GHLP85] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: a VLSI Decoupled Architecture," *Proceedings of the Twelfth Annual Symposium on Computer Architecture*, pp. 20-27, June 1985.

[GrPa82] G. F. Grohoski and J. H. Patel, "A Performance Model for Instruction Prefetch in Pipelined Instruction" Units," *Proceedings of the Ninth International Symposium on Parallel Processing*, pp. 248-252, August 1982.

[HCSS87] M. Horowitz, P. Chow, D. Stark, R.T. Simoni, A. Salz, S. Przybylski, J. Hennessy, G. Gulak, A. Agarwal, and J.M. Acken, "MIPS-X: A 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache," *IEEE Journal of Solid-State Circuits*, vol. SC-22, pp.790-799, Oct. 1987

[Henn84] J. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers*, vol. C-33, No. 12, pp.1221-1246, Dec. 1984

[Hill87] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Doctoral Thesis, Department of Computer Sciences, University of California, Berkeley, California.

[HJBG82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross and J. Gill, "Hardware/Software Tradeoffs for Increased Performance," *Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 33-54, March 1983.

[HJPR83] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, and T. Gross, "Design of a High Performance VLSI Processor," *Proceedings of the Third Caltech Conference on VLSI*, pp. 2-11, March 1982.

[KMOM87] H. Kadota, J. Miyake, I. Okabayashi, T. Maeda, T. Okamoto, M. Nakajima, and K. Kagawa, "A 32-bit CMOS Microprocessor with On-Chip Cache and TLB," *IEEE Journal of Solid-State Circuits*, vol. SC-22, pp.800-807, Oct. 1987

[McMa84] F. H. McMahon, "LLNL FORTRANS KERNELS: MFLOPS," Lawrence Livermore Laboratories, Livermore, CA, March 1984.

[PaSe80] D. A. Patterson and C. H. Sequin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Transactions on Computers*, Vol. C-29, No. 2, February 1980.

[PlFa86] A. R. Pleszkun and M. K. Farrens, "An Instruction Cache Design for use with a Delayed Branch," *Advanced Research in VLSI: Proceedings of the Fourth MIT Conference*, April 1986.

[Radi82] G. Radin, "The 801 Minicomputer," *Symposium on Architectural Support for Programming Languages and Operating* Systems," pp. 39-47, March 1982.

[RaRo77] B. R. Rau and G. E. Rossman, "The Effect of Instruction Fetch Strategies upon the Performance of" Pipelined Instruction Units," *Proceedings of the Fourth Annual Symposium on Computer Architecture*, pp. 80-89, June 1977.

[Smit81] James E. Smith, "A Study of Branch Prediction Strategies", *Proceedings of the Eighth Annual Symposium on Computer Architecture*, pp. 135-148, May 1981.

[SmGo85] J. E. Smith and J. R. Goodman, "Instruction Cache Replacement Policies and Organizations," *IEEE Transactions on Computers*, Vol. C-34, No. 3, pp. 234-241, March 1985.

[Smit82] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September 1982.

[YoGo84] H. C. Young and J. R. Goodman, "A Simulation Study of Architectural Data Queues and Prepare-to-branch" Instruction," *Proceedings of the IEEE International Conference on Computer Design*, pp. 544-549, October 1984.