A DECOUPLED INSTRUCTION PREFETCH MECHANISM FOR HIGH THROUGHPUT

BY

SNEHAL RAJENDRAKUMAR SANGHAVI

B.E., University of Mumbai, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

ABSTRACT

Superscalar processors today have aggressive and highly parallel back ends, which place an increasing demand on the front end. The instruction fetch mechanisms have to provide a high bandwidth of instructions and often end up as the bottleneck. The challenge is to mitigate the problems arising due to the vase difference in processor and memory speeds.

This thesis proposes a solution to the problem. It presents a prefetching architecture that decouples the instruction fetch stage from the rest of the processor pipeline by an *instruction queue* (iQ). This allows the fetch stage to continue making future fetch requests while waiting for a miss to be serviced. The *early* fetch requests, or prefetching, warms up the cache with instructions. Thus, this system does useful work during the cache miss latency period. This thesis demonstrates that the mechanism reduces the number of stall cycles in the fetch stage, and enables the caches to be made smaller without a decrease in performance.

I dedicate this thesis to my parents Mainakee and Rajendra Sanghavi and my brother Sujay for their love, care, selflessness, and companionship.

ACKNOWLEDGMENTS

I would like to express my gratitude to my adviser, Professor Matthew Frank, for his advice, guidance, and patience, and for clearing up many concepts. I would also like to thank my colleague Kevin Woley for answering my numerous questions and helping me with simulation-related issues.

I also take this opportunity to thank my friend Amit for his support and interest.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Modern superscalar processors have a large instruction window and issue multiple instructions simultaneously. This level of parallelism places pressure on the front-end stages of the pipeline. To fully exploit the advantages of the parallel back end, instructions have to be supplied at a high bandwidth. It is a challenge because of the vast difference between processor and memory speeds, instruction cache (I-cache) misses, and the nonsequential execution behavior of programs.

One solution to alleviate this problem is p*refetching* [1] - [3]. During prefetching, a cache miss is anticipated in advance, and a fetch is initiated well before the address is actually referenced. Prefetching, as described in [1], is a technique in which instruction fetch requests are generated before they are actually needed. Thus, instructions are brought into the cache in advance, rather than on demand. This method hides or lessens the visibility of cache miss latency. By placing a fetch request in advance, a cache miss that would have occurred in the future, occurs now. So when the instruction is *really* needed, the line has already been brought into the cache, causing a real hit; or the line is in transit from a lower-level cache, resulting in a visible latency that is less than the maximum cache miss latency.

The proposed design aims to implement an instruction prefetching scheme that fits snugly into the existing processor design, without modifying the existing branch predictors and other stages of the pipeline. An *instruction queue* (iQ) is inserted between the instruction fetch (ifetch) and decode stages. This serves as a buffer for future instructions before they are consumed by the decoder and later stages. It also allows the branch predictor to continue making predictions even on an I-cache miss. No changes need to be made to any of the other stages of the processor pipeline.

The intuition behind this design is that it would be useful in reducing the size of the L1 cache. Normally, large caches are used to minimize the miss rate. Since a large cache can hold many lines at once, capacity and conflict misses are minimized. However, a large cache has a higher access time and consumes more power. There are advantages to having a smaller cache. The access time is shorter. If the size of the cache is smaller than or equal to the page size, then address translation can occur in parallel with cache access. Smaller caches also save silicon and consume less power.

The proposed prefetching mechanism facilitates the use of smaller caches for achieving the same instructions per cycle (IPC). When the prefetching mechanism is in place, it serves to warm up the cache before the instruction addresses are actually referenced. Thus, future capacity and conflict misses occur in advance, and lines that might be referenced in the future are brought into the cache. Because of this mechanism, it is possible to tolerate higher miss rates, essentially meaning that the cache can be smaller in size for the same level of performance.

Chapter 2 briefly describes previous research and implementations related to prefetching and other solutions. Chapter 3 describes the proposed design in further detail. The experimental setup used for the purpose of simulations is outlined in Chapter 4. The performance of this scheme is presented in Chapter 5. The main performance metric is the IPC. Chapter 6 concludes the work.

CHAPTER 2

RELATED WORK

Modern superscalar processors with aggressive execution cores require a high rate of instruction delivery from the front end. Much research has been done to address this issue. This chapter gives an insight into relevant research specific to prefetching as well as a brief overview of some other solutions that have been proposed.

2.1 Prefetching

A lot of work has been put into the design of prefetching mechanisms for both data and instructions. Decisions have to be made regarding what to prefetch and when to initiate it, how many bytes or lines to prefetch, where they should be stored and how the cache would support this system. Prefetching, as described in [3] followed the one block lookahead algorithm. In this algorithm, only the line immediately sequential to the current referenced line can be prefetched. It was believed that this is necessary to facilitate fast hardware implementation. This scheme does not take into account the non-sequential execution behavior of many programs.

Guided prefetching for data was introduced in [1]. The architecture uses a look-ahead program counter (LA-PC) which is incremented and maintained in the same fashion as the regular PC. It is guided by a dynamic branch prediction mechanism and runs ahead of the normal instruction fetch engine. The LA-PC is used along with a reference prediction table to generate data prefetching requests. Keeping LA-PC well ahead of the normal PC allows data to be brought in advance, thus hiding memory latency in case of a miss. This scheme was applied for prefetching data only.

An instruction prefetching technique discussed in [2] has a fetch target queue (FTQ), which stores address predictions made by the predictor. These are consumed by other optimizers. A selection mechanism called *cache probe filtering* selects candidate addresses to be prefetched and stores them in a prefetch instruction queue (PIQ). These addresses are then used to prefetch from L2, and the instructions are placed in a prefetch buffer. The instruction fetching mechanism has to decide whether to obtain the next instruction sequence from the I-cache or the prefetch buffer. This design implements many structures other than the prefetch queue: a selection mechanism, a PIQ, a prefetch buffer, and a fetch target queue. The mechanism would not be able to fit into the current processor without significant changes. It is also more complex to implement compared to the design proposed in this thesis.

2.2 Other Research

Several other solutions have been proposed to increase the fetch bandwidth. Some solutions aim at increasing the number of instructions fetched per cycle by using additions like the collapsing buffer [4], or by using multiple narrow fetch units in parallel to achieve a large combined fetch bandwidth [5], and then combining it with parallel renaming to further increase the number of available instructions [6].

Yet another set of solutions aim at reducing the impact of cache misses by doing useful work during the latency period. In [7], the impact of an I-cache miss is reduced by writing instructions into reservation stations out of order. On a miss, the branch predictor continues to make predictions, the processor fetches the instructions and writes them in the reservation stations.

CHAPTER 3

DESIGN DETAILS

3.1 Basic Concept

It is useful to know how the prefetching scheme is beneficial. Figures 1 and 2 show a timeline of operation for both the original and prefetching processors. I_i are the addresses of the instructions. Assume that I_1, I_2, and I_3 are sequential instructions, but I_3 lies on a different line. Further, I_3 is a branch instruction pointing to I_4 which lies in an altogether different line. I_5 sequentially follows I_4.

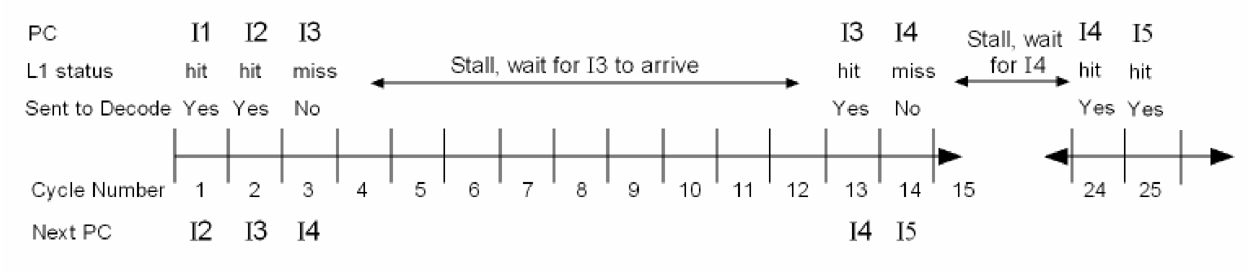

Figure 1 Timeline for the original processor

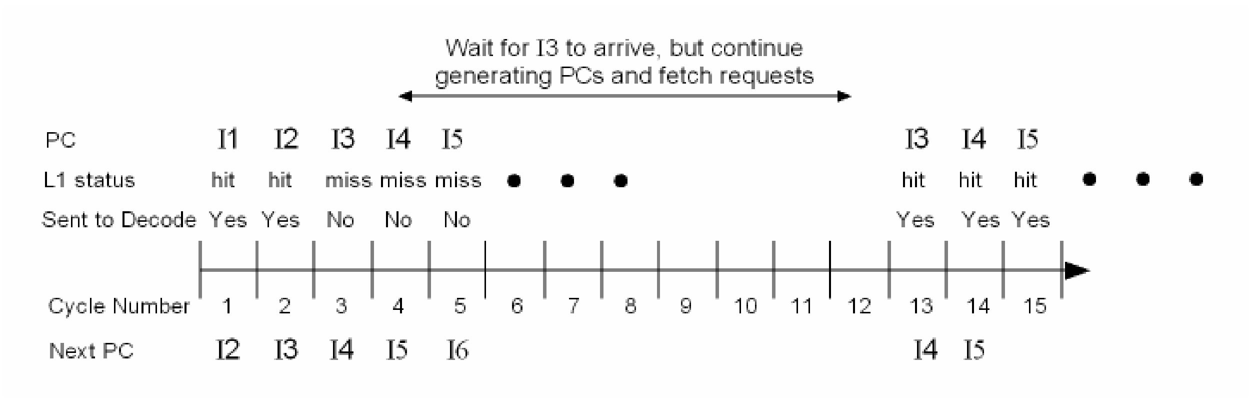

Figure 2 Timeline for the prefetching processor

In the original processor, after fetching I_1 and I_2, there is a miss on I_3 since it is on a different line. The ifetch stage has to stall for 10 cycles while the miss is serviced. Normal operation continues on cycle 13, and on cycle 14 there is a miss for I_4. The processor again stalls for 10 cycles while the miss is serviced. I_5 is available only in cycle 25.

In the prefetching processor the ifetch does not stall on the I_3 miss, but continues to generate next PCs and placing requests to the I-cache. This is effectively like pipelining fetch requests such that there are multiple requests active in parallel. Thus, I_5 is fetched earlier and is available in cycle 15 itself. As we can see, prefetching populates the I-cache with useful instructions in advance, and reduces the number of stalls in the fetch stage.

3.2 Processor Structure

The processor in which this design is incorporated is modeled on the MIPS R10000 processor. The framework of the processor pipeline is shown in Figure 3. The first stage of the pipeline is the address generation and instruction fetching unit. It is elaborated in Figure 4. The address generation is done by a branch predictor with the help of a branch target buffer (BTB). The branch predictor is capable of making one prediction per cycle. The ifetch unit checks for a hit in the L1 I-cache. If there is a hit, the instruction is returned in one cycle, is latched and sent to the decoder. On a miss, the L2 unified cache is accessed, and the ifetch stage stalls till the line is brought into the L1 I-cache. While the ifetch stage is stalled, the branch predictor does not make any new predictions and a NOP is latched and sent to the decoder. What this means is that instructions are always sent to the decoder in the correct predicted order. After the missed line arrives, the instruction is sent to the decoder, and the branch predictor resumes operation. A return address stack (RAS) is also maintained in this stage.

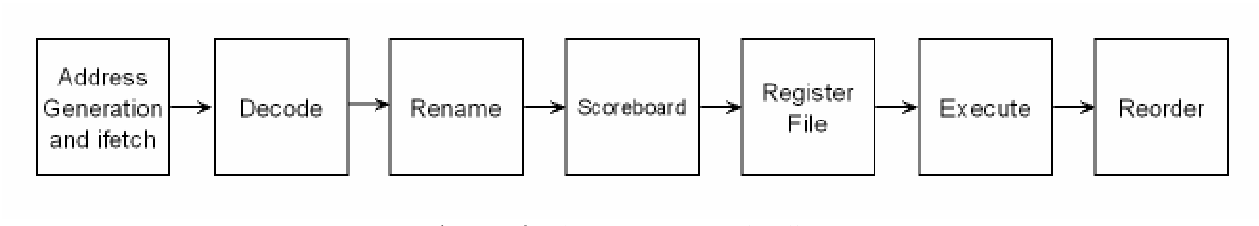

Figure 3 The processor pipeline

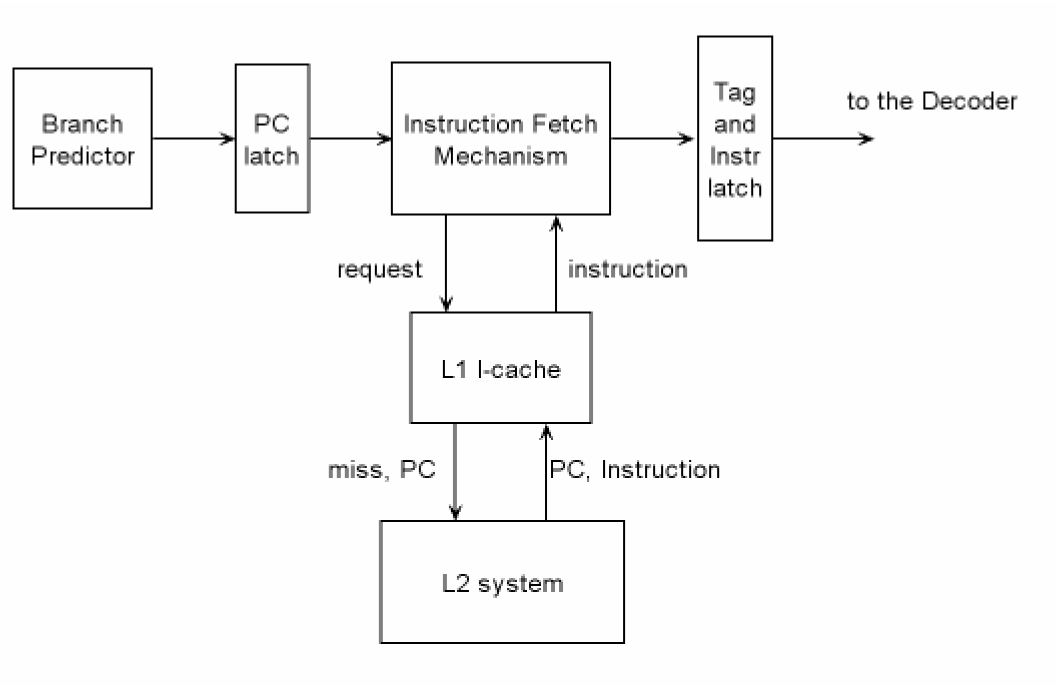

Figure 4 Detailed instruction fetch unit

3.3 Modifications to the Structure

To be able to warm up the cache with instructions in advance, as opposed to on demand, future instruction addresses have to be generated. The branch predictor is used to generate the future addresses, which will be used for prefetching. There are many algorithms describing what and when to prefetch. In this design, the decision on *what* to prefetch is made by the branch predictor. The reason for this is that it more closely resembles the actual execution sequence that the program undertakes. As for *when* to initiate prefetch, it is done every time there is a miss in the L1 I-cache. That is when useful work is done during the cache miss latency.

Even with prefetching in place, it is essential to send instructions to the decoder in the original order, so that no changes are required in any of the other stages of the pipeline. Thus, the instruction queue has a FIFO structure.

Two modifications are made to the ifetch stage. In the original implementation, the ifetch stage stalls completely on an I-cache miss. It waits for the line to be brought in from the L2 cache or memory system. A NOP is sent to the decoder, and the branch predictor makes no predictions. Normal operation resumes after the line is brought in. In the proposed design, this stalling condition is lifted so that branch prediction continues normally, even on a miss. The second modification is the introduction of an iQ between the ifetch and decode stages. Figure 5 shows this organization.

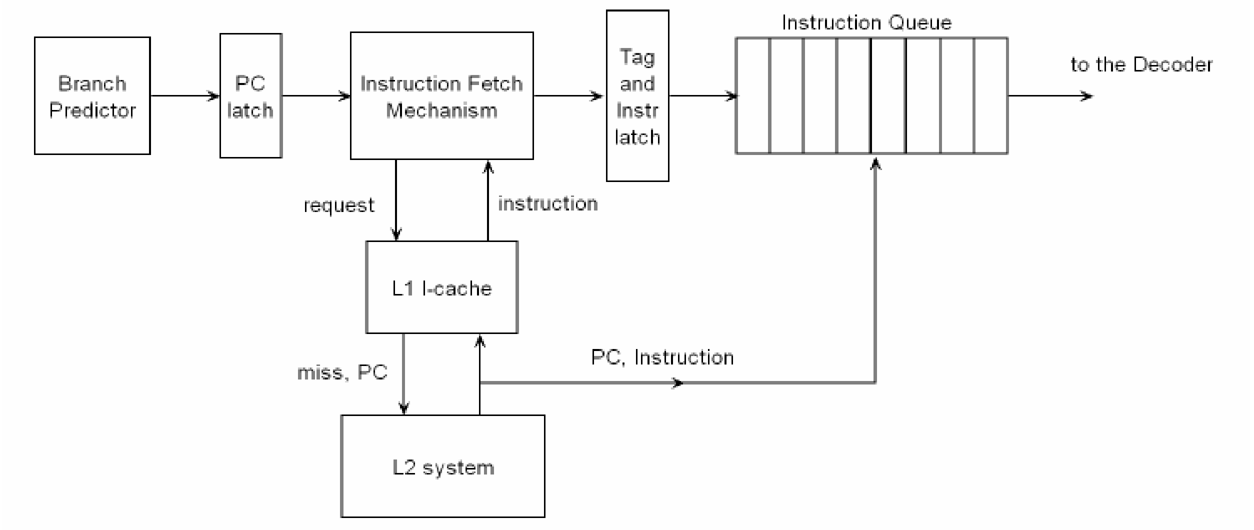

Figure 5 Modified ifetch unit: Introduction of the instruction queue

3.4 The Branch Predictor

The branch predictor is a part of the instruction fetch stage of the processor pipeline. It is capable of making one prediction per cycle. Each prediction is essentially the next PC that has to be fetched. This new PC could be sequentially after the previous PC or could be in another line if the previous PC was a control instruction.

The branch predictor design is not changed. However, since the condition in which the fetch stage stalls on a miss is lifted, the branch predictor continues to make one prediction per cycle even on an I-cache miss. The predicted addresses are sent to the fetching mechanism immediately, and an entry is also made in the iQ.

3.5 The Instruction Queue and L1 I-cache

The iQ is a first in first out (FIFO) structure placed between the ifetch and decode stages as shown in Figure 3. The iQ acts like a buffer and decouples the instruction fetching mechanism from the rest of the processor pipeline, thus permitting instructions to be fetched even on a miss. The queue stores both the predicted PCs and the corresponding fetched or prefetched instructions. On each cycle, the branch predictor predicts the next PC which is used to access the I-cache. An entry for the predicted PC is made in the next iQ slot. If there was a hit in the I-cache, the *fetched* instruction is placed in the iQ corresponding to its PC.

If there was a miss in the I-cache, an entry is made in the iQ, but contains only the PC and not its corresponding instruction. While the L2 cache and memory system service the miss, the branch predictor continues making predictions into the future. These future PCs are inserted into the iQ, and prefetch requests are sent to the fetching mechanism. Some of these prefetch requests could result in a hit, and the *prefetched* instructions are placed in the iQ corresponding to their PCs. When the missed line becomes available, the L2 system sends the requested instruction to the iQ and an entry is made corresponding to its PC. The line is also placed in the L1 I-cache. This means the L1 I-cache is being populated by lines which might contain instructions referenced in the future.

On each cycle, the entry at the head of the iQ is checked for availability. If the entry has a complete PC-instruction pair, then it is consumed by the decode stage. If it is incomplete and the instruction has not yet arrived (waiting for a miss to be serviced), a NOP is sent to the decoder. This functionality is the same as in the original simulator i.e. instructions are sent to the decoder in the same order as before.

One essential requirement for this scheme is that the I-cache should be nonblocking. This means that even on a miss, it should continue servicing subsequent read requests. Each read request might miss in the cache, and each of these misses should be sent to the L2 system. The I-cache also needs to have one read port and one write port.

The iQ acts like a buffer and gives a glimpse of a stream of future instructions and addresses which includes branches. As mentioned in [2], this can be used to guide various PC-based predictors like data-cache prefetchers, value predictors and instruction reuse tables.

3.6 Recovery Mechanism

If there is any type of misprediction or misspeculation, it means that the instructions and addresses currently in the iQ are not on the right path. The PC is restored, the iQ is flushed, the history and RAS are restored, and the branch predictor starts making fresh predictions along the correct path. These new predictions make entries in the iQ in the normal fashion.

CHAPTER 4

SIMULATION SETUP

The simulator used is based on the MIPS R10000 processor. Parameters for the base case processor are outlined in Table 1.

Table 1 Baseline Simulation Parameters

L1 Cache (Instruction cache)	8KB, 2-way set associative, 32 lines, 128-byte line, hit-time 1 cycle, miss latency 10 cycles
L2 Cache (Unified cache)	128KB, 8-way set associative, 512 lines, 128-byte line, miss latency 100 cycles
Execution Width	Superscalar factor of 4
Scoreboard	128 entry FIFO
Branch Target Buffer	4096 entries, 4-way set associative, 4 bytes per line
Branch Predictor	Gshare predictor, 8192 entries, 8 bits of global history

A sample set of benchmarks from the SPEC INT 2000 suite are compiled for MIPS and used for simulation. Each instruction is 4 bytes long. The benchmarks are simulated till a maximum of 300 million instructions. Three different cases are studied.

First, for emphasizing the usefulness of the instruction queue, the L1 instruction cache parameters are modified. Both the number of lines and size of each line are progressively reduced to see the effect of smaller caches on performance. Another indirect way of looking at the effects is to see how often the ifetch stage stalls for the original and prefetching architecture. Identical configurations are used for both the prefetching scheme and the original processor and their results are compared.

Second, the size of the instruction queue is varied to study its impact on performance. The sizes considered are 512, 256, 128, 64, 32, and 16. One base simulation is also run where the iQ is unbounded in size. This is useful while comparing the effect of bounded and unbounded FIFO sizes.

Third, the L2 unified cache is made almost negligible at 32 bytes and can hold only 8 instructions. The size of the I-cache is kept constant at 8 kB as in the baseline processor. The purpose of reducing the L2 to such a small size is to see whether and how prefetching can maintain good performance with just the L1 I-cache.

CHAPTER 5

RESULTS AND ANALYSIS

This chapter provides a comparative analysis of the performance achieved by the iQ-based prefetching mechanism under different architecture configurations. The effect of decreased cache size is discussed first. The impact of a restricted iQ size is examined next. Finally, the performance of the prefetching scheme under the extreme case of a negligible L2 cache is evaluated. All comparisons are made with the original architecture ─ with same configurations and without the iQ. The metric used to measure performance is instructions per cycle (IPC).

5.1 Reduced L1 I-cache Size

As discussed earlier, the instruction queue based prefetching mechanism allows the L1 I-cache to be made smaller with only a small drop in performance. The main reason for this is that prefetching warms up the cache with to-be-addressed instructions, thereby preventing an I-cache miss or lessening its effect. Figure 6 shows the effects of decreasing cache size for both the original processor and the prefetching processor for six different benchmarks. The x-axis shows the progressively decreasing L1 I-cache size. Both the number of lines and the size of the line are reduced. It is a two-way set associative cache. When the cache size is the same as the original configuration of 8 kB, the IPCs for both the original and prefetching processors are almost similar.

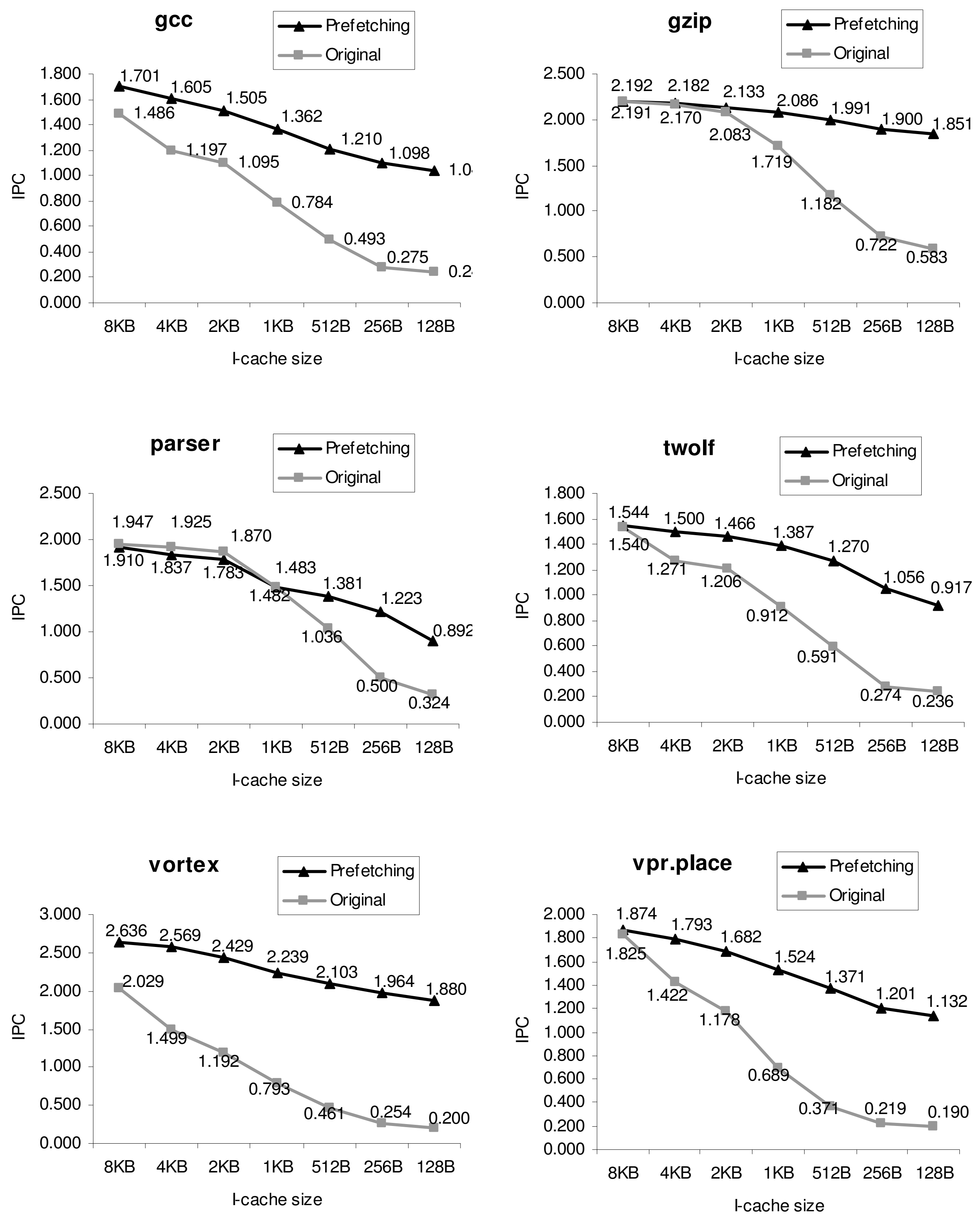

Figure 6 Effect of prefetching on performance at reduced cache sizes

Two observations are made from these figures. The first is that with a decrease in L1 I-cache size the IPC for the original processor drop much quicker than the prefetching processor. For example, for the *gzip* benchmark, when the cache is reduced from 8 kB to 512 B, i.e., 1/16th of its original size, the IPC drops by 10% in the prefetching processor. In the original processor, it drops by 22%. If the size of the cache is further reduced to 256 B, the drop is only 14% for prefetching, and a drastic 67% for the original one.

The second observation is that for the same performance level, the prefetching processor can use a smaller cache. For example, for *gcc,* a 2 kB cache in the prefetching processor achieves the same IPC as the 8 kB cache in the original processor. As the cache size reduces, the performance of the original processor falls much quicker than the prefetching processor. This observation is in agreement with the original performance intuition that prefetching allows usage of smaller cache sizes without a significant decrease in performance. This effect is more apparent in Figure 7 where the performance of the original and prefetching processors is shown for the original cache size of 8 kB and for a reduced cache size of 512 B. It is clearly seen that the original processor's performance drops drastically for almost all benchmarks.

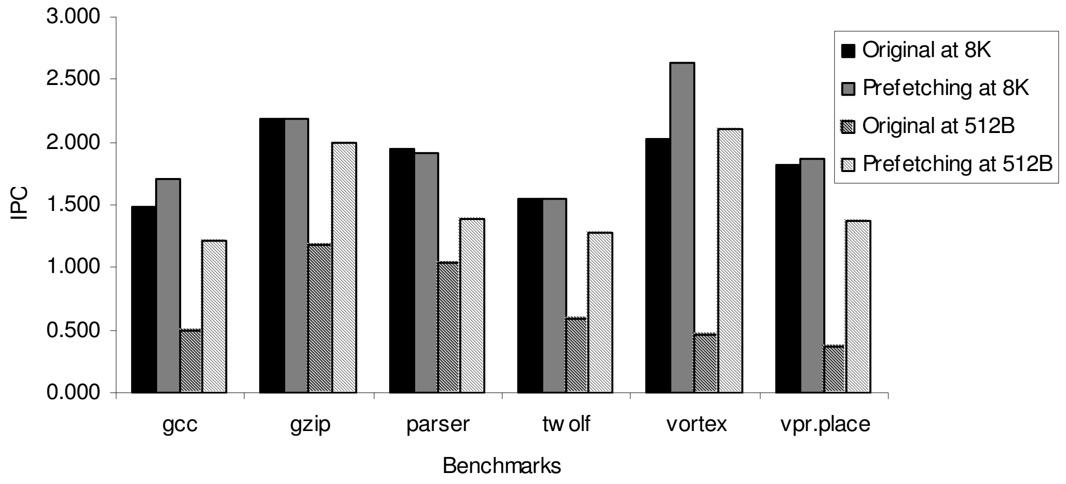

Figure 7 Performance at 8 kB and 512 B I-cache

The ifetch stage stalls every time the next instruction to be sent to the decoder is not available, and sends a NOP. This is true for both the original and prefetching processors. However, prefetching has the effect of reducing the number of times the ifetch stage stalls. With prefetching, future lines have been brought into the cache in advance, and so the referenced line either hits in the cache, or sees a smaller latency (when the line is in transit from L2). Thus, the instruction fetching mechanism has to wait for a smaller number of cycles for the line to arrive. As a result, the ifetch stage stalls less frequently. A representation of this is shown in Figure 8. It gives a comparison between the number of stall cycles for the original processor and the prefetching processor, as the cache size is decreased. As is expected, the number of stalls would increase as the cache is made smaller, because of the higher rate of misses. But it can easily be observed that the number of stalls rises much more rapidly in the original processor since there is no prefetching.

5.2 Restricted Instruction Queue Size

The prefetching processor was simulated with an unbounded FIFO, as well as with various fixed sizes. Figure 9 shows the how the performance is affected by the different FIFO sizes. As can be seen, smaller iQ sizes have almost no impact on performance. The IPC remains constant for all FIFO sizes and takes a slight dip at size 16. This shows that restricting the FIFO size will not adversely affect advantages obtained by prefetching. Thus, a feasible size of the FIFO is possible, which doesn't take up too much area and is simpler to implement.

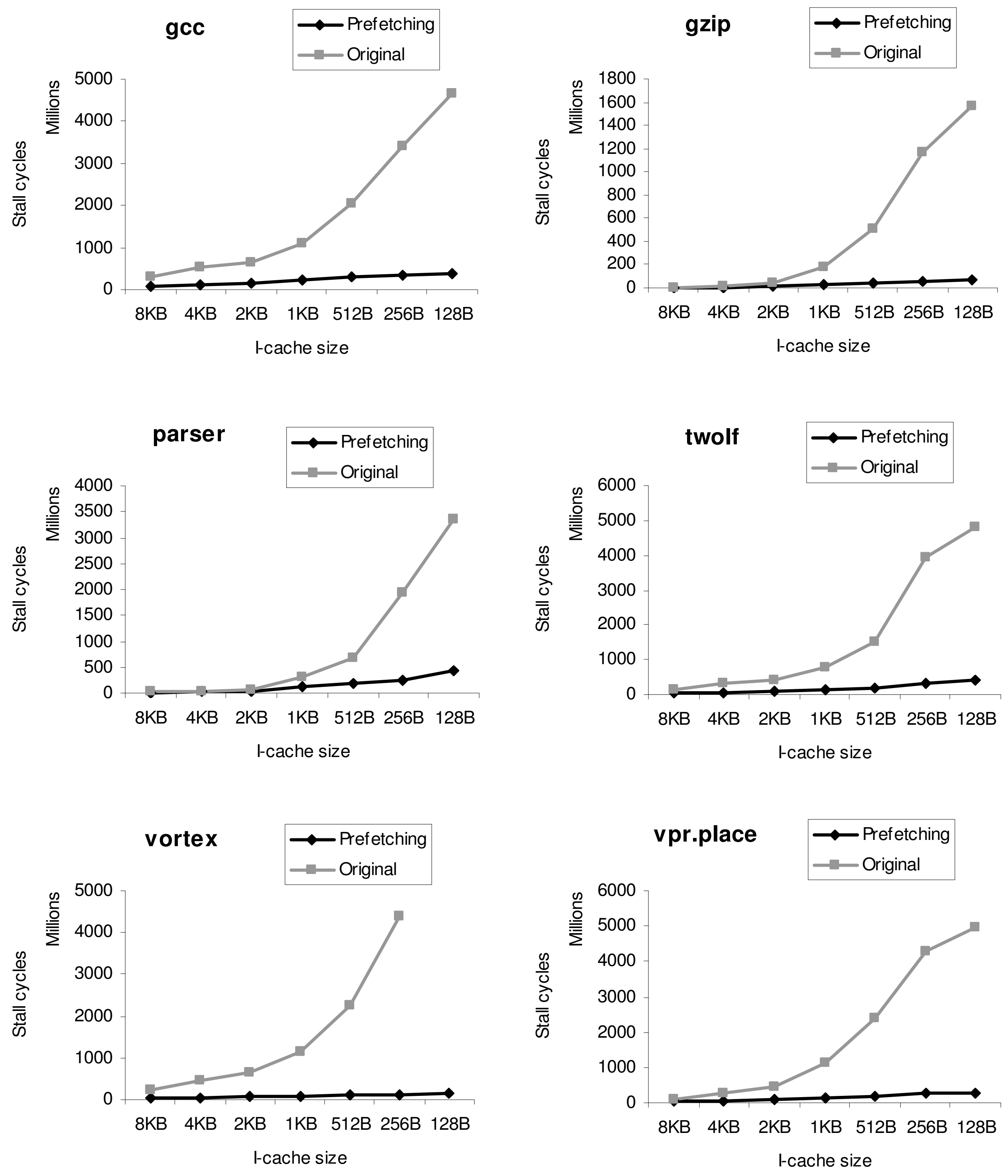

Figure 8 Effect of prefetching on ifetch stall frequency

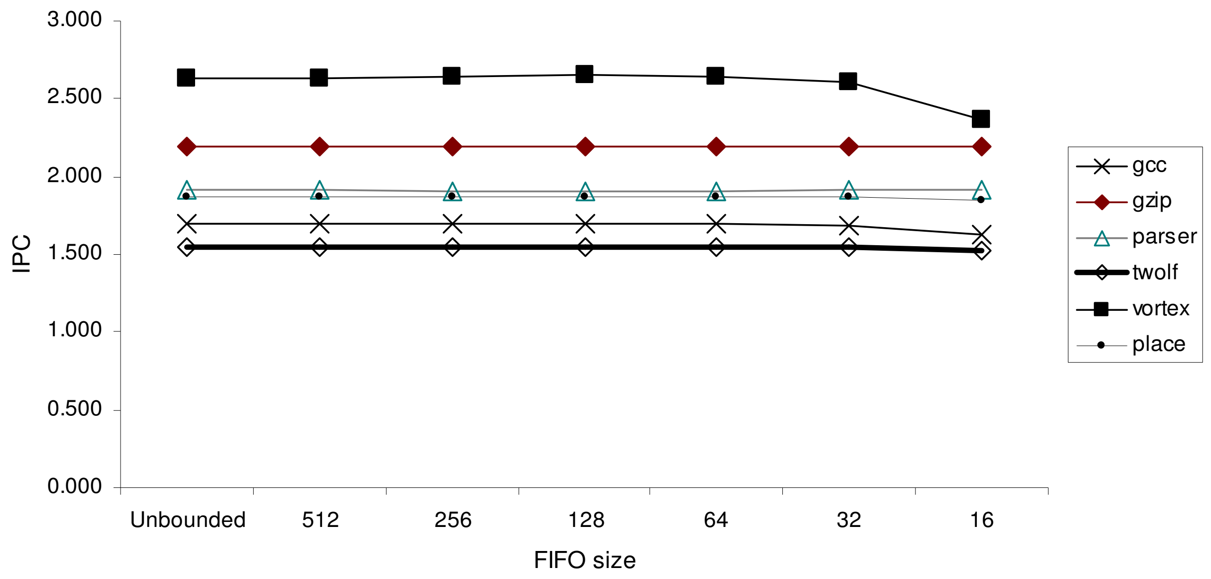

Figure 9 Impact of varying iQ size

5.3 Negligible L2 Cache

To demonstrate the effectiveness of the prefetching mechanism, the L2 cache is reduced to a negligible size of 32 B, holding only 8 instructions. Normally, the L2 is so large (128 kB in the baseline configuration) that almost no line misses in it and L1 miss latency is 10 cycles. By completely removing the L2, every miss in L1 would suffer a latency of 100 cycles since the request has to go to the memory system. The L1 has its baseline configuration of 8 kB.

Since this is an extreme case where there is hardly any L2 cache to support misses in L1, there is bound to be performance degradation. However, the prefetching processor outperforms the original processor and the loss in performance is not as much. Figure 10 shows the performance of the original and prefetching processors at both the baseline configuration and also at the small L2 configuration, for six benchmarks. Figure 11 shows how the prefetching mechanism works better than the original by expressing the percentage of improvement.

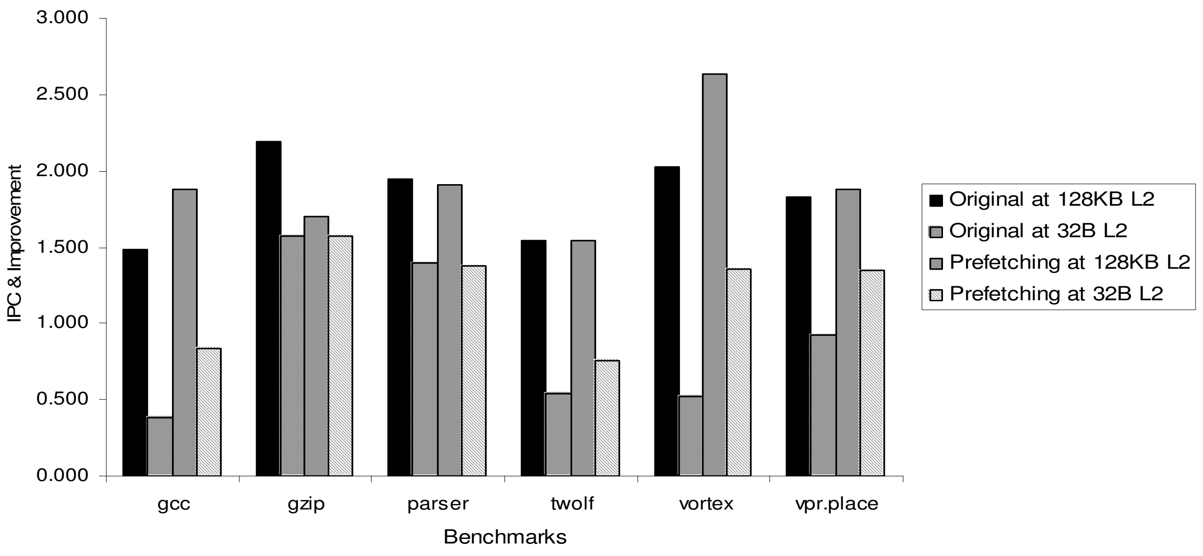

Figure 10 Performance at baseline configuration and at a negligible L2 cache

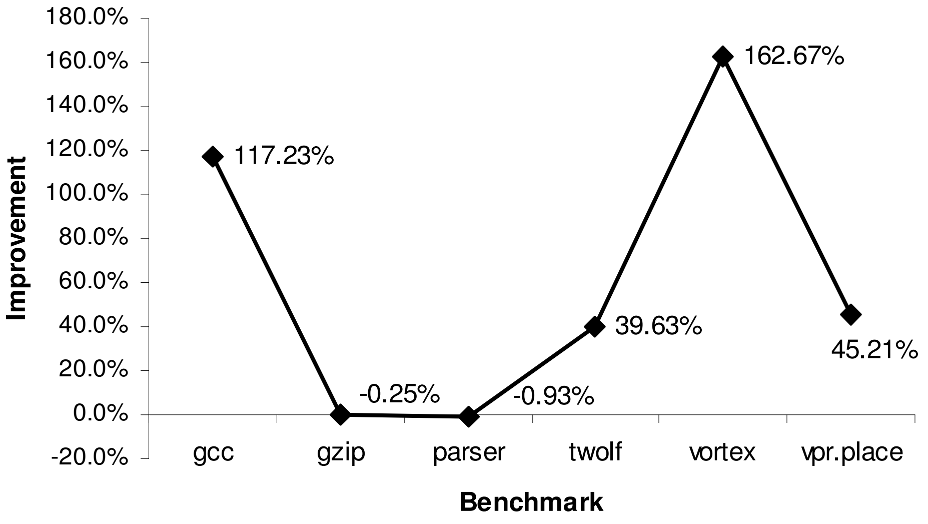

Figure 11 Performance improvement at a negligible L2 cache

It must be kept in mind that the L2 is a unified cache containing both instructions and data. Making the L2 negligible in size affects both the instruction fetching mechanism as well as the data fetching mechanism, and thus the efficiency of the back end. The reason there exists any degradation at all, is because of the increased latency for *both* instruction and data misses in the L1 caches.

We can see in Figure 11 that there is no improvement in the case of *gzip* and *parser*. The reason for this is that both these benchmarks use a small working set of instructions. Since the L1 I-cache cache is reasonably sized at 8 kB, it can hold most of these instructions, thereby reducing the number of misses. Prefetching shows its usefulness in the case of *gcc* and *vortex* benchmarks which have a large instruction working set. These benchmarks miss in the L1 I-cache more often, and prefetching is beneficial in reducing the effect of the large 100 cycle miss latency. Therefore, the performance improvement is much larger.

CHAPTER 6

CONCLUSION

Highly parallel back-ends of modern processors place huge demands on the instruction fetching mechanisms. This is because of the differences in speed between the processor and memory, the existence of branches and to a certain degree, because of misses in the caches.

This thesis proposed a decoupled instruction prefetching mechanism as a solution to mitigate the problems faced by the front-ends of processors. The decoupling of the ifetch stage from the rest of the pipeline with the help of an instruction queue allows fetching to continue beyond misses. This helps to populate the L1 I-cache with useful instructions, thereby reducing the number of misses and stall cycles in the ifetch. The results show that the number of stalls is significantly less in the prefetching processor as compared to the original processor. Furthermore, since the direction of prefetching is determined by the branch predictor, the I-cache contains instructions which are more likely to be on the execution path of the program.

Since the caches are populated with useful instructions, this mechanism allows the caches to be made smaller in size for the same level of performance. This is a huge advantage since smaller caches means smaller access times and less area. It proves that the prefetching mechanism is resilient to I-cache misses. The results also show that this is achievable with a fairly small size of the instruction queue, which makes it practical for implementation.

REFERECES

[1] T. Chen and J. Baer, “Reducing memory latency via non-blocking and prefetching caches,” in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, R. L. Wexelblat, Ed. ASPLOS-V, New York, NY: ACM Press, 1992, pp. 51-61.

[2] G. Reinman, B. Calder, and T. Austin, “Optimizations enabled by a decoupled front-end architecture,” in *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 338–355, April 2001.

[3] A. J. Smith, “Cache memories,” *ACM Computing. Surveys*, vol. 14, no. 3, pp. 473-530, Sept. 1982.

[4] T. M. Conte, K. N. Menezes, P. M. Mills and B. A. Patel, “Optimization of instruction fetch mechanisms for high issue rates,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 333-344.

[5] P. Oberoi and G. Sohi, “Out-of-order instruction fetch using multiple sequencers,” in *Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*, Washington, DC, 2002, p. 14.

[6] P. S. Oberoi and G. S. Sohi, “Parallelism in the front-end,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pp. 230-240.

[7] J. Stark, P. Racunas, and Y. N. Patt, “Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order,” in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, Washington, DC, 1997, pp. 34-43.