Improving On-chip Data Cache Using Instruction Register Information

By

Lau Siu Chung

Supervised by: Dr. Chi Chi Hung

Submitted to Department of Computer Science and Engineering

in partial fulfillment of the requirement for the

Master of Philosophy

at the Chinese University of Hong Kong

June 1996

Dedicated to Sharon...

ABSTRACT

With advance of CMOS technology, the difference of speed between processors and main memories is getting larger and larger. The introduction of fast cache memories offers great help to tackle the problem. Because of their high speed and cost, the cache memories found in current computer systems are usually small in size. In order to fully utilize these small cache memories, numerous hardware and software based prefetching schemes have been proposed in the past two decades. In this dissertation, we propose and evaluate a collection of three hardware cache control schemes, called the Instruction Opcode and Addressing Mode Prefetching (IAP) scheme, that is based on the run-time information obtained from the instruction register. It is found that in the latest processor architecture design, there are certain instruction opcodes and addressing modes containing valuable information about how data will be referenced in the near future. Furthermore, these features have already been fully utilized by the current compiler optimization techniques to produce highly efficient program codes. Since most of these processors have on-chip data caches, it would be beneficial for the on-chip cache controllers to use this information for efficient cache management. In this dissertation, we will present [1] the problems of the memory systems that most of current computer systems have, [2] the observation that gives us the motivation of our scheme, [3] the evolution of our scheme from an intuitive idea to our final proposal, and [4] the experimental results that show the substantial performance improvements obtained from our scheme over the past prefetching schemes.

Acknowledgments

Just want to say THANKS.

Contents

Abstract i

Acknowledgment ii

List of Figures v

Chapter 1 Introduction 1

1.1 Hiding memory latency 1

1.2 Organization of dissertation 4

Chapter 2 Related Work 5

2.1 Hardware controlled cache prefetching 5

2.2 Software assisted cache prefetching 9

Chapter 3 Data Prefetching 13

3.1 Data reference patterns 14

3.2 Embedded hints for next data references 19

3.3 Instruction Opcode and Addressing Mode Prefetching scheme 21

3.3.1 Basic IAP scheme 21

3.3.2 Enhanced IAP scheme 24

3.3.3 Combined IAP scheme 27

3.4 Summary 29

Chapter 4 Performance Evaluation 31

4.1 Evaluation methodology 31

4.1.1 Trace-driven simulation 31

4.1.2 Caching models 33

4.1.3 Benchmarks and metrics 36

4.2 General Results 41

4.2.1 Varying cache size 44

4.2.2 Varying cache block size 46

4.2.3 Varying associativity 49

4.3 Other performance metrics 52

4.3.1 Accuracy of prefetch 52

4.3.2 Partial hit delay 55

4.3.3 Bus usage problem 59

4.4 Zero time prefetch 63

4.5 Summary 67

Chapter 5 Conclusion **68**

5.1 Summary of our research 68

5.2 Future work 70

Bibliography **71**

List of Figures

Figure 3.1: Example of array transversal 17
Figure 3.2: Example of 2-D array transversal 18
Figure 3.3: Example of 2-D array transversal with memory placement stride 19
Figure 3.4: Operations of *LOAD/STORE-UPDATE* instructions 21
Figure 3.5: Architectural model for IAP scheme 22
Figure 3.6: Control flow of the basic IAP scheme 23
Figure 3.7: Multiple blocks prefetching 25
Figure 3.8: Next Cache Block vs. Next Data Prefetching 26
Figure 3.9: Control flow of the enhanced IAP scheme 28
Figure 3.10: Control flow of the combined IAP scheme 30

Figure 4.1: Trace-driven simulator using *xtrace* 33
Figure 4.2: Memory model of the simulator 35
Figure 4.3: Percentages of *LOAD/STORE-UPDATE*s in SPEC92 Benchmark Suite 39
Figure 4.4: Baseline CPIs of SPEC92 Benchmark Suite 41
Figure 4.5: MCPI and stall reduction of simulations with varying cache size 45
Figure 4.6: MCPI and stall reduction of simulations with varying block size 47
Figure 4.7: MCPI and stall reduction of simulations with varying associativity 50
Figure 4.8: Accuracy of prefetching 54
Figure 4.9: Percentage partial hit delay to overall delay for the basic IAP scheme 57
Figure 4.10: Partial hit delay reduction over the basic IAP scheme 59
Figure 4.11: Retry after prefetch killed 62
Figure 4.12: Number of successful prefetch blocks over the basic IAP scheme 63
Figure 4.13: Experimental results with zero time prefetch for espresso 66
Figure 4.14: Experimental results with zero time prefetch for nasa7 67

Chapter 1

Introduction

Starting in the 1990s, single chip processors have become the basic building block for high performance computer systems, ranging from personal workstations, mainframes to highly scalable parallel processing systems. With the maturity of CMOS technology, today's processors already have over 3 million transistors and run at a clock speed of over 100 MHz. It is expected that in the next three to five years, processors with more than 5 to 10 million transistors and with clock speed of 300 to 500 MHz will be available commercially. At the same time, the memory latency and bandwidth have also been improved, but in a much slower rate. Due to cost and practical reasons, the speed gap between the processors and the memory systems is kept increasing and the cache miss penalties become relatively larger. When the cache hit rate is low, the processor will be stalled most of the time, waiting for data to be fetched from the memory. This results in low processor utilization. It is not difficult to observe that in future high performance processors and computer systems, their primary performance bottleneck will be in their supporting memory systems [HeP90] [Smi82]. To solve this speed gap problem, one common approach is to use on-chip caches [Smi82]. It is generally agreed that the performance of on-chip caches is one of the primary factors to determine the overall system performance [HeP90].

1.1 Hiding memory latency

It is clear that the performance of a computer system will be greatly improved if the memory latency is “hidden” from processor execution. That is, the processor is kept running while data are fetched from or written back to the memory. Numerous techniques have been proposed on this topic, for example, compiler(code) optimization, which reduces the data and control dependence of program codes in order to minimize the processor stalls [GiM86]; write buffer, which allows the processors to keep running during memory update [Smi82].

One important technique to improve the system performance is cache prefetching [Smi78a] [Smi78b] [Smi82] [HeP90]. By fetching data into the cache before they are actually used, delay due to memory accesses might be able to overlap with the program execution time, thus resulting in less processor idle time. In superscalar processors and future processors, data prefetching will become more important. It is because the consumption rate of data and instructions per unit time increases as the total number of functional units increases, and hence increasing the demand for better cache performance and higher chip I/O data transfer rate.

For instruction references, even simple prefetching schemes like "one block look ahead" and "prefetch-on-miss" [Smi82] are good enough to provide reasonable cache performance. This is due to the strong sequential nature of instruction references. In fact, many cache design techniques and control mechanisms such as prefetch buffer and large cache block size are based primarily on the instruction reference characteristics. However, for data references, most of current data prefetching schemes lose their effectiveness. It is because the reference pattern of data is considered to be "random" instead of "sequential" and is much less predictable when compared to the reference pattern of instructions. As the accuracy for data prefetching decreases, cache pollution can be so serious that any gain in data prefetching might be offset by the loss due to the replacement of "useful" information in cache by "'useless prefetched" information. The demand for extra bus bandwidth by the inaccurate data prefetch requests also contributes to the degradation of cache performance.

In this dissertation, we propose a novel cache control scheme, called the **Instruction Opcode and Addressing Mode Prefetching (IAP),** to perform very accurate hardware driven data cache prefetching. It is found that in the design of latest processor architectures, there are certain instruction opcodes and addressing modes containing valuable information about how data will be referenced in the near future. For example, each *LOAD/STORE-UPDATE* instruction in the IBM RS/6000 [IBM89] and PowerPC series [Mot92] [WeS94] [IBM94] or *LOAD/STORE-MODIFY* instruction in the HP Precision Architecture 1.1 [HP94] updates the content of one register used in the address calculation for the current data reference and it is expected that this updated register content will be used in the address calculation for some data references by the same instruction in the near future. Furthermore, these new features have already been fully utilized by current compiler optimization techniques to produce highly efficient program code. Since most of these processors have on-chip data caches, it would be beneficial for the on-chip cache controllers to use this information for efficient cache management. Our experiments on the SPEC92 showed that with proper fine tuning of the IAP scheme, the processor idle time due to memory accesses can be reduced substantially, yet the additional hardware required is very simple. This IAP scheme is especially good in prefetching array data references with constant strides, where the opcode used is the *LOAD/STORE-UPDATE,* the addressing mode used is the index-displacement, and the register used in the address calculation is updated by a constant displacement specified in the instruction.

1.2 Organization of dissertation

The organization for the rest of this dissertation is as follows:

In Chapter 2, a brief survey on the current data prefetching schemes will be given. Introductions to both the hardware controlled and software assisted cache prefetching will be included.

In Chapter 3, our three IAP prefetching schemes — basic, enhanced and combined, will be presented. In the basic IAP scheme, prefetch requests can be generated for data with one iteration look ahead. Next, the concept of cache block prefetching is introduced in the enhanced IAP scheme to tackle the problem of limited memory bus bandwidth. Finally, the combined IAP scheme, equipped with default prefetching, try to exploit the spatial locality left by the former two schemes. The general architectural model and control flow diagrams for the IAP schemes will also be included in this chapter.

Chapter 4 will give the evaluations of the results of our experiments. The IAP schemes are evaluated using trace-driven simulations of eight SPEC92 benchmarks based on the architecture of an IBM RS/6000 machine. With comparison to the traditional hardware controlled prefetching scheme — prefetch-on-miss, the results show that the IAP schemes are generally more effective in reducing the memory latency time by performing very accurate data prefetching.

Finally, the dissertation will be concluded in Chapter 5. Future development and directions of the research will also be suggested in this chapter.

Chapter 2

Related Work

The concept of data cache prefetching is not new. The idea of fetching data into the cache before they are used was first suggested in the early 1970s [Sak72]. Since then, proposals on various cache prefetching schemes have been proposed and countless efforts have been spent to fine-tune the effectiveness of cache prefetching. In general, cache prefetching schemes can be classified either as hardware controlled or software assisted.

2.1 Hardware controlled cache prefetching

Most hardware controlled cache prefetching schemes are based on the sequentiality property of references [Smi78a] [Smi78b] [Smi82] [Lee87] [Jou90] [FuP91]. It is suggested that when datum i is referenced, the probability for datum $i+1$ to be referenced in the near future is very high. Thus, datum $i+1$ is a good candidate for data prefetching. Under this approach, the two most commonly used prefetching techniques are large cache block size and one block look ahead.

Cache block size is the basic unit of data transferred between the cache and the main memory. When the cache block size is greater than one, data cache prefetching will be achieved whenever a cache miss due to the reference of a single datum occurs. The missing datum is fetched into the cache on demand while other data in the same cache block are prefetched into the cache. Cache performance will be improved if the prefetched data in the cache block are referenced. For instruction caches, large cache block sizes (e.g. 16-32 bytes) are often found to be useful in improving cache performance. However, this is not true for data cache. As the cache block size increases, the average percentage of data in a cache block that will be referenced before the block is replaced might decrease very quickly. For example, for array references with large strides, since only one datum in each cache block might be referenced, increasing the cache block size to perform data prefetching does not improve data cache performance. On the other hand, the number of cache blocks that can be stored in a fixed size cache decreases as the block size increases. This might result in the degradation of cache performance because the amount of useful data that can be stored in cache decreases. Furthermore, transferring a larger cache block would take longer time than transferring a smaller cache block — another performance overhead. All these factors impose a practical limit to the block size of data caches. In fact, the technique of using large block size for data prefetching has been shown to be ineffective in data caches [Lee87] and small block size (e.g. 4 bytes) is preferred, especially in multiprocessors.

One block look ahead [Smi82] is the other commonly used technique in hardware controlled cache prefetching schemes. The basic idea of this technique is to prefetch the next cache block from the current one being referenced. This prefetching action can be triggered by different situations. Some of the common ones are:

[1] when a cache miss for a cache block occurs,

[2] when the beginning of a cache block is referenced, or

[3] when the end of a cache block is referenced.

Again, this technique is based on the sequentiality property of references and it only works for instruction caches, not for data caches. In data caches, since the chance for the prefetched data to be actually referenced is not high, prefetching might degrade cache performance. It is because placing prefetched cache blocks into the cache means replacing some blocks from the cache.

The basic reason why these two techniques do not work in data caches is that the reference behavior of data, which is actually a mix of portions of different reference patterns (see the next chapter for more details), is very different from the reference behavior of instructions, which is highly sequential. Applying prefetching schemes that are designed primarily for one type of reference behaviors to another type of reference behaviors, of course, will not be effective.

Some variations of one block look ahead scheme have also been studied in many previous researches. As proposed by Fu and Petal [FuP91], stride information carried by vector instructions can be used for prefetching data in the vector processors. Instead of only one block look ahead, multiple blocks are prefetched on a cache miss. The cache load size l is defined as the number of bytes loaded into the cache when a miss occurs and it will be equal to $(p+1)*b$ where p is the number of blocks prefetched on a miss and b is the basic block size. Two prefetch-schemes are suggested based on the cache load size:

[1] the sequential-prefetch scheme — when a miss occurs, the cache prefetches p **consecutive** blocks for a reference which is a scalar or a short stride (i.e. the stride < b) array access;

[2] the stride-prefetch scheme — in addition to sequential-prefetching, the cache prefetches p blocks for long stride (i.e. the stride > b) vector accesses when a cache miss occurs, where the blocks are separated by the stride.

After performing some simulations on a 64K cache (32-byte block size, 2-way set associative) with a load size of 128 bytes, the sequential-prefetch scheme improves the system performance improvement by 30%-50%, while the stride-prefetch scheme shows no significant improvement over the sequential prefetch.

After that, Fu and Patel [FuP92] carried on their work to the scalar processors. In this scheme, data prefetching is based on the prediction of the execution of the instruction stream. The prefetch requests are generated by a hardware history table, which records the instruction address of *LOAD/STORE* instructions and the corresponding memory addresses requested. Whenever a *LOAD/STORE* instruction is encountered, its instruction address is checked with those from the table. If it is found in the table, the stride is then calculated as the difference between the previous and current memory address. A prefetch request will be sent out for the next datum with memory address equal to the sum of current memory address and the stride. Their results show a significant improvement for vectorized programs. That is, when the data references of the programs are mainly of regular strides. However, due to the lack of control of preventing prefetches for irregular data references, many unnecessary prefetches will be generated and they may displace useful data out of the cache. As a result, the performance improvement drops quickly for the non-vectorized programs. Moreover, with only one iteration look ahead, the memory bus may not have enough time to finish the prefetch request before the data are actually needed by the processor, introducing another performance loss.

In order to tackle the above problems, Baer and Chen [BaC91] [ChB92] [FuP91] from the University of Washington suggested their Data Preloading scheme. The hardware for this data preloading scheme consists of a reference prediction table (which is a refined history table) and a look ahead program counter and its associated logic. Each entry in the reference prediction table consists of a tag, a previous address, stride and the state of preloading. As compared to the history table, the stride, in addition to the instruction address and memory address, is also recorded in the reference prediction table. The regular access pattern of data references is confirmed by comparing the sum of the previous address and the stride in the reference prediction table to the current reference address. If they are equal, a prefetch request for datum with address equal to the sum of the current data reference address and the stride is sent out to the cache. With this scheme, preloading of data with regular data reference patterns can be achieved while preloading of data with irregular data reference patterns can be avoided. The look ahead program counter is used to allow multiple iterations look ahead. The result of this data preloading technique is very good and is very useful in scientific computation applications, where data references mainly consist of array references with constant strides. The major drawback of this scheme, however, is the size of the reference prediction table required. It is reported [BaC91] that the size of the reference prediction table needed to achieve good data preloading result is about half of the cache size. This makes the preloading technique more suitable for the second level off-chip cache than for the first level on-chip cache because the on-chip space available for data cache is always very limited.

2.2 Software assisted cache prefetching

In the past few years, cache designers start to look for new ways to improve the effectiveness of data cache prefetching. With advances in compiler optimization and in data flow analysis, software assisted cache prefetching using *PREFETCH* instructions is now possible. Recently, a number of software assisted (or compiler driven) cache prefetching schemes [Tha81] [Bre87] [Por89] [GoG90] [CaK91] [ChM91] [KlL91] [MoG91] [MoL92] have been proposed. All these schemes share some common properties:

[1] Some non-blocking *PREFETCH* instruction is defined to preload a block of data into the cache.

[2] *PREFETCH* instructions are inserted into some inner loops of a program by the compiler.

[3] Prefetching candidates are array references with constant stride.

All these software assisted prefetching schemes are very successful in prefetching array references with constant strides. However, the use of these schemes is often limited by their runtime overhead. When *PREFETCH* instructions are inserted into some inner loops of a program, program execution time will be spent to execute these instructions, independent of whether these *PREFETCH* instructions can help eliminating cache misses. For any data caches with block size greater than one, the **same *PREFETCH*** instruction in some loop of a program might execute more than once, each time to prefetch the same cache block. In other words, to avoid one cache miss, the runtime overhead introduced by these schemes might range from 1 instruction to *block_size* instructions, where *block_size* is the size of the cache block. In fact, Porterfield reported in [Por89] that using the computing intensive programs in RiCEPS as the benchmark programs, he found that the percentage of *PREFETCH* instructions that are found to be useful only ranges from 1.7% to 58.2%, with the average[1] of 28.4%. However, the software prefetching overhead (in execution time) introduced is substantial, ranging from 6% to 34%, with the average of 28%.

To reduce this runtime overhead of software assisted prefetching schemes, Mowry and Lam [MoL92] proposed the concept of prefetch predicates, which determines if a particular iteration needs to be prefetched. Then, with the loop splitting

[1] Even with overflow iteration technique [Por89], the average percentage of useful *PREFETCH* instructions can only be improved to about 60%.

technique, the runtime overhead of data prefetching is reduced by decomposing the loops into different sections so that the predicates for all instances for the same section evaluate to the same value. This implies that either the first iteration of the loop is peeled for temporal locality reason or the loop is unrolled by a factor of the cache block size for spatial locality reason. This prefetch predicate concept definitely improves over previous software prefetching schemes. However, there are a number of important issues which still need to be solved by this predicate approach. As mentioned in their paper, for large cache block size, peeling and unrolling multiple levels of loops can potentially expand the code by a significant amount; also existing optimizing compiler is often ineffective for large procedure bodies. On the other hand, for small cache block size, the amount of runtime overhead that can be reduced by this technique is small because their improvement factor is a linear function of the cache block size. In fact, when the cache block size is one or when the distance between two successive array references[2] is greater than the cache block size, no reduction in runtime overhead can be achieved. Furthermore, the algorithm for this peeling and unrolling for prefetch predicates is quite complicated to be implemented in the compiler.

One other complication of using *PREFETCH* instructions is the amount of data that should be prefetched by one *PREFETCH* instruction. Since there is some runtime overhead associated with each *PREFETCH* instruction execution, one would prefer to prefetch a larger block of data per instruction. In this way, the overall runtime overhead of the scheme can be reduced. However, if a large block of data is prefetched into the cache, the current working set of references in the data cache might be destroyed by the prefetched data (especially for on-chip caches with a smaller cache size). This will

[2] This is the case for stride Row_Size or Column_Size array references and is found to be very common.

introduce additional cache misses. Furthermore, execution of the *PREFETCH* instruction for a larger block of data might hold up the memory bus for too long and this can cause unnecessary delay to other bus requests such as demand-driven cache misses. One the other hand, if each *PREFETCH* instruction only prefetches a small block of data, the performance gain by the software prefetching scheme might not be large enough to offset its overhead.

People have been arguing that while the cost issue of software assisted data prefetching in scalar machines is important, the cost of software cache prefetching in superscalar machines is negligible. It is because the *PREFETCH* instructions will be fallen into unused slots for free. We do not agree with this argument. First, the *PREFETCH* instructions are not scheduled to fill in the free slots available. Instead, they will be executed whenever they are encountered (because there is no data dependency). Thus, the execution of *PREFETCH* instructions cannot be considered as free. Rather, they should be treated as the execution for some additional data dependent free instructions. Second, even if some free slots are found, they are still not free. If the same slot is not occupied by the *PREFETCH* instruction, it can be used by some other instructions. Decoding one *PREFETCH* instruction in superscalar machines also implies that one fewer instruction is looked ahead and this can potentially reduce the parallelism width.

Unless the problem of runtime overhead associated with *PREFETCH* instructions is solved, software assisted cache prefetching schemes can only be used in some very limited and specific situations such as some hand optimized engineering library routines for vector processing.

Chapter 3

Data Prefetching

As we discussed in the last chapter, current prefetching schemes are not very effective in reducing the processor idle time due to data memory accesses. Traditional hardware driven data prefetching schemes are simple to implement, but the prefetching accuracy is usually not high enough to get significant improvement in data cache performance. Accurate hardware driven prefetching of array data references with constant strides is possible, but the hardware cost such as the reference prediction table is often too high for them to be used in the first level on-chip data cache. Software cache prefetching schemes can also be very accurate in prefetching array data references with constant strides, but the runtime overhead of these schemes often limits their practical use. Furthermore, software assisted prefetching schemes need architectural support (such as the definition of some new *PREFETCH* instructions) and new compiler support[3]. These might make the use of software assisted data prefetching schemes in current processors and computer systems difficult.

In this chapter, we propose a new cache control mechanism, called the **Instruction Opcode and Address Mode Prefetching (IAP),** to improve data cache performance. The unique feature of this IAP scheme is that very accurate data prefetching can be carried out using the runtime information in the instruction register. The motivation of our schemes and the general data reference patterns will be presented in Section 3.1. Section 3.2 will give a deeper understanding on how current processor architecture embeds data reference hints in the instruction opcodes and addressing

[3] Note that the compiler optimization tcchniques to support data cache prefetching for array referenccs with constant strides are not too difficult to implement, but they need to be added to the current compiler optimizers. Furthermore, fine tuning of software cache prefetching scheme such as loop splitting will add another level of complexity to the compiler.

modes. Then, in Section 3.3, our three IAP schemes — basic, enhanced and combined will be proposed. Finally, a summary of this chapter will be given in Section 3.4.

3.1 Data reference patterns

As we have discussed earlier, the reference behavior of data is actually a mix of portions of different reference patterns and this solely accounts for the failures of the traditional hardware prefetch techniques. It is constructive for us to have a closer look on these different reference patterns in order to have more insight for a successful prefetching scheme.

As suggested by Baer and Chen [BaC91], the data references can be generally classified into one of the four categories: scalar, zero stride, constant stride and irregular. Suppose there is a program segment with m-nested loops with index i_1 , i_2,..., i_m and LP_j be the set of instructions with data references in the loop of level j.

[1] Scalar type refers to those references to the simple non-array variables such as the indexes of the loops and those variables used as the counters.

[2] Zero stride type refers to those references to the indexed array elements with the indexes being unchanged in the inner loops but will be modified in the outer loops (that is, it is zero stride in the inner loop but not in the outer loop). For example, $a[i_1,i_2]$ and $RECORD[i_2].element$ are zero stride in LP_3.

[3] Constant stride type refers to those references to the indexed array elements with the indexes being increased (or decreased) linearly with the indexes of the loops. For example, $a[i_2]$, $a[i_1,i_2]$ and $a[i_2,i_1]$ are constant stride in LP_2.

[4] Irregular type refers to those references to the variables with strides between each iteration but the strides are not constant, for example, *a[b[i_1]]* in *LP_1* and some link-listed variables.

For the scalar and zero stride references, the references for the next iteration are just the same as current references. As the stride values of irregular type references are kept changing from iteration to iteration, it is difficult to predict those stride values and to find out the memory locations of next data references. Therefore, these types of references offer us no hint to issue prefetch request. On the other hand, the strides of the constant stride type references will remain unchanged for the whole loop execution. It is easier to discover the stride values, find out the locations of next data references and issue accurate prefetch requests. Obviously, a secure way to a successful prefetching scheme is to single out the constant stride references from all the other data references and generate prefetch requests for the corresponding next data references.

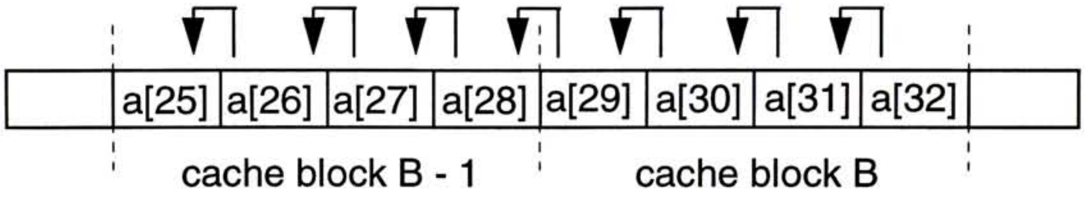
Beside singling out the constant type references, the next important issue is to find out the strides of the references, since one can determine the location of the cache block needed to be prefetched with the current memory address and the stride value. However, this important information is simply neglected by one block look ahead strategies and their prefetching actions are solely based on the memory block addresses only. As a result, caches with these strategies can only generate correct prefetch requests for constant stride data references when the strides are small and positive. However, they may fail for cases like the following examples (all examples are based on the assumption of 4 bytes element size and 16 bytes memory block size):

[1] when the direction of references is opposite to the memory placement — consider the program segment in Figure 3.1(a), the elements of the array *a* is referenced in a

descending order which is usually opposite to the placement of data in memory. In memory block level, the referencing pattern in memory will be block *B*, block *B-1*,..., block *B-7*, as illustrated in Figure 3.1(b). If one block look ahead strategy is used, block *B+1* will be prefetched when block *B* is referenced, where the prefetched block is not included in the array reference. The reference to block *B-1*, block *B-2*,... and block *B-7* will also issue a prefetch to a former referenced block which will do no help to facilitate the memory traffic.

```
int a[32];
for i = 31 to 0
   a[i] = a[i] + 1;
```

(a) an array transversal



(b) memory reference pattern

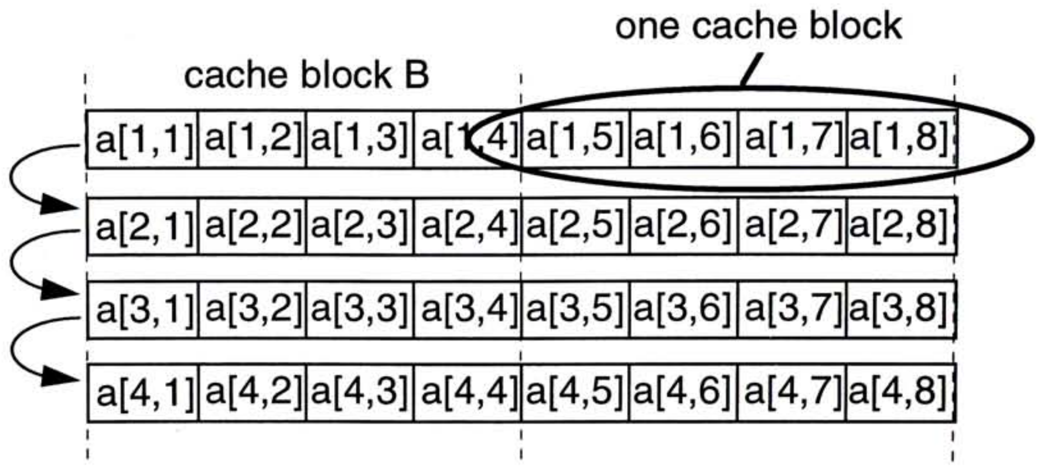
Figure 3.1: Example of array transversal

[2] when the stride is larger than one memory block size — for example (as illustrated in Figure 3.2), there is a two-dimensional array (a matrix) *a* and the elements of *a* are packed in memory in a row by row manner. That is, starting with *a[1,1]*, *a[i,j+1]* follows *a[i,j]* when *j+1* < *8* (i.e. the row size) and *a[i+1,1]* follows *a[i,8]*. When the data are referenced with advancement of one row at a time, the stride will be equal to 32 bytes (i.e. *size_of_one_element* * *number_of_elements_each_row*) which is larger than the size of a memory block. In this example, each row is perfectly fitted into two memory blocks and the referencing pattern will be like

block *B*, block *B+2*, block *B+4*,..., block *B+14*. With one block look ahead strategy, only useless blocks *B+1*, *B+3*,... and *B+15* will be prefetched, which will not be referenced in this array transversal and may displace useful data out of the cache.

```
int a[8,8];
for i = 1 to 8
   a[i,1] = a[i,1] + 1;
```

(a) a two-dimensional array transversal


one cache block
cache block B
a[1,1] a[1,2] a[1,3] a[1,4] a[1,5] a[1,6] a[1,7] a[1,8]
a[2,1] a[2,2] a[2,3] a[2,4] a[2,5] a[2,6] a[2,7] a[2,8]
a[3,1] a[3,2] a[3,3] a[3,4] a[3,5] a[3,6] a[3,7] a[3,8]
a[4,1] a[4,2] a[4,3] a[4,4] a[4,5] a[4,6] a[4,7] a[4,8]

(b) memory reference pattern

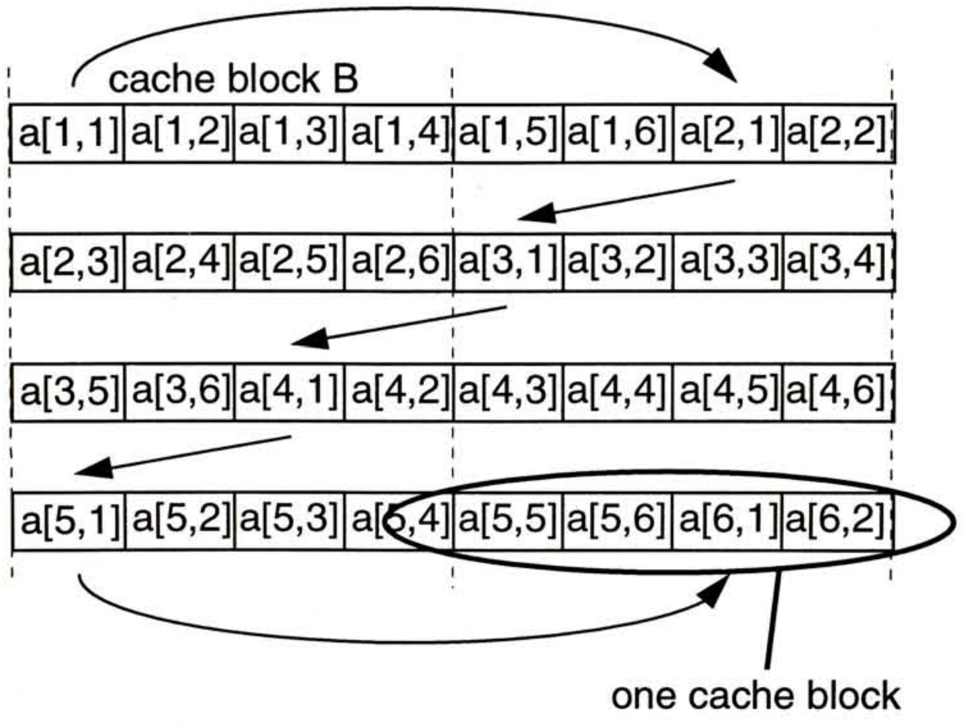
Figure 3.2: Example of two-dimensional array transversal

[3] when the stride or block size is not a multiple of another — if the block size is a multiple of (or equal to) the stride, the memory block will be referenced one by one and the reference pattern will be like block *B*, block *B+1*, block *B+2*,... (or block *B*, block *B-1*, block *B-2*,... depending on the direction of the stride). On the other hand, if the stride is a multiple of the block size, the reference pattern will be like block B, block *B+j*, block *B+2*j*,... (or block *B*, block *B-j*, block *B-2*j*,...), where *j* = *stride_size/block_size*. In these two cases, their patterns are still quite "regular". However, the situation becomes more complicated when the stride or block size is not a multiple of the other one (that is, the elements are placed in the memory with

placement strides between the rows). For example, as shown in Figure 3.3, when the stride = 6 * 4 bytes = 1.5 * *block_size*, the memory reference pattern will be block *B*, block *B+1*, block *B+3*,.... The regularity of this constant type reference is somewhat blurred out in the memory block level and this "irregular" pattern puts further burden on the job of prefetching.

```
int a[8,6];
for i = 1 to 8
   a[i,1] = a[i,1] + 1;
```

(a) a two-dimensional array transversal


cache block B
a[1,1] a[1,2] a[1,3] a[1,4] a[1,5] a[1,6] a[2,1] a[2,2]
a[2,3] a[2,4] a[2,5] a[2,6] a[3,1] a[3,2] a[3,3] a[3,4]
a[3,5] a[3,6] a[4,1] a[4,2] a[4,3] a[4,4] a[4,5] a[4,6]
a[5,1] a[5,2] a[5,3] a[5,4] a[5,5] a[5,6] a[6,1] a[6,2]
one cache block

(b) memory reference pattern

Figure 3.3: Example of two-dimensional array transversal with memory placement stride

Up to this moment, the importance of classifying the data references and determining the strides of data references to accurate prefetching is discussed. In next section, we will go on to present how the data reference information is embedded in instruction opcodes and how this information can be extracted and used to help issuing accurate prefetch requests.

3.2 Embedded hints for next data references

It is very interesting to find out that in the design of latest processor architectures, the instruction set often has some mechanism to support the address calculation for the expected future data references while the current datum is being referenced. And these built-in features are usually found in the instruction opcodes and the address modes of the architecture definition.

In RISC architecture, one technique to reduce the program execution pathlength is to use compound instructions. From the program instrumentation and tracing, it is found that certain simple RISC instructions often execute as a pair. Thus, it might be useful to define a single compound or extended opcode that can execute the instruction pair. This is especially useful if the new instruction opcode does not affect the processor clock cycle. For example, HP's Precision Architecture 1.1 has *ADD-AND-BRANCH, COMPARE-AND-BRANCH, LOADWORD-AND-UPDATE,* etc.; IBM RS/6000 and PowerPC has *LOAD-UPDATE, STORE-UPDATE, LOAD-MULTIPLE,* etc.. Note that despite of the superscalar architecture design, many latest processors find these compound or extended opcodes to be very useful. As a result, the total number of instructions defined in current RISC processors ranges from 150 to 200, which is much larger than the number of instructions defined in early RISC processors (about 50 to 70 instructions).

One type of compound instructions that we found to be very helpful in managing on-chip cache activities is *LOAD/STORE-UPDATE* (or *LOAD/STORE-MODIFY)* instruction. In typical programs, one major type of data references is array or pointer references to a large set of data, that is, the constant stride type. Usually, this constant stride type of accesses is done using "index-displacement" addressing mode and data will be referenced successively one after the other (e.g. in some loops). Since this occurs so frequently, many architectures have some mechanisms to speed up this data reference process. For example, in RISC processors such as HP's Precision Architecture 1.1, IBM RS/6000 and PowerPC series, compound opcodes such as *LOAD-UPDATE* and *STORE-UPDATE* are defined. Besides loading or storing a datum into a register, each of these instructions will update the content of one register used in the address calculation for the current data reference with the current effective data reference address. Figure 3.4 shows the operations of the *LOAD-UPDATE* and *STORE-UPDATE* instructions using "index-displacement" addressing mode.

(a)	(b)
LDU R_T *,*(R_x*+Disp)*	*STU* R_T*,(*R_x*+Disp)*
Equivalent to	*Equivalent to*
Eff. Addr. = (R_x)*+Disp*	*Eff. Addr.* = (R_x)*+Disp*
R_T = *(Eff. Addr.)*	*(Eff. Addr.)* = R_T
R_x = *Eff. Addr.*	R_x = *Eff. Addr.*

Figure 3.4: Operations of (a) *LOAD-UPDATE* and (b) *STORE-UPDATE* using the "index-displacement" addressing mode

The updating action of the *LOAD-UPDATE* or the *STORE-UPDATE* instruction is to prepare the content of *Rx* needed for the address calculation for the next expected data reference, which is equal to the sum of the current data reference address *Eff. Addr* and the displacement *Disp*. Thus, accurate data cache prefetching can be carried out and the prefetching data address is equal to *(Eff. Addr. + Disp)*. Note that the values of *Eff. Addr.* and *Disp* are available to the cache prefetching unit during the execution of the *LOAD* R_T*,*(R_x *+ Disp)* or *STORE* R_T*,*(R_x *+ Disp)* instruction.

3.3 Instruction Opcode and Addressing Mode Prefetching scheme

3.3.1 Basic IAP scheme

In the last section, we showed that certain instruction opcodes and addressing modes of current processor architectures actually contain information about how data will be referenced in the near future. Based on these hints of data references, we can now propose our IAP scheme — a scheme that uses these hints for accurate data prefetching for on-chip cache. To make our discussion easier, we will use the IBM POWER architecture (or the PowerPC architecture) here as an example to show how the IAP scheme should be designed and implemented. In the chapter of conclusion and future work, we will discuss how the proposed IAP scheme in this section can be extended to non-POWER architectures.

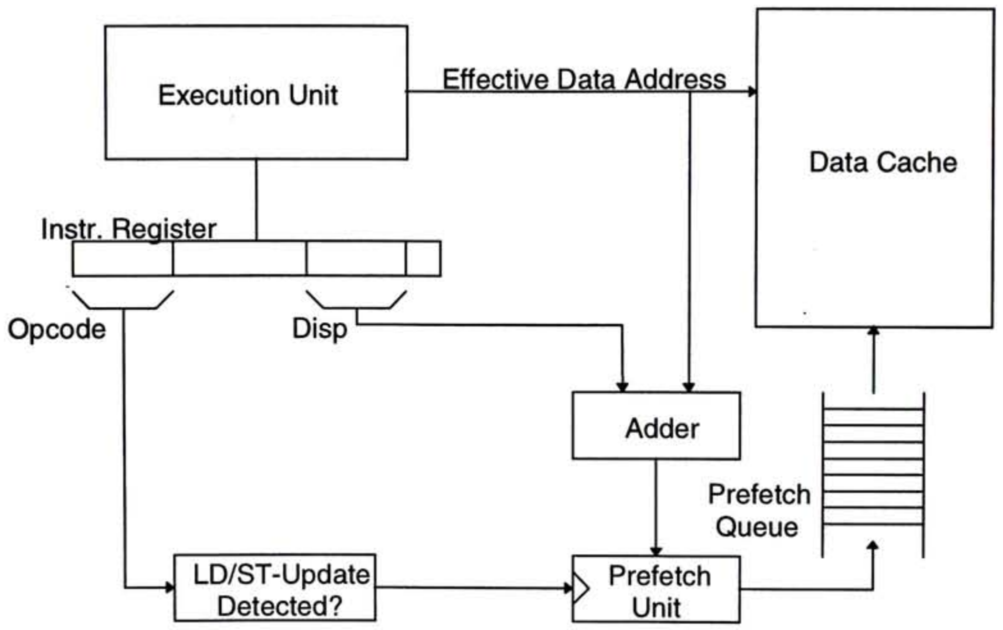


Figure 3.5: Architectural model for IAP scheme

Figure 3.5 shows the architectural model for the basic IAP scheme. For each instruction *I* that is decoded and executed, its opcode is checked to see if it belongs to the instruction type of *LOAD-UPDATE* or *STORE-UPDATE.* Whenever some *LOAD-UPDATE* or *STORE-UPDATE* instruction is detected, the address of the next datum that is expected to be referenced in the near future will be re-calculated using the same addressing mode of *I* but with the updated values of all register contents used in the address calculation of *I*. Then, with this new address, the prefetch unit is able to calculate the memory block address to be prefetched and this prefetch address will be sent to the cache prefetch unit for accurate data prefetching. For example, an instruction *LOAD-UPDATE* $R_2,R_1(1024)$ *is* executed and the content of R_1 is 1237. The addressing mode of this instruction *I* is "index-displacement", the index register is R_1, the displacement value *Disp is* 1024; and the current data reference address of this instruction is 2261 (i.e. 1237 + 1024). After the execution of instruction *I*, the value of R_1 *is* updated from 1237 to 2261 and an address of 3285 (i.e. 2261 + 1024) will be exported from the adder to the data prefetch unit and the corresponding memory block address will be calculated and sent to the prefetch queue for accurate data prefetching. The control flow of the basic IAP scheme is summarized in Figure 3.6.

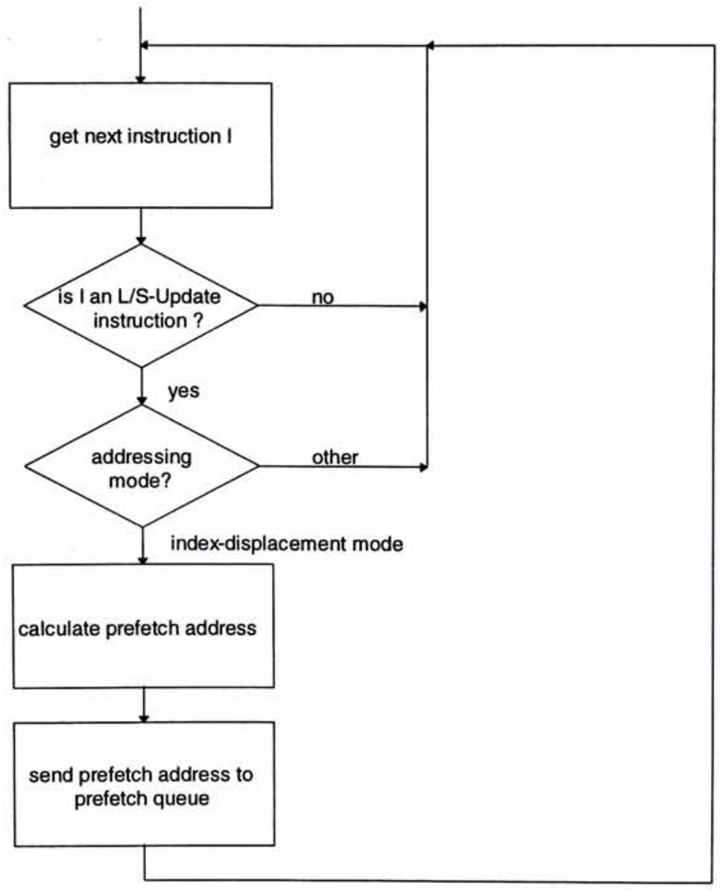

get next instruction I
is I an L/S-Update instruction ?
no
yes
addressing mode?
other
index-displacement mode
calculate prefetch address
send prefetch address to prefetch queue

Figure 3.6: Control flow of the basic IAP scheme

One point needed to be noted is that in all current cache prefetching schemes, the unit of data prefetching is always one cache block. For simple hardware driven cache prefetching schemes such as one block look ahead, the concept of unit cache block prefetching is obvious. In software assisted cache prefetching schemes such as the use of *PREFETCH* instructions, only the starting (byte or word) address of the datum to be prefetched is specified and only one cache block is prefetched[4]. However, the datum that is expected to be referenced in the near future might be contained in multiple cache blocks. It is because the size of the datum can be single, double, or quadwords, and the alignment of the datum in memory can start with any memory address. As a result, if the unit of cache prefetching is always assumed to be one cache block, it is possible that the first part of the datum can be prefetched very accurately into the cache, but the reference to the rest of the datum will cause cache misses. In cache designs where the cache block size is small or in architectures where *LOAD/STORE-MULTIPLE* instructions are available, this situation will become more serious.

For those cache prefetching schemes where the size of the prefetch datum is not available, there is nothing that can be done. On the other hand, in the IAP scheme (and also all software assisted cache prefetching schemes), the size of the prefetch datum as well as the size of the current data reference can be made available to the cache prefetch unit if it is required. Thus, to further improve the overall system performance by the basic IAP scheme, the concept of multiple cache blocks prefetching (Figure 3.7) is incorporated into the basic IAP scheme as follows.

[4] Note that in software prefetching schemes, if multiple cache blocks prcfetching is implemented, multiple *PREEETCH* instructions per datum will be needed. This will increase the runtime overhead further and will make the schemes less effective.

Multiple blocks prefetching:

For each *LOAD/STORE-UPDATE* instruction that is executed, the size of the prefetch datum from the IAP scheme is assumed to be the same as the size of the current data reference. Furthermore, both the starting memory address and the size of the prefetch datum will be sent to the prefetch unit. Then, either multiple cache blocks requests, each of which is for one cache block, will be put into the prefetch queue, or the data size will be tagged along with a single data prefetch request in the prefetch queue for accurate data prefetching[5].

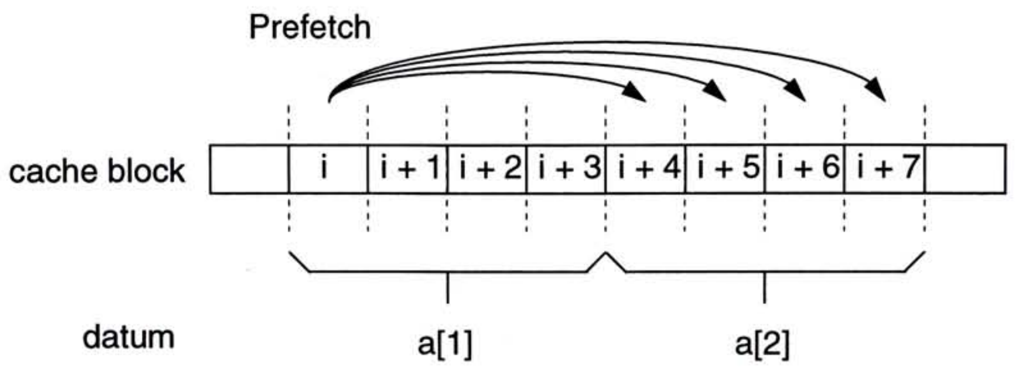


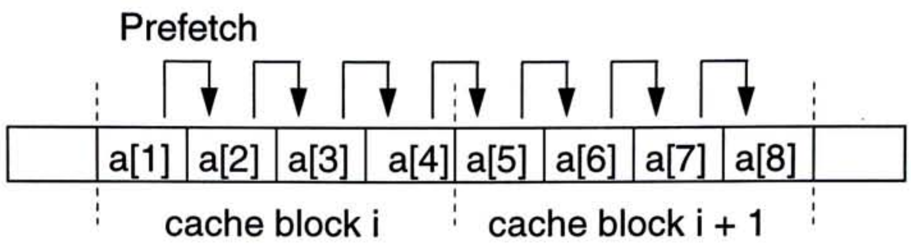
Figure 3.7: Multiple blocks prefetching

3.3.2 Enhanced IAP scheme

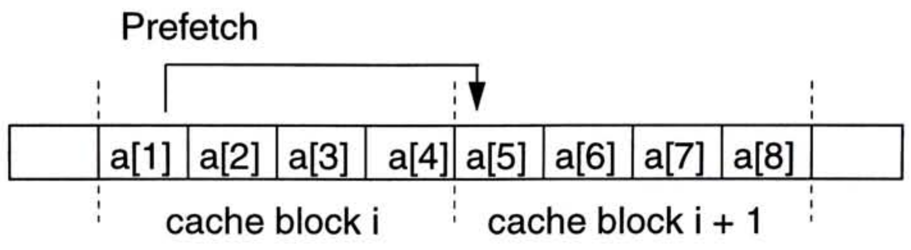
The basic model of the IAP scheme that we described so far seems to be very simple and straight-forwards — whenever a *LOAD/STORE-UPDATE* instruction is encountered, a prefetch request with address that is calculated using the same address mode of the instruction but with the updated register values will be issued to the prefetch queue. However, after some initial experiments of the IAP basic scheme, we found out that there are some hidden problems in the scheme. Detailed analysis showed that some

[5] Of course, this assumes that the memory unit can accept and use the information — starting memory address and the number of bytes to be fetched — for data fetching/prefetching.

enhancements needed to be made to tackle the problems and to have further performance improvements.



(a) next data prefetching



(b) next cache block prefetching

Figure 3.8: Next Cache Block vs. Next Data Prefetching

All current software(compiler) assisted data prefetching schemes and the basic IAP scheme proposed here try to prefetch the next datum that is expected to be referenced in the near future. If the prefetched datum and the current referenced datum are in the same cache block, no prefetching request will be issued. Consequently, this will create a potential problem of insufficient time for the prefetch requests to be finished in time for the data to be used by the processor. As an example, suppose an array of data is referenced using constant stride of 1 (shown in Figure 3.8) in a loop and the cache block size is 4 words (or 16 bytes). When *a[1]* is referenced, *a[2] will* be prefetched. Since *a[l]* and *a[2]* are in the same cache block *i*, no prefetch request will be issued. This situation will go on until datum *a[4] is* referenced. At this time, the next cache block *i+1* that contains *a[5]* will be prefetched. The potential danger of this approach is that if there is not enough free bus cycles between the accesses of *a[4]* and *a[5]* for the prefetching of cache block *i+1* to finish, additional processor stall time will be introduced. From our experiments, we found out that this actually happened very often. In order to make up for this situation, what we would like to have is to *predict the next data reference based on the data reference addresses, but to prefetch the next data reference based on the cache block addresses.* For example, when *a[1]* in cache block *i* is referenced, cache block *i+1* should be prefetched. This will give more time for the prefetch request to finish. It is because instead of one iteration look ahead prefetching, four iterations look ahead prefetching is resulted in this example. The enhancement of the IAP scheme to accommodate this is summarized as follows.

Enhanced IAP Scheme:

> For each data prefetch request that is generated by the IAP scheme, if the cache block *j* containing the prefetch candidate is not the same as the cache block *i* containing the current data reference, a prefetch request for block *j* will be issued. On the other hand, if the cache block *j* and the cache block *i* are the same, then the next cache block *i+1* (or *i-1* if the stride is negative) will be prefetched.

Figure 3.9 shows the control flow diagram of the enhanced IAP scheme.

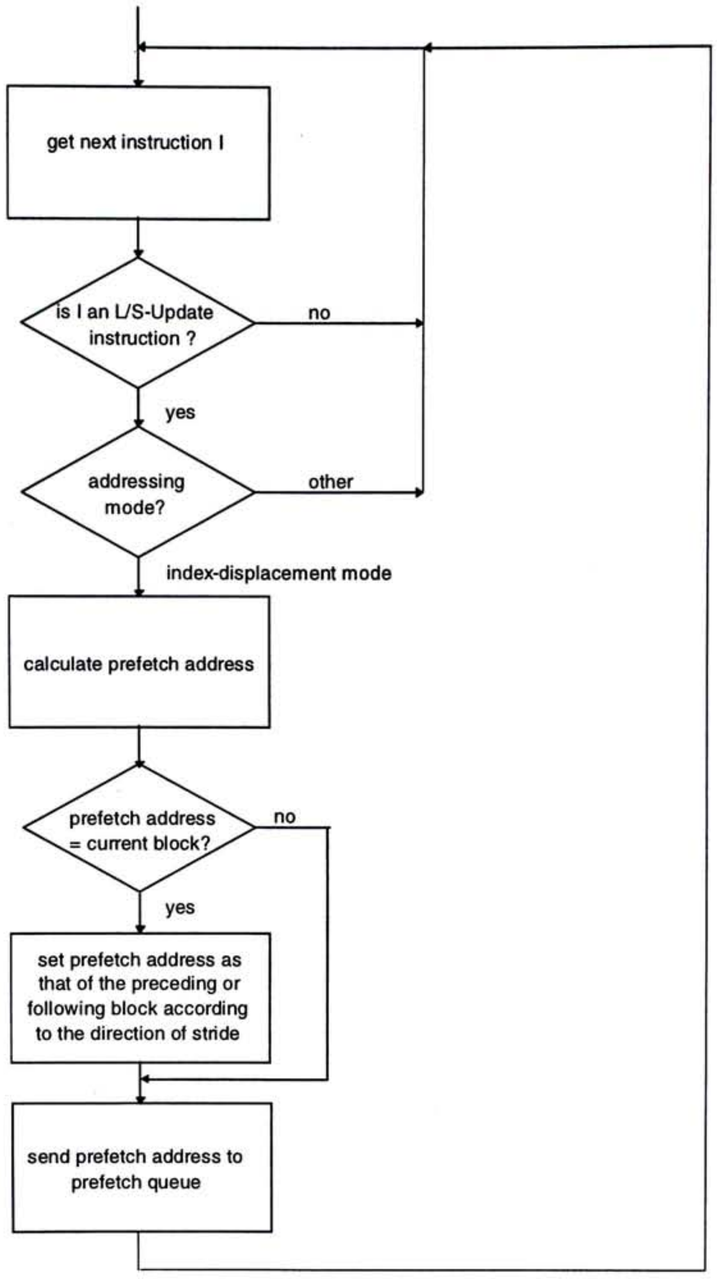

get next instruction I
is I an L/S-Update instruction ?
no
yes
addressing mode?
other
index-displacement mode
calculate prefetch address
prefetch address = current block?
no
yes
set prefetch address as that of the preceding or following block according to the direction of stride
send prefetch address to prefetch queue

Figure 3.9: Control flow of the enhanced IAP scheme

3.3.3 Combined IAP scheme

The goal of the IAP scheme is to perform very accurate prefetching on those data that are referenced using *LOAD/STORE-UPDATE* instructions. For other *non-LOAD/STORE-UPDATE* instructions, the IAP scheme will not issue any data prefetch request. The advantage of this selective prefetching by the IAP scheme is that very accurate data prefetching can be carried out and the potential problem of cache pollution by the IAP scheme can be minimized. However, there is some drawback to this approach — no cache prefetching is carried out for data that are referenced using *non-LOAD/STORE-UPDATE* instructions even though these data might process some degree of spatial locality and they will be benefited by simple prefetching schemes such as prefetch-on-miss. As a result, a default prefetching scheme is implemented on top of the IAP scheme and the enhancement can be summarized as follows.

Combined IAP scheme:

> For each *LOAD* or *STORE* instruction *I* that is executed, if *I* belongs to the *LOAD/STORE-UPDATE* instruction group, the enhanced IAP scheme will be used for data prefetching, else the default prefetching scheme — the prefetch-on-miss scheme will be used for data prefetching.

The control flow diagram of the combined IAP scheme is shown in Figure 3.10.

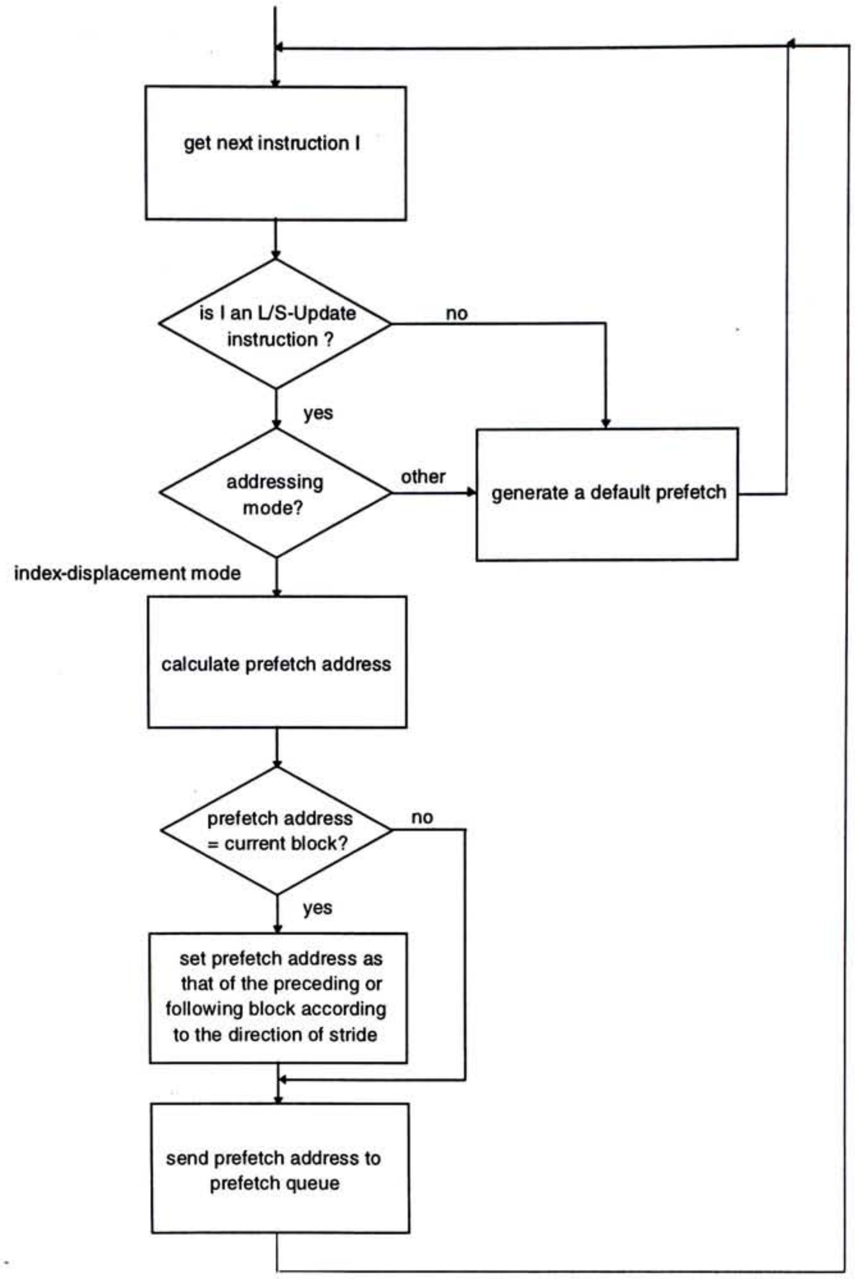

get next instruction I
is I an L/S-Update instruction ?
no
yes
addressing mode?
other
generate a default prefetch
index-displacement mode
calculate prefetch address
prefetch address = current block?
no
yes
set prefetch address as that of the preceding or following block according to the direction of stride
send prefetch address to prefetch queue

Figure 3.10: Control flow of the combined IAP scheme

3.4 Summary

In this chapter, a general design for hardware controlled prefetching scheme is proposed. By using information embedded in the instruction opcodes, our design will be able to single out the data references with constant strides from the pool of all data references and also able to find out the corresponding stride values. With this valuable information, accurate prefetching can be accomplished and consequently, the CPU stall time due to data cache misses can be reduced. Based on the time when a prefetch request is issued and the present of a default prefetching action, three variations of IAP prefetching schemes — basic, enhanced and combined are proposed. In the basic IAP scheme, prefetch requests are generated for data with one iteration look ahead. Next, the concept of cache block prefetching is introduced in the enhanced IAP scheme to tackle the problem of limited memory bus bandwidth. However, in the basic and enhanced IAP scheme, prefetch requests are issued only when *LOAD/STORE-UPDATE* instructions are encountered. In order to exploit the spatial locality left by the former two schemes, the combined IAP scheme is equipped with default prefetching to issue prefetch requests for the *non-LOAD/STORE-UPDATE* instructions.

Chapter 4

Performance Evaluation

In this chapter, the three proposed IAP schemes are evaluated using trace-driven simulation based on the architecture of IBM RS/6000 machine. In section 4.1, the evaluation methodology, including a brief introduction of trace-driven simulation, caching models, benchmarks and metrics used, will be presented. Next, in section 4.2, the general results of the IAP schemes, with comparison to a traditional hardware prefetching scheme — prefetch-on-miss, will be evaluated using "cycle per instruction due to memory (data cache) misses" (MCPI) as the main metric. To have a more intensive study of how and why the IAP schemes work, some other implicit performance parameters will be investigated in section 4.3. Finally, results from experiments with the zero time prefetching assumption will be shown to investigate the performance impacts of the bus bandwidth on the IAP schemes.

4.1 Evaluation methodology

4.1.1 Trace-driven simulation

In order to have a deeper understanding of the IAP schemes and to show their potentials, detailed trace-driven cache simulation study was performed using SPEC92 (see Section 4.1.3) as our benchmark suite. With the help of *xtrace* facilities, each of the benchmark programs in the SPEC92 suite was traced on the IBM RS/6000 workstation and 100 million instructions per benchmark were collected. The process of trace-driven simulation is summarized in figure 4.1.

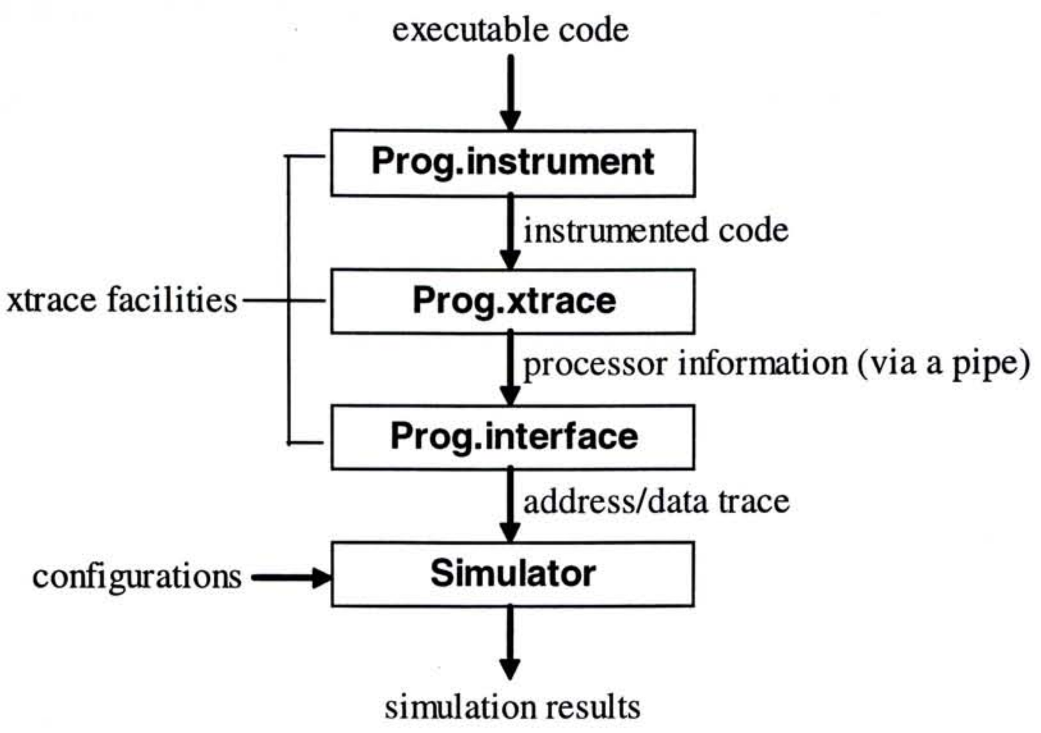


Figure 4.1: Trace-driven simulator using *xtrace*

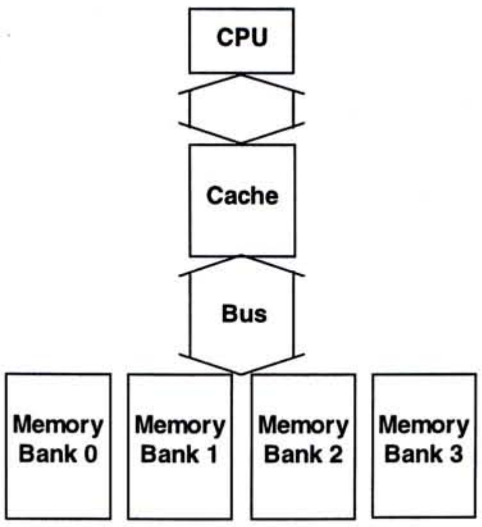
The SPEC92 benchmarks were compiled on the RS/6000 workstation. The executable codes were then instrumented by the program *instrument,* which inserted extra codes into the executable codes in order to extract the processor information during the program execution. The instrumented codes were then handled by *xtrace* and were executed on the RS/6000 machine. The processor information was then passed to the program *interface* via an explicit pipe. The program *interface* could be defined by the users to produce the desired trace format. In the simulation, the following information (a trace record) was recorded for each instruction that was traced:

Inst_address, Inst_content, <data_ref_address if any>, <No._of_bytes_ref if any>

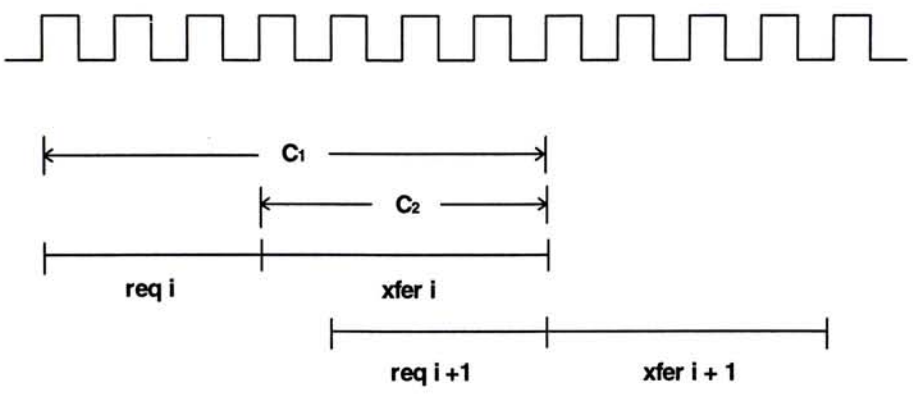
The simulator was configurable in the sense that it could read in the configuration descriptions (for examples, cache size, set-associativity, block size and memory latency time) and create simulation objects (the items model the CPU and the memory system) based on these parameters. As the tracing processes were so time consuming, the trace data were stored in hard disks and the data could be shared by a number of simulators which ran in parallel on different machines to speed up the simulations. The simulator read in the trace records one by one. Then the content of each instruction was decoded and the opcode, the addressing mode, and the registers used in the address calculation were also found.

4.1.2 Caching models

An elementary architectural model, which consists of a processor with perfect pipelined and a 4-way associative data cache with a block size of 32 bytes and a total size of 16K bytes, is defined for the simulations and the replacement algorithm is assumed to be LRU (Least Recently Used). For comparison, each dimension of the cache (cache size, block size and set associativity) is varied respectively for different simulations (cache sizes range from 8 Kbytes to 32 Kbytes, block sizes from 16 bytes to 64 bytes, and set associativities simulated from 1 to 4) while the other two are kept constant.



(a) Interleaved memory



(b) Timing of data access

Figure 4.2: Memory model of the simulator

The memory model of the second level cache in the simulations is assumed to be interleaved and its design and timing characteristics are shown in figure 4.2. The memory is organized into a number of banks (or modules) to handle multiple words at one time rather than a single word. Each bank is one word wide which is the same as the first level cache and the bus. A cache block usually consists of a number of words (for example, a 32-byte block consists of 8 4-byte words). Whenever a cache miss occurs in the first level cache and a fetching request is sent to the second level cache, the banks will work simultaneously — bank 0 will start reading for the first word in the block, bank 1, the second word, bank 2, the third word,... etc. However, since there is only one memory bus between the first and second level cache, the transfers of the words must be processed sequentially. As a result, the time for a demand fetch request to transfer a cache block between the first level cache and the second level cache memory can generally be summarized by the equation $C_1 + C_2 * (block\_size - 1)$, where C_1 is the delay time for the first word to arrive after a cache miss (that is, *startup_overhead* + *transfer_time_for_a_word*) and C_2 is a parameter that indicates the bus bandwidth between the first level cache and the second level cache (that is, transfer time for a word). In our experiments, C_1 was assumed to be 6 and C_2 to be 1.

For a given cache block size, the time for a demand fetch request (due to the first level cache miss) to finish is assumed to be equal to the time for a prefetch request (to the second level cache) to finish. The prefetch requests are resided in the prefetch queue and the request R_P at the beginning of the queue will be sent to the second level cache if the queue is not empty and the bus to the second level cache is free. If the address of the pending prefetch request immediately follows that of the current request R_c (demand fetch or prefetch) being processed in the second level cache (i.e. $address\_of\_R_p = address\_of\_R_c + 1$), the interleaved memory, that is, the second level cache, does not need to wait until the request R_c is completely finished. It can continue to process R_p when some memory banks are free, although the memory bus may be still transferring the data of R_c. In this case, the startup overhead of R_p can be hidden and the time for completing the prefetch request will be equal to $C_2 * block\_size$.

The second level cache can only handle one request at a time, no matter it is a demand fetch or prefetch request. When a demand fetch miss occurs in the first level cache, it will try to fetch the data from the second level cache. However, it may be in a situation that the second cache is serving a prefetch request. In case of such conflict, the priority will be given to the demand fetch miss and the prefetch request will be aborted and the demand fetch request will be started next cycle.

For simplicity, the second level cache(memory) is assumed to be infinitely large. That is, there is no cache miss in the second level cache.

Each instruction is assumed to be executed in one cycle and no superscalar architecture is simulated and cache access upon a cache hit is assumed to be one cycle. Totally, five cache prefetch models were simulated:

[1] Data cache without any prefetching;

[2] Data cache with "prefetch-on-miss";

[3] Data cache with the basic IAP scheme — prefetch requests are issued only for data referenced by *LOAD/STORE-UPDATE* instructions. Multiple-block prefetches will be used for multiple-block data;

[4] Data cache with the enhanced IAP scheme — using the basic IAP scheme to issue prefetch requests. If the prefetch address resides in the same memory block as the address of the current datum, the next block (according to the direction of stride) will be prefetched;

[5] Data cache with the combined IAP scheme — using the enhanced IAP scheme for *LOAD/STORE-UPDATE* instructions and the default "prefetch-on-miss" for non-*LOAD/STORE-UPDATE* instructions.

4.1.3 Benchmarks and metrics

Eight applications from the SPEC benchmarks are compiled on the RS/6000 with optimization options enabled. Brief descriptions of them are presented here:

- Espresso: An integer benchmark performs set operations such as union, intersect and difference. Espresso minimizes Boolean functions. It takes as input a Boolean function and produces a logically equivalent function possibly with fewer terms. Both the input and output functions are represented as truth tables.
- Spice2g6: A floating point benchmark, spice2g6 is a circuit simulation program for nonlinear dc, nonlinear transient, and linear ac analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, transmission lines, and semiconductor devices: diodes, BJTs, JFETs, and MOSFETS.
- Li: Li is a CPU intensive integer benchmark written in C. The benchmark performs minimal I/O. Li is a Lisp interpreter itself. The time li takes to solve the 9-queens problem is measured in the program. The input file to the interpreter contains Lisp code that defines the 9-queens problem.
- Compress: Compress reduces the size of the files using adaptive Lempel-Ziv coding. The amount of compression obtained depends upon the size of the input, the number of bits per character, and the distribution of common substrings. Typically, text such as program source code or English is reduced by 50-60%.
- Wave5: A large, single precision floating point FORTRAN benchmark and a two-dimensional, relativistic, electromagnetic particle-in-cell simulation code used to study various plasma phenomena. Wave5 solves Maxwell's equations and particle equations of motion on a Cartesian mesh with a variety of field and particle boundary conditions.

- Tomcatv: Tomcatv is a highly vectorizable double precision floating point FORTRAN benchmark. In this program, two-dimensional, boundary-fitted coordinate systems around general geometric domains are generated and studied.
- Su2cor: The program is a vectorizable FORTRAN program with double precision floating-point arithmetic. In this application program from the area of quantum physics, masses of elementary particles are computed in the framework of the Quark-Gluon theory. The data are computed with a monte carlo method taken over from statistical mechanics.
- Nasa7: Nasa7 is a collection of 7 heavily floating point intensive kernels. For each kernel, the program generates its own input data, performs the kernel and compares the result against an expected result The seven kernels executes operations used frequently in NASA applications, such as Fourier transforms and matrix manipulations.

Benchmark	Percentage of Instruction Executed					
	Total *LOAD*	*LOAD-UPDATE*	Total *STORE*	*STORE-UPDATE*	Total *LOAD/STORE*	Total *LOAD/STORE-UPDATE*
espresso	22.0	11.1	3.9	1.3	25.9	12.4
li	25.4	0.8	15.2	2.3	40.6	3.1
compress	21.5	0.0	9.2	0.3	30.7	0.3
spice2g6	18.3	1.4	9.9	1.1	28.2	2.5
wave5	26.5	1.4	9.7	2.2	36.2	3.6
tomcatv	29.6	18.3	11.1	10.1	40.7	28.4
su2cor	26.4	8.5	14.1	6.1	40.5	14.6
nasa7	42.8	42.0	1.7	1.4	44.5	43.4

Figure 4.3: Percentages of *LOAD/STORE-UPDATEs* in SPEC92 Benchmark Suite

Figure 4.3 shows the percentages of the *LOAD/STORE-UPDATE* instructions found in the instruction mixes of the SPEC92 benchmarks programs. The percentages in the figure clearly confirm our statement that the current compiler technology is able to fully utilize the *LOAD/STORE-UPDATE* compound opcodes to produce highly efficient code. The proportion of *LOAD/STORE* instructions that also belongs to *LOAD/STORE-*

UPDATE instructions ranges from a few percent to over 95 percent. For example, in the nasa7 benchmark, about 97.5% of the *LOAD/STORE* instructions executed are *LOAD/STORE-UPDATE* instructions.

Experiments with the elementary caching model, with varying caching parameters, are performed. Two main metrics are used here to evaluate the performances of different caching schemes. The first one is "cycle per instruction due to memory (data cache) misses" (MCPI). As its name suggests, this performance parameter measures the average additional processor stall time due to the first level cache misses. It also helps to show the degree of degradation of CPU performance due to the data cache misses in terms of memory cycle stalls per instruction. Generally, it can be calculated with the execution CPI and baseline CPI by the equations:

$$MCPI = CPI_{execution} - CPI_{baseline}$$

where $CPI_{execution} = \dfrac{total\_number\_of\_cycles\_executed}{total\_number\_of\_instructions}$

and $CPI_{baseline} = \dfrac{total\_number\_of\_cycles\_executed\_for\_no\_cache\_miss}{total\_number\_of\_instructions}$.

We feel that this is a better measurement parameter than the cache hit (or miss) ratio because the penalty of a cache miss depends on the cache block size. More important, with limited bus bandwidth between the first level cache and the second level cache(memory), there were always situations when a cache block i is being prefetched, the cache block i is actually referenced. This is what we called the **partial cache hit (or miss)** situation. The data prefetching will be allowed to finish and then the requested data are sent to the CPU. Under this situation, the penalty of this kind of partial cache hits is not a constant — it ranges from 1 to (*maximum cache miss penalty* - 1) (i.e. C_1 + C_2 * *block_size* - 1) - 1). This makes the cache hit ratio even more difficult to reflect the actual cache performance because some part of the data fetching time is overlapped with the processor execution while the other part of the data fetching time is visible to the processor.

Since we assume that the processor can execute each instruction in one cycle and there is an ideal instruction cache in the system, one may intuitively deduce that the baseline CPI is always equal to one. However, this assertion may be wrong due to the fact that the memory bus between the processor and the first level cache is only 32-bit (4-byte) wide. It also means that at most 4 bytes can be transferred between the processor and the first level cache in one cycle. If the data needed to be loaded or stored by an instruction is longer that 4 bytes, the instruction can only be finished after all required data are loaded in the processor and the execution time is sure to be longer than one cycle even when there is no cache miss. For examples, a *LOAD/STORE-DOUBLEWORD* instruction will be executed for two cycles even when the data needed is found in the cache. As a result, the baseline CPI will probably be greater than one and this effect is more significant in the double precision floating point benchmarks such as tomcatv and nasa7, in which most of the data are doublewords of eight bytes long. Figure 4.4 shows the baseline CPIs found for the eight benchmarks programs used in our simulations.

Benchmark	baseline CPI
espresso	1.011
li	1.075
compress	1.032
spice2g6	1.084
wave5	1.124
tomcatv	1.407
su2cor	1.286
nasa7	1.444

Figure 4.4: Baseline CPIs of SPEC92 Benchmark Suite

The other metric used is "percentage memory stall time reduction over no prefetch", which is defined as:

$$\%\_stall\_reduction = \frac{memory\_stall_{no\_prefetch\_cache} - memory\_stall_{prefetch\_cache}}{memory\_stall_{no\_prefetch\_cache}} \times 100 .$$

The metric can be used to show the extent that memory stall time due to data cache miss is reduced with respect to an elementary cache using no prefetch scheme.

4.2 General Results

In this section, the experimental results are presented to show the benefits of the prefetching schemes. The architecture with the elementary caching model using no prefetch scheme is compared with the same architecture augmented by each of our three prefetching schemes. As a reference, the architecture with prefetch-on-miss scheme is also included. The results for caching models with varying cache size, block size and associativity will be presented one by one. However, we would like to summarize some common observations here since they can be generally found from experiments for all the varying models.

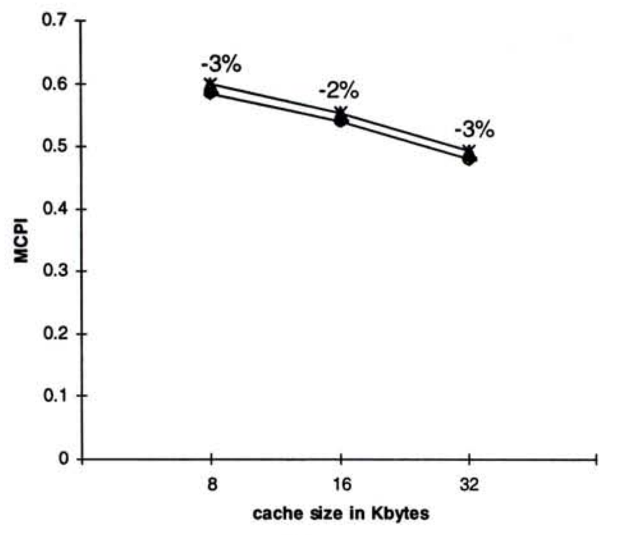
- All the prefetching schemes seem to have no effect on the benchmark compress — as we can find out in Figure 4.3, only 0.3 percent of the total instructions (less than 1 percentage of *LOAD/STORE* instructions) is of the type *LOAD/STORE-UPDATE*. As described in the previous sections, the prefetching actions of our IAP schemes are triggered by the *LOAD/STORE-UPDATE* instructions. With this inconsiderable amount of *LOAD/STORE-UPDATE* instructions in the compress program, only a few prefetch requests will be generated for the basic and enhanced IAP schemes and their effects will be negligible, as we can find that the curves corresponding to these schemes overlap with that of the model with no prefetch. Moreover, the combined IAP scheme will revert back to a simple prefetch-on-miss scheme and the two schemes suffer a slight performance degradation with respect to the no prefetch cache, which is probably due to the lack of constant stride references in the program (as reflected by the lack of *LOAD/STORE-UPDATE* instructions).

- Prefetch-on-miss, the traditional hardware prefetching scheme, generally has some improvements over most of caching models tested except for the benchmark compress. The basic IAP scheme shows performance improvement over all caching models for almost all benchmarks used (except compress). The enhanced IAP scheme has similar performance as the basic IAP scheme does when the cache block size is small (the explanation will be given in Section 4.2.2) but it outperforms the basic IAP scheme over all other models tested. Although these two schemes show no improvement over the compress program, they also cause no degradation in the same time. It is because only highly accurate prefetches can be issued under these schemes.
- The effects of the combined IAP scheme, which adds a default prefetch-on-miss scheme on top of the enhanced IAP scheme to handle data referenced by *non-LOAD/STORE-UPDATE* instructions, can be classified into two main streams. For some of the benchmarks such as nasa7, tomcatv and su2cor, the default prefetching scheme seems to have no impact to the cache performance. The curve for the enhanced IAP scheme and the curve for the combined IAP scheme almost overlap with each other. However, for espresso, li, spice2g6 and wave5, adding the default prefetch-on-miss scheme to the enhanced IAP scheme helps to reduce the memory stall time further. This can be explained as follows. In the nasa7, su2cor and tomcatv programs, most of data references with strong spatial locality are referenced by

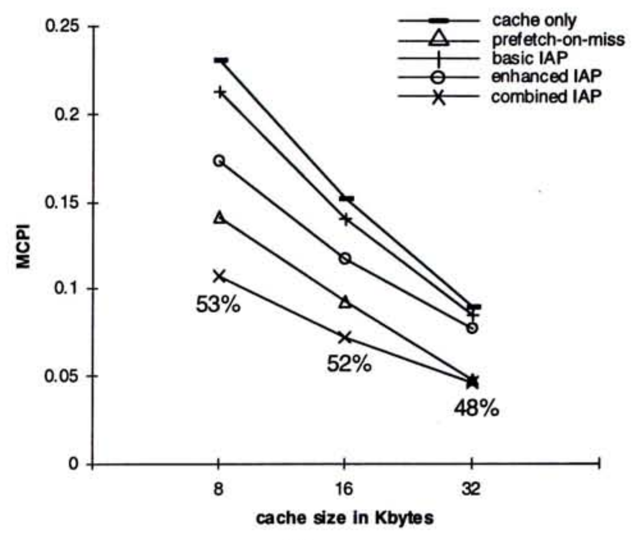
LOAD/STORE-UPDATE instructions and they can be prefetched very accurately by the enhanced IAP scheme. Thus, the addition of the default prefetch-on-miss scheme to the enhanced IAP scheme cannot provide additional improvement in cache performance. However, for espresso, li, spice2g6 and wave5, a significant portion of the data references with strong spatial locality are referenced by *non-LOAD/STORE-UPDATE* instructions. They cannot be prefetched by the enhanced IAP scheme. On the other hand, these *non-LOAD/STORE-UPDATE* references can be prefetched quite accurately by simple cache prefetching schemes such as the prefetch-on-miss scheme. This also shows the flexibility of the IAP schemes. The IAP schemes can be implemented together with other prefetching algorithm to achieve better cache performance.

- The memory stall time reduction that can be achieved by the enhanced IAP scheme or by the combined IAP scheme ranges from about a few percentages to over 90%, with an average of about 50%. These figures really show the potentials of the IAP schemes. This kind of improvement in cache performance over a wide range of benchmark programs (instead of some small routines or kernels such as Livermore Kernels) is really substantial. Furthermore, this performance improvement can be obtained by just modifying the on-chip cache hardware and no change to the processor architecture (such as the instruction set) is required. This makes the IAP schemes even more attractive.

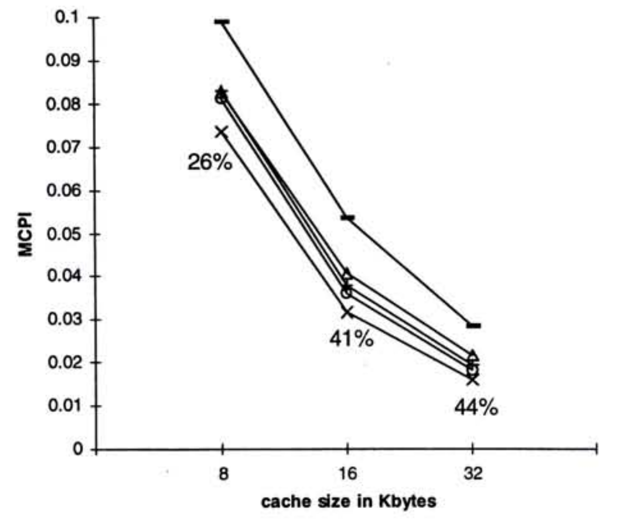
4.2.1 Varying cache size



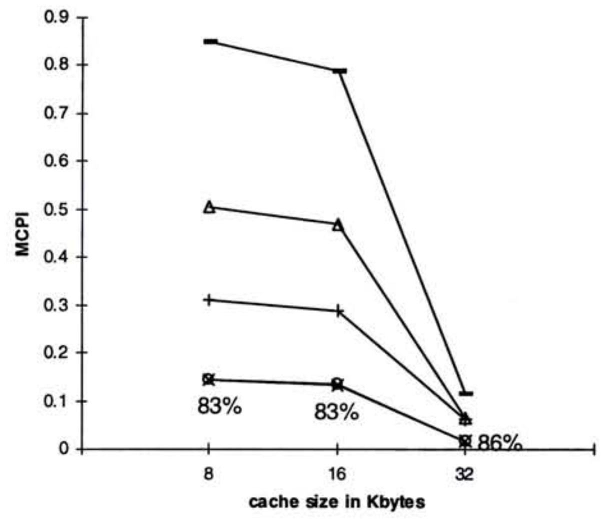
(a) compress



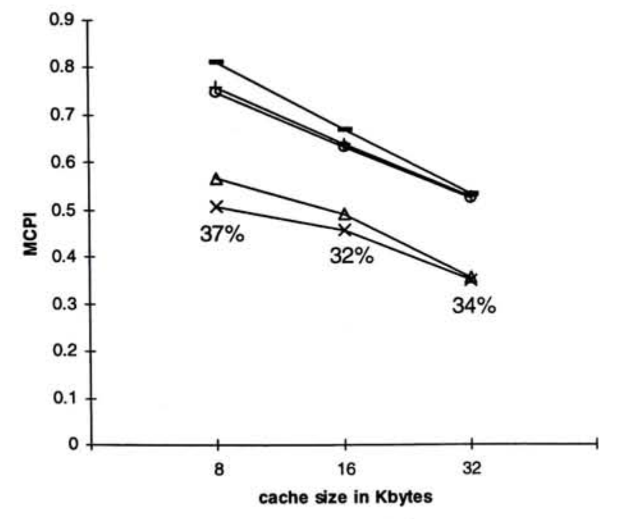
(b) expresso



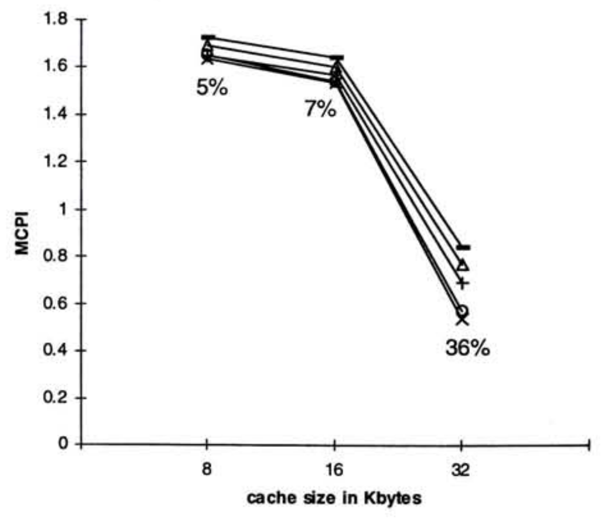
(c) li



(d) nasa7

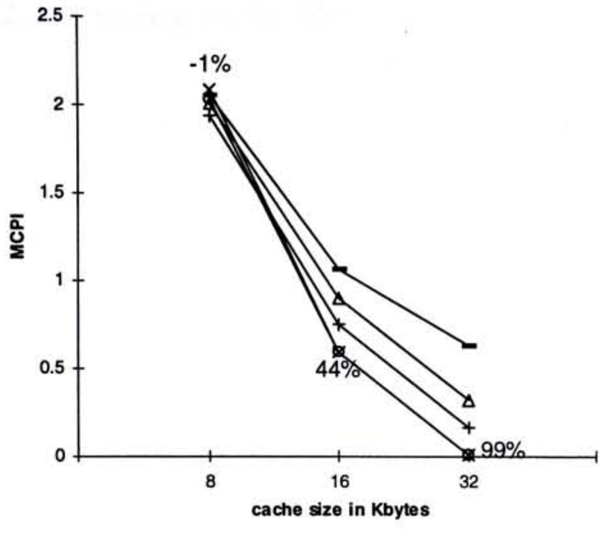


(e) spice2g6

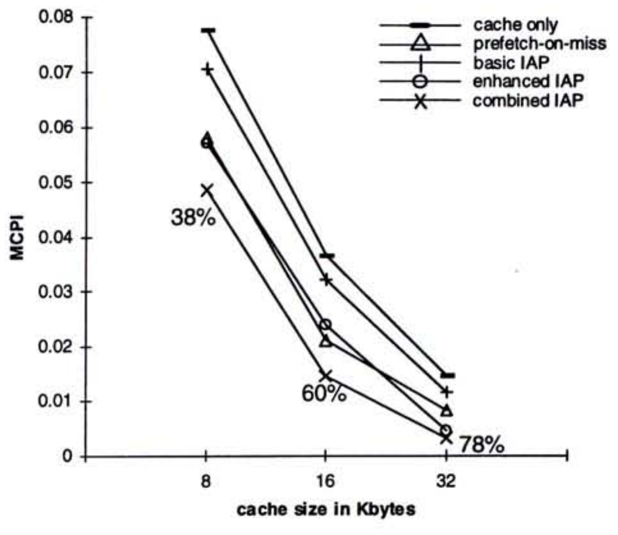


(f) su2cor

Figure 4.5: MCPI and stall reduction of simulations with varying cache size



(g) tomcatv



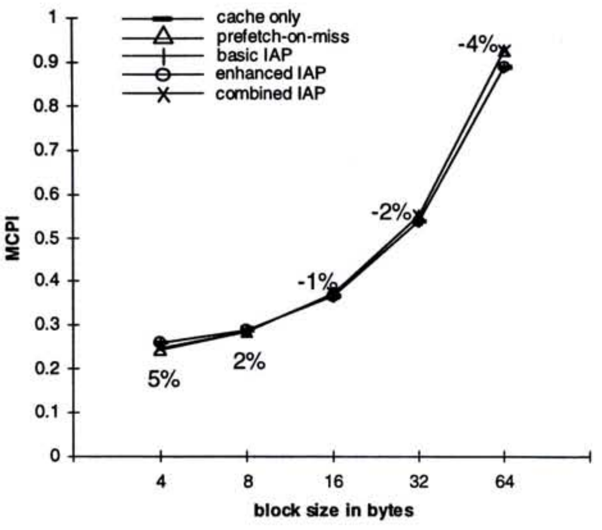
(h) wave5

Figure 4.5 (cont.): MCPI and stall reduction of simulations with varying cache size

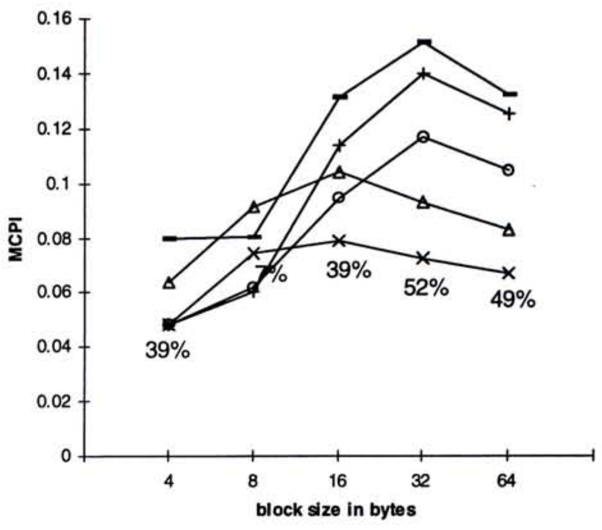
Figure 4.5 shows the simulation results for the eight benchmark programs using cache size varies from 8K to 32 K bytes. All experiments are done with caching models of 32-byte block size and 4-way associativity. The numbers attached show the "percentage stall reduction over no prefetch" of the combined IAP schemes.

As expected, one can find that the MCPI decreases as the cache size increases. However, for some benchmarks, such as su2cor and tomcatv, the IAP schemes show little improvement when the cache size is small, that is, 8K bytes, but exhibit substantial improvement when the cache is increased to 16K and 32K bytes. For tomcatv (shown in Figure 4.5(g)), when the cache size is very small (8K bytes), the combined IAP scheme actually degrades the performance instead of improving it. This is probably due to the small cache size and the aggressive cache prefetching scheme. Even though the prefetching can be very accurate, those accurately prefetched data will replace each other away from the data cache before they have the chance to be used. However, as the cache size increases from 8 Kbytes to 16 Kbytes, this cache conflict problem is minimized and the three IAP schemes start to have substantial cache performance improvement.

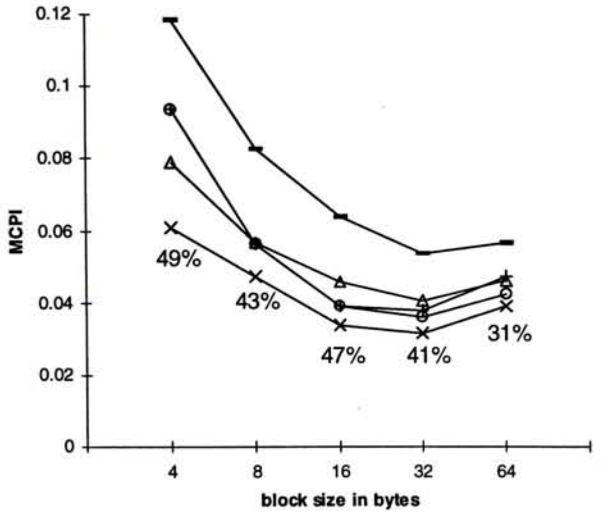
4.2.2 Varying cache block size



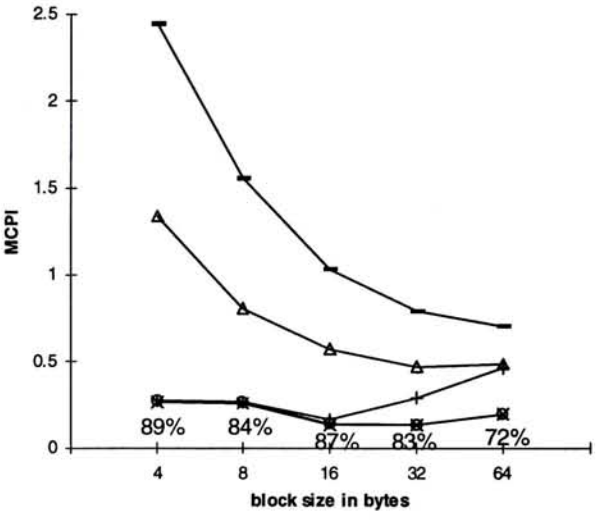
(a) compress



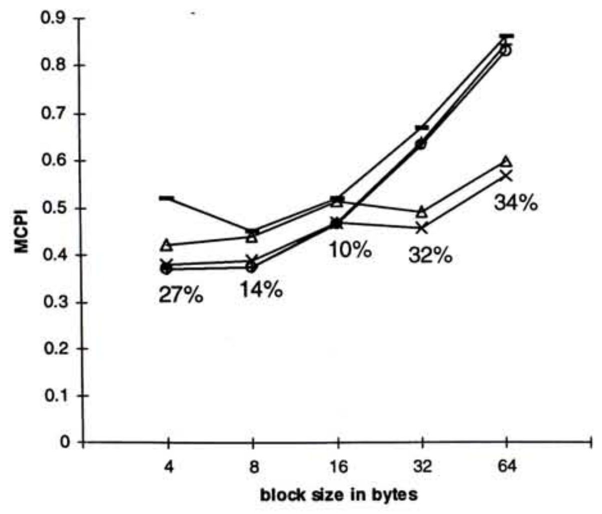
(b) expresso



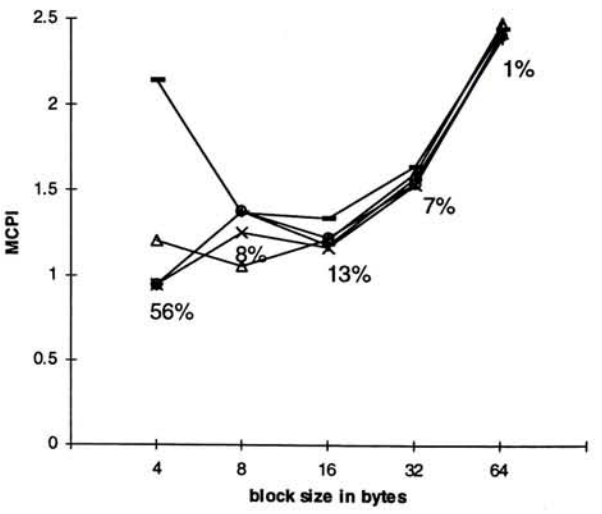
(c) li



(d) nasa7

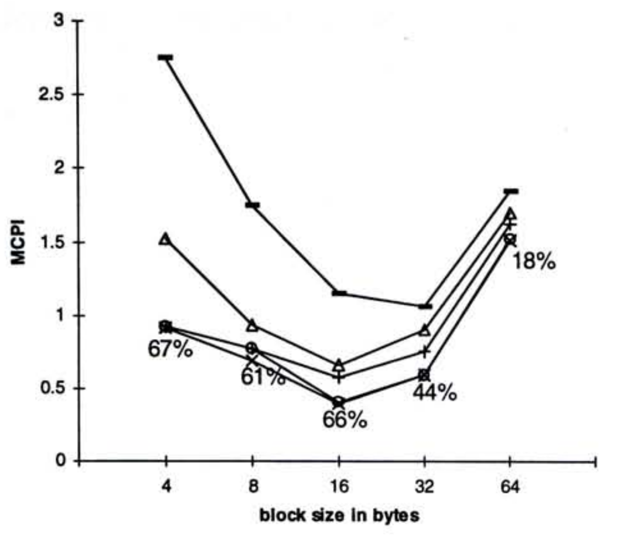


(e) spice2g6

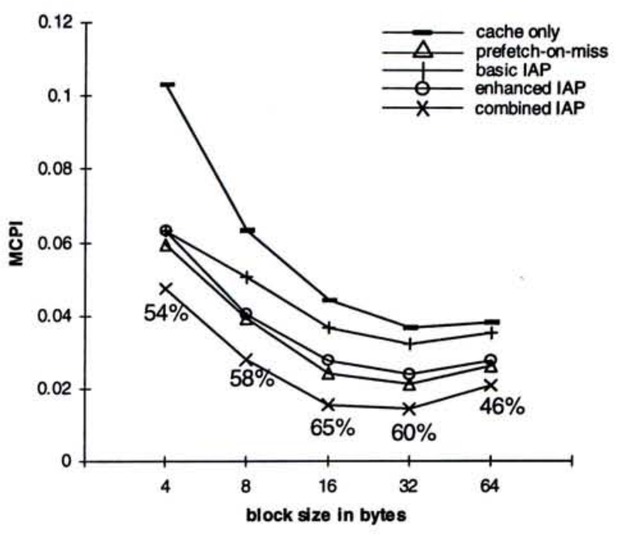


(f) su2cor

Figure 4.6: MCPI and stall reduction of simulations with varying block size



(g) tomcatv



(h) wave5

Figure 4.6 (cont.): MCPI and stall reduction of simulations with varying block size

Figure 4.6 shows the simulation results for the eight benchmark programs using different prefetching schemes. The experiments are done with caching models of 16K-byte cache size, 4-way associativity and varying cache block size from 4 to 32 bytes.

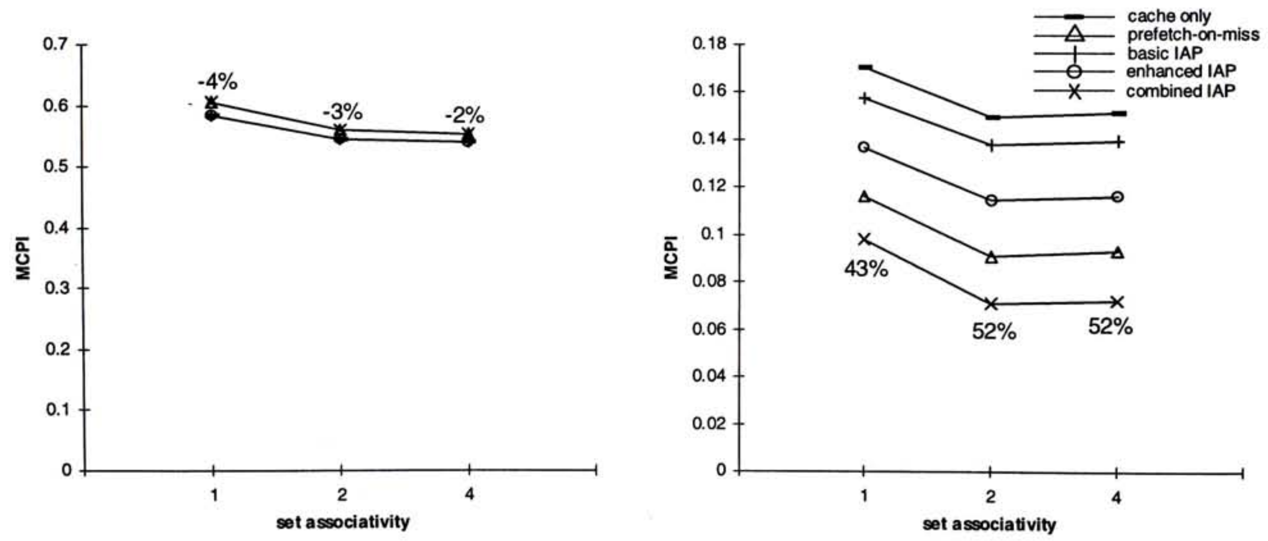
The MCPI curves for IAP schemes generally have U-like shapes. That is, the MCPIs of the programs first decline from small block size to the optimal block size. Then, the directions of the curve reverses and the MCPIs keep rising after the optimal block sizes. These observations are common to be found in most of the cache simulations. As the block size increases, more data will be fetched one time and the spatial locality between these data may be beneficial to processor execution. Moreover, it is also more economical on average to fetch a larger block one time than to fetch a smaller block several times separately because the time to fetch a block from memory is equal to C_1 + C_2 * (*block_size* - 1). As the block size increases from the smallest size to the optimal one, these effects are dominant and the MCPI continues to drop in this range. However, as the block size keeps increasing after that point, using larger cache block size for sequential prefetching seems to be not so effective. As the block size further increases, greater portions in the blocks will contain data that will not be referenced in the near further and the blocks will be kicked out without these data being touched. Moreover, increasing the cache block size elongates the time for fetching a block from the memory. This means the CPU must wait longer for the same amount of data needed (for example, the CPU is stalled longer for a 4-byte datum in a 64-byte block than a 4-byte datum in a 32-byte block). At the same time, this also increases the risks of killing the prefetches by demand fetches caused by real cache misses. Finally, the larger block size reduces the total number of distinct blocks that can be put into the data cache and increases the conflicts between blocks in the cache which may cause some useful data to be kicked out before it is referenced by the CPU. When these adverse effects of larger block size outweigh the benefits brought, increasing block size will mean higher miss rate, more processor idle time and lower CPU performance. These explain why the MCPI curves rise after passing the optimal block sizes.

For some programs (compress, espresso, spice2g6 and su2cor), the MCPI curves show that the caches work better when the block size is small (4 bytes). It is probably because the data of consecutive references are separated far apart and do not reside in the same block. As a result, only small portions of the large blocks fetched from the secondary memory will be referenced in the near future and the locality introduced by the large block size does not help much. On the other hand, with the smaller block size, the cache with the same size can contain more blocks and it gives more flexibility for the IAP schemes to do accurate prefetching. As conclusion, smaller cache block size is preferred in these situations. This also agrees with what people found [Lee87] about smaller block sizes for data cache.

However, for espresso (Figure 4.6(b)), the increasing MCPI curves of the IAP schemes turn around and begin to drop when the block size increases from 32 bytes to

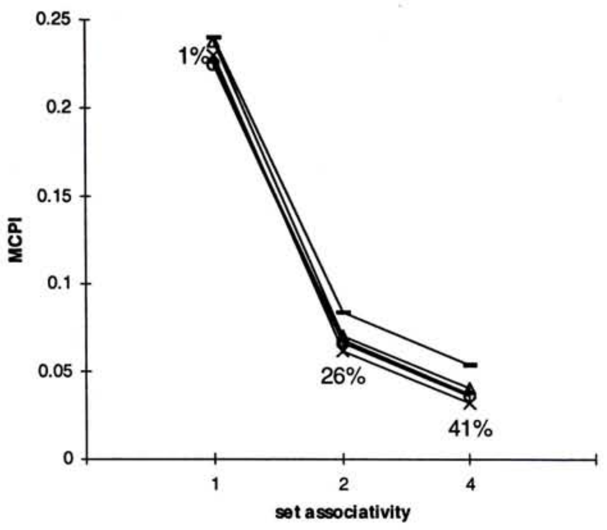
64 bytes (similar phenomena are also observed for the cache only and prefetch-on-miss curves when the block size increases from 16 bytes to 32 bytes). Although the explanation for this phenomenon is not very clear, we suspect that this is related to the data accesses with large stride values of 32 to 64 bytes in the program. From 4 bytes to 32 bytes cache block size, the number of blocks that can be stored in the cache is reduced by half each time when the block size is doubled. However, if the stride size of the data accesses is large, a small increase in the block size does not capture more useful data. Consequently, increasing the cache block size below 32 bytes block size only causes cache pollution and results in poor cache performance. When the cache block size is increased from 32 bytes to 64 bytes, sequential data prefetching using large block size starts to have some effect and the cache performance is improved.

4.2.3 Varying associativity

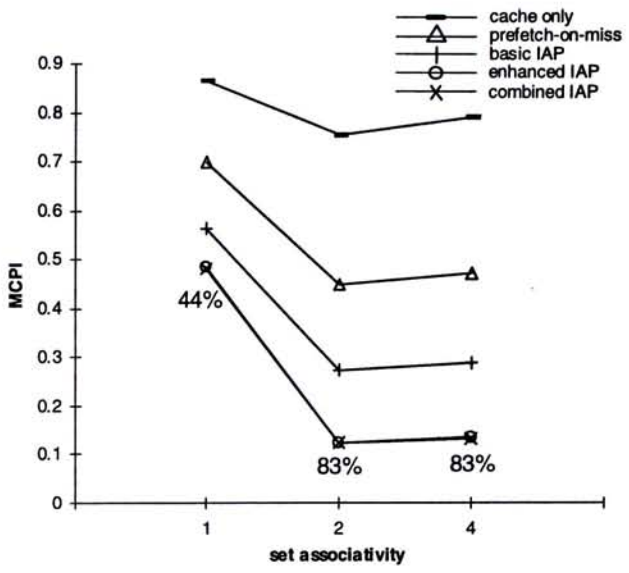


(a) compress (b) expresso

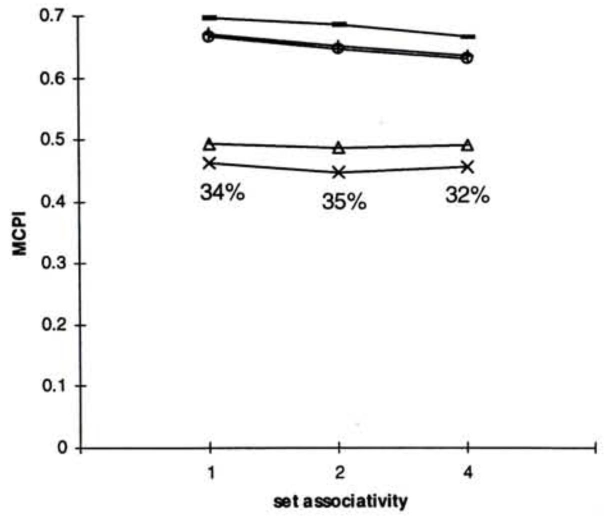
Figure 4.7: MCPI and stall reduction of simulations with varying associativity



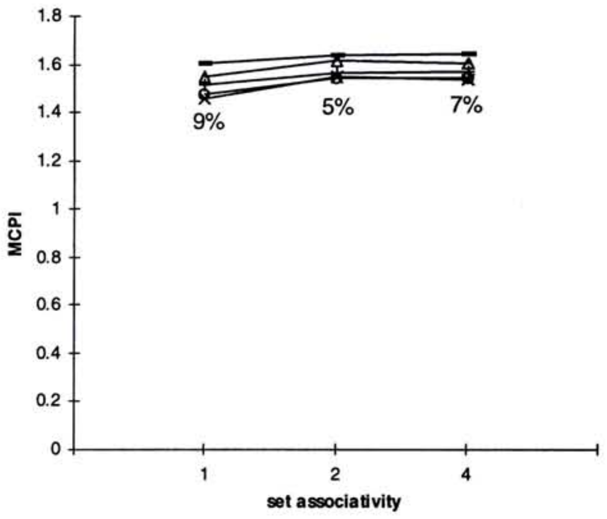
(c) li



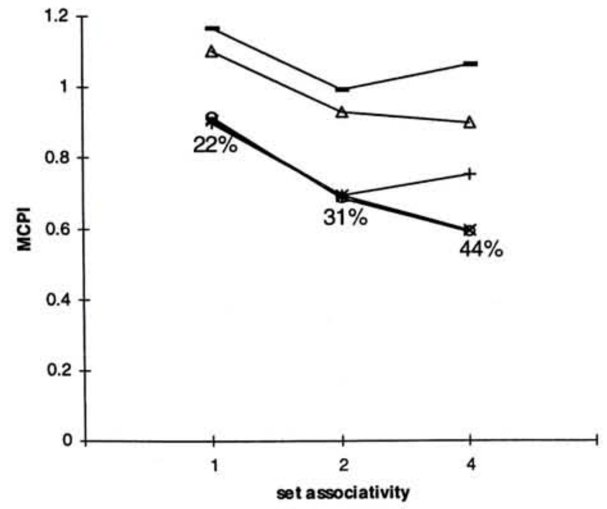
(d) nasa7



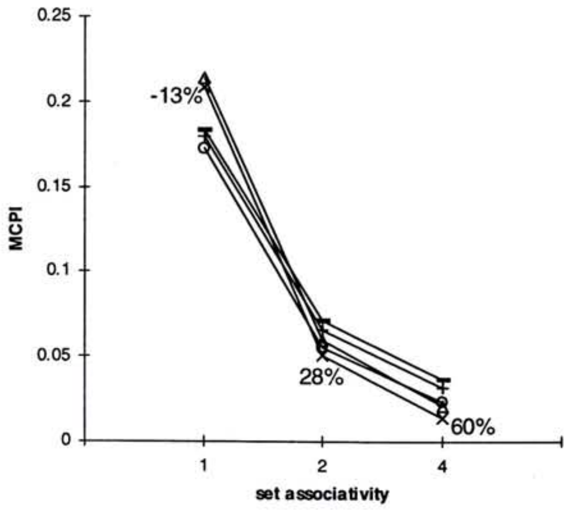
(e) spice2g6



(f) su2cor



(g) tomcatv



(h) wave5

Figure 4.7 (cont.): MCPI and stall reduction of simulations with varying associativity

Figure 4.7 shows the effect of increasing the cache set associativity. As it is expected, from the set associativity of 1 to 2, the performance is generally improved (except the benchmark su2cor). With a one-way associativity (direct mapping) cache, every block can be placed at only one position. If it happens that two sequences of data accesses, for example, two arrays inside the same loop, are mapped to similar sets, they will continually displace each other's data block in the cache, although the displaced block may contain data that will be referenced in the near future. As a consequence, miss rate will be increased and the cache performance will be degraded. It accounts for the large improvement from one-way to two-way set associative cache. This effect is more obvious for the benchmark nasa7 (Figure 4.7(d)). From Figure 4.3, we can find that almost all (over 90%) of the data references belong to the *LOAD/STORE-UPDATE* (constant stride) type and are mainly chains of array or pointer references. With this large amount of constant stride references, the chance of conflicts induced by the address mapping will probably be very high. However, the cache performance is more or less the same for 2-way and 4-way set associativity. Although increasing the associativity gives more flexibility for cache block placement, at the same time, the number of sets in the cache will be halved and the performance will be less dependent on the associativity in these situations.

One exception needed to be noted is that — for the benchmark su2cor, the performance of a one-way associative cache is superior to the other two configurations. The results are not surprising in the sense that the MCPIs are quite high for the program (about 1.5 cycle). The 16 Kbytes cache is unable to hold the full working set and the miss rates are high for the program. Under these situations, the conflict between the cache block is so serious that a rigid mapping scheme may help separating the cache blocks and reduce the conflict to give a better performance. This is similar to what Smith and Goodman suggested [SmG83] that a small cache using direct mapping could consistently outperform one using fully associative with least recently used replacement scheme.

4.3 Other performance metrics

In the last section, we showed the performance of our IAP schemes with some "explicit" metrics, which can always give the users of the system a direct feeling of how the memory system performs. For example, the users can feel that the system is faster when it execute programs with smaller MCPIs or bigger stall time improvement over the cache only model. Now, we would like to have a more intensive study to see why and how the IAP schemes work and to measure the performance with some "implicit" parameters. They are somewhat hidden but can give one more confidence that the IAP schemes will general perform well in other applications. For the sake of brevity, only representative results are shown here.

4.3.1 Accuracy of prefetch

Up to now, the IAP schemes are always emphasized with their accuracy. We would like to define the accuracy of prefetching as:

$$Accuracy = \frac{number\_of\_prefetched\_lines\_referenced\_before\_displaced}{total\_number\_of\_lines\_prefetched} \times 100$$

In other words, it is the percentages of prefetched lines that are actually referenced at least once when they are in cache. The results for the program nasa7 and tomcatv are presented in Figure 4.8.

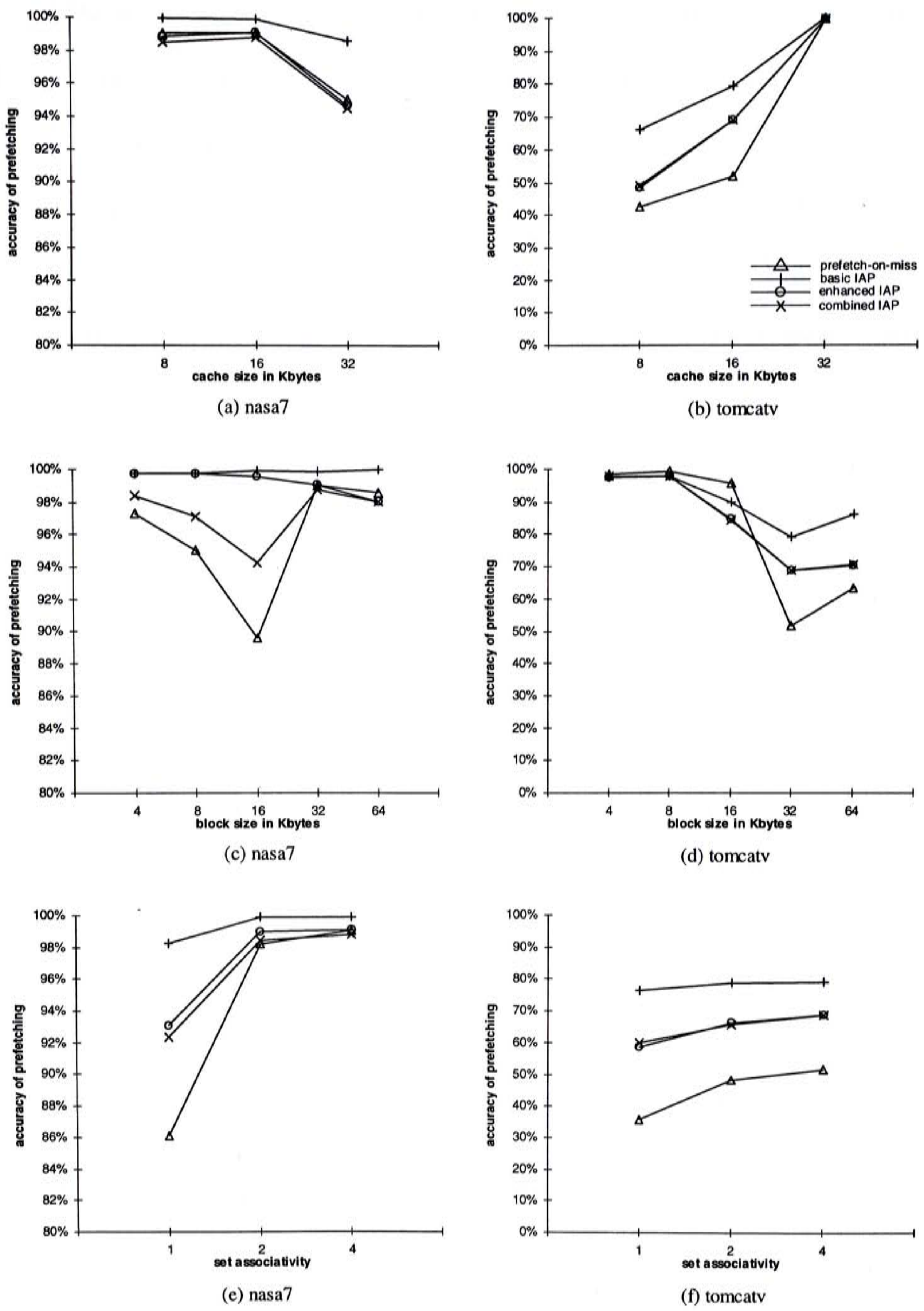


Figure 4.8: Accuracy of prefetching

The results show that the basic IAP scheme has very high prefetching accuracy of more than 80% and it is generally higher than the accuracy of the "prefetch-on-miss" scheme. As it is expected, the basic IAP scheme only prefetches data when the *LOAD/STORE-UPDATE* instructions are encountered and the data to be prefetched for the next iteration are not in the cache. It is almost certain that the prefetched cache block will be referenced in the near future. The deviation of the accuracy of basic IAP scheme from 100% may be probably due to [1] the conflict between the cache blocks which causes the prefetched blocks to be displaced out before they are referenced, and [2] the last elements of arrays (or pointer references) reside at the ends of the cache blocks and inappropriate prefetch requests for other irrelevant data are issued. The enhanced IAP scheme generally has a slightly lower accuracy. With more than one iteration look ahead in the scheme, the cache blocks are brought into the cache earlier and it increases the chance of the blocks being displaced without being referenced. Moreover, the prefetched block will be required to stay longer in the cache before it is referenced and this further increases the conflict of cache blocks in the cache. Furthermore, in the enhanced IAP scheme, the same "last element effect", which causes inaccurate prefetch to be issued, will occur even if the last elements do not reside at the ends of the blocks. The combined IAP scheme, which incorporates the less accurate prefetch-on-miss scheme as a default scheme, has a lower accuracy than the former two IAP schemes as expected.

The only situation where the prefetching accuracy of the basic IAP scheme is below 80% is when the cache size of 8 Kbytes is applied to tomcatv (Figure 4.8(b)). Further analysis shows that this relatively low percentage is not related to the accuracy of prefetching. Instead, it is due to the high conflict between the cache blocks at this small cache size, resulting in the replacement of accurately prefetched blocks from cache before they have the chance to be used. Note that we also found that these replaced blocks are often referenced soon after they are replaced from cache.

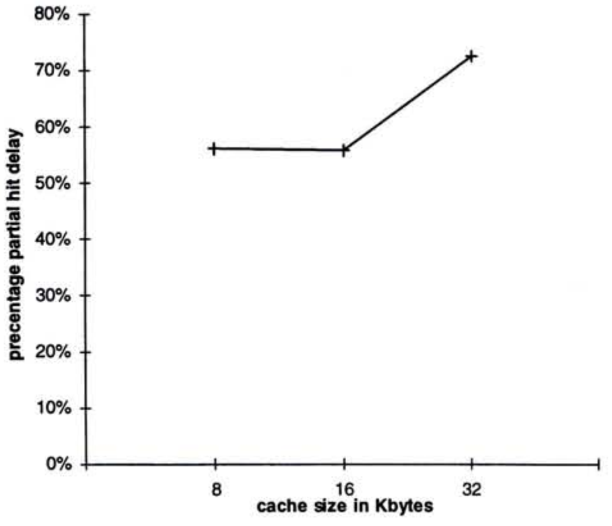
Although the enhanced and combined IAP schemes have accuracy lower than that of the basic IAP scheme, it does not imply that they have worse performances. As illustrated in the following section, the enhanced IAP scheme offers great help in reducing the memory stall time caused by the partial hits. The combined IAP scheme can still perform accurate prefetching for the *LOAD/STORE-UPDATE* instructions, its lower accuracy is due to the default prefetches issued for the *non-LOAD/STORE-UPDATE* instructions. However, these less accurate prefetches are still benefitical to the cache performance, although with a smaller improvement than those brought up by the accurate prefetches issued for the *LOAD/STORE-UPDATE* instructions.

4.3.2 Partial hit delay

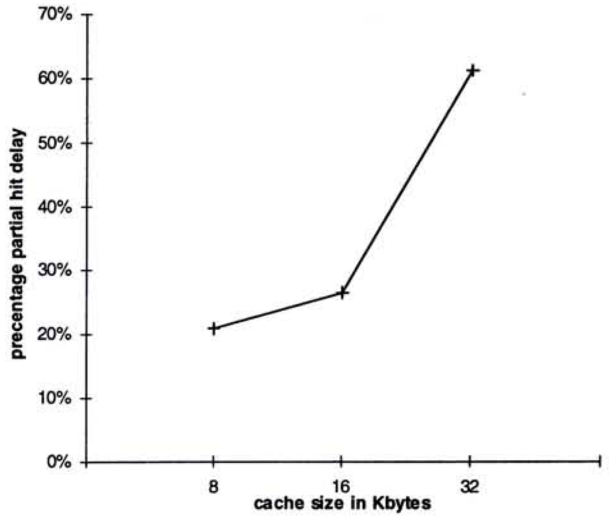
As explained earlier, with limited bus bandwidth between the first and second level cache memory, there are always cases when a cache block is being prefetched, it is actually referenced by the processor. The delay time caused in this situation is called **partial hit (or miss) delay** and the problem will become more severe if [1] the cache block size is large and the time to retrieve a block from memory is longer, and [2] the time between two successive references to two different blocks is short. The basic IAP scheme, although equipped with a highly accurate prefetching mechanism, suffers great performance loss due to the partial hit delay with only one iteration look ahead. In order to study the effect, the "percentage partial hit delay" is defined as:

$$\%\_partial\_hit\_delay = \frac{memory\_delay\_time\_due\_to\_partial\_hit}{overall\_memory\_delay\_time} \times 100$$

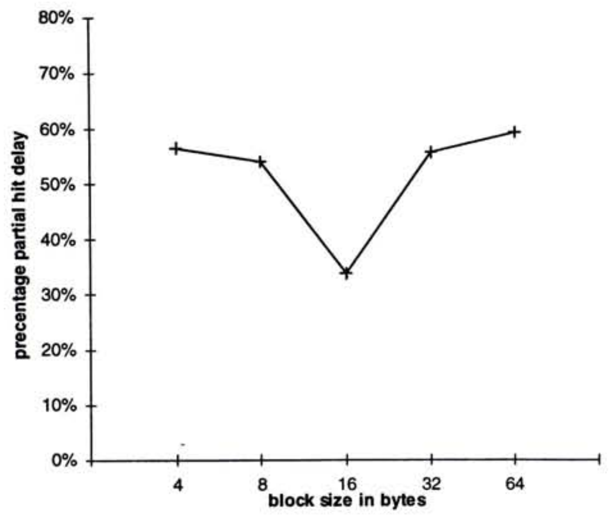
to show the percentage of overall memory stall time due to partial hit. The results for nasa7 and wave5 are shown in the Figure 4.9.



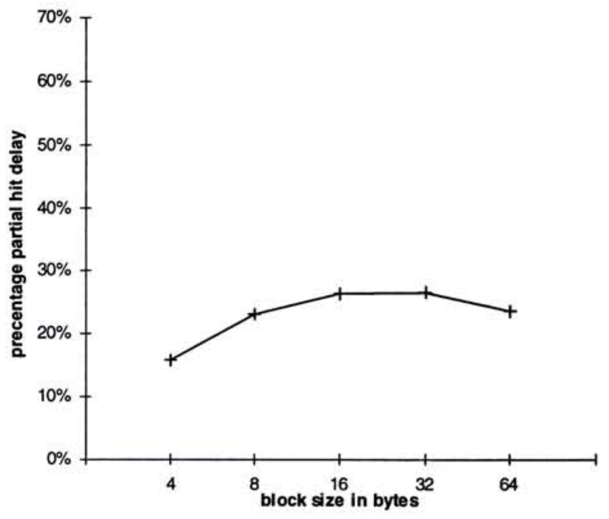
(a) nasa7



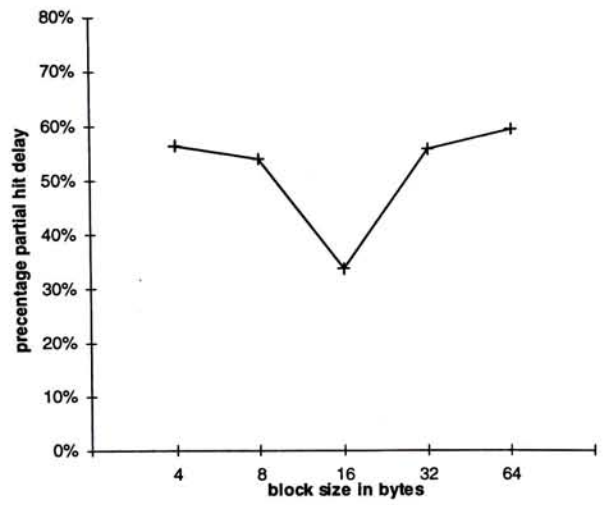
(b) wave5



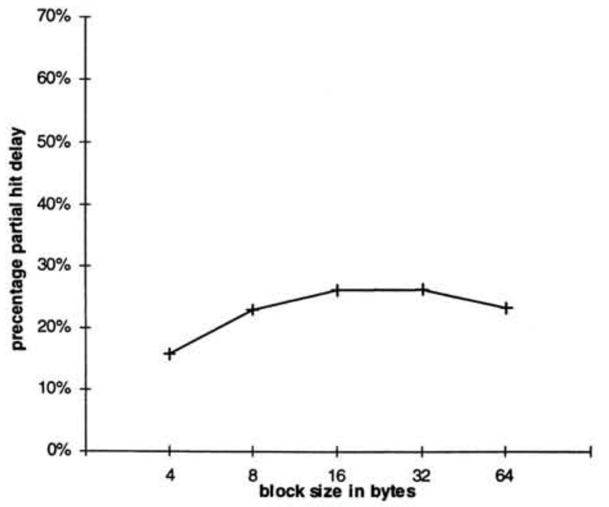
(c) nasa7



(d) wave5



(c) nasa7



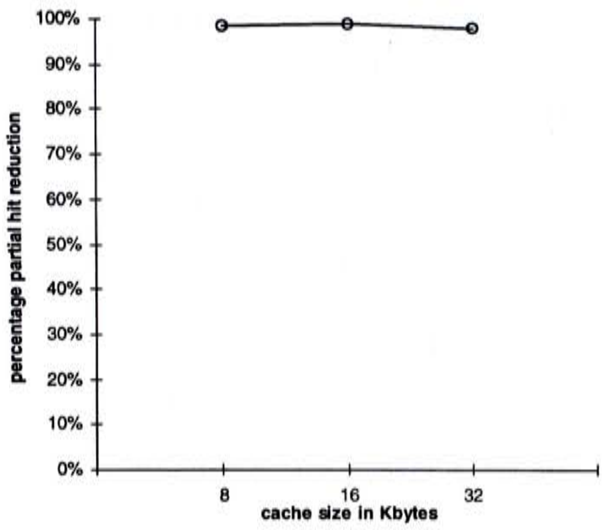
(d) wave5

Figure 4.9: Percentage partial hit delay to overall delay for the basic IAP scheme

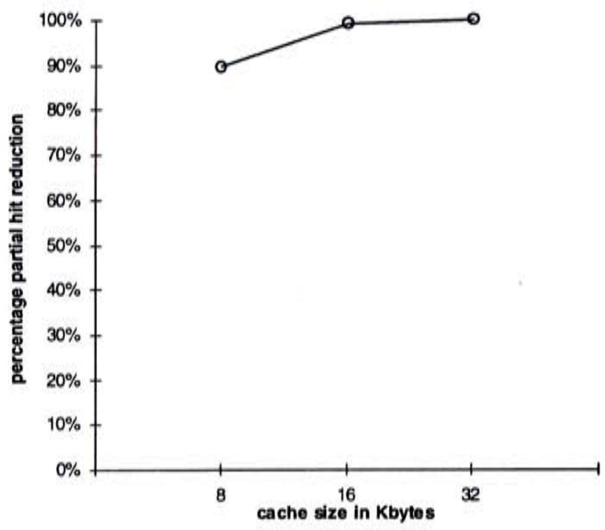
As shown in Figures 4.9, for the basic IAP scheme, an average of about 20% of the memory stall time in the program wave5 is caused by the partial hits. The situation is more serious in the program nasa7, where the partial hit delay accounts for more than 50% of the total memory delay for most cases.

In order to handle the problem, the enhanced IAP scheme incorporates the concept of cache block prefetching — based on the stride, the scheme will prefetch the cache block that follows the current referencing block if the prefetched address and current address correspond to the same cache line. Actually, it has similar effect as a scheme with multiple iterations look ahead. The number of iterations look ahead will be equal to *cache_block_size/stride_size*. Figure 4.10 shows the percentage partial hit delay reduction of the enhanced IAP scheme over basic IAP scheme for the program nasa7 and wave5.

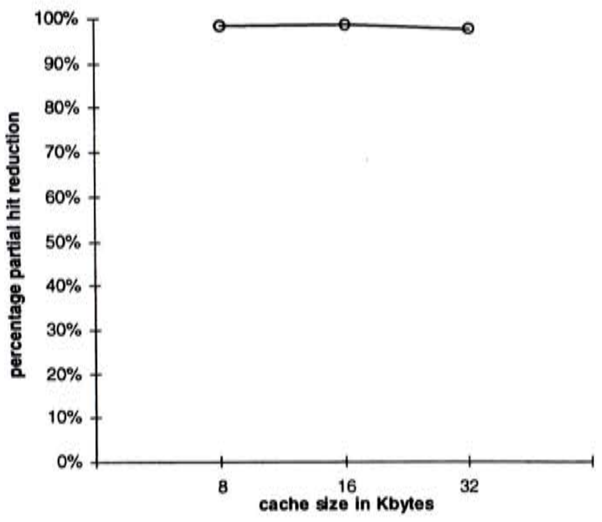
$$\%\_reduction = \frac{partial\_hit\_delay_{basic\_IAP} - partial\_hit\_delay_{enhanced\_IAP}}{partial\_hit\_delay_{basic\_IAP}} \times 100$$



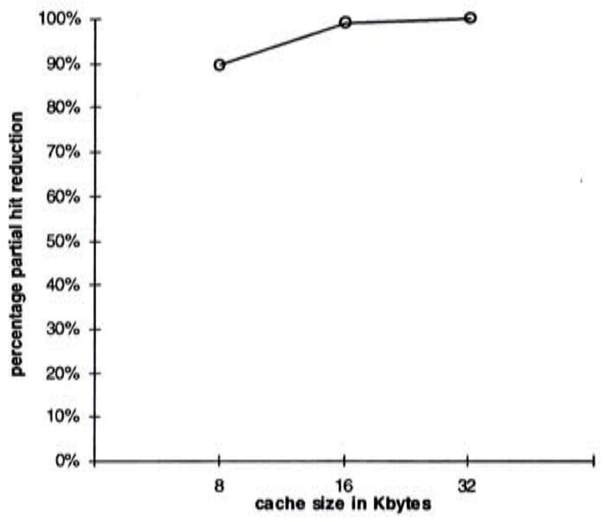
(a) nasa7



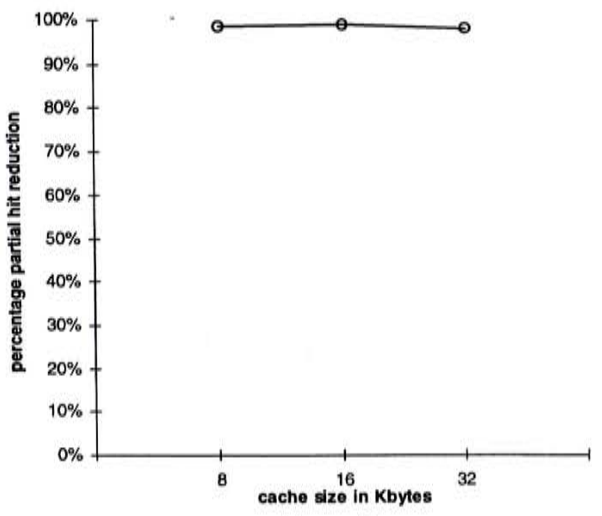
(b) wave5



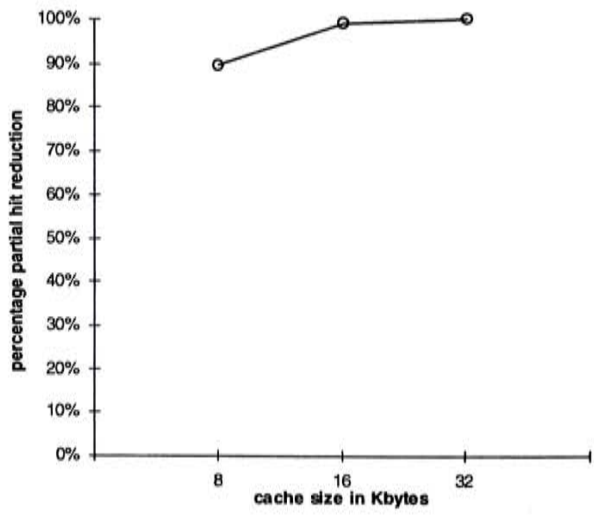
(a) nasa7



(b) wave5



(a) nasa7



(b) wave5

Figure 4.10: Partial hit delay reduction over the basic IAP scheme

From the results, we are confident that the enhanced IAP scheme has substantial improvement over the basic IAP scheme in reducing the partial hit delay. In most cases, over 90% of partial hit delay is eliminated by the enhanced IAP scheme. However, when observing the curves with varying block size (Figure 4.10(c,d)), one can find out that the enhanced IAP scheme seems to have no effect when the block size is small (4 or 8 bytes) and the percentage only rises after the block size is increased. The results are not surprising and let us take nasa7 as an example for explanation. Nasa7 is a double precision floating point benchmark and the data used in the program are solely 8 bytes in size. That is, the stride size will probably greater than 8 bytes. With a cache block size of only 4 or 8 bytes, as the formula *cache_block_size/stride_size* suggests, the enhanced IAP scheme will have only one iteration look ahead and will revert back to the basic IAP scheme and shows no improvement over the basic IAP scheme. The situation will be better for wave5, since it is a single precision program and the enhanced IAP starts to show some effect for 8 bytes cache block size.

4.3.3 Bus usage problem

In most of the current memory designs, one will probably find that there is only one memory bus between the first and second level cache. With the capability of serving only one request at a time, there are always situations when one type of requests is being served, the other type of requests collides — the bus contention problem. The processor must be stalled until the data needed is available in the first level cache because any delay to the demand fetch requests will definitely introduce extra processor stall time. Therefore, demand fetch requests are always given higher priorities to use the bus than prefetch requests. That is, if a demand fetch request due to some cache miss is issued and the bus is busy serving another prefetch request, the prefetch request will be killed and then the demand fetch request is served. This kind of bus contention problem is quite annoying, as all the efforts spent on the killed prefetch request will be in vain. Moreover, in common cache designs, if there is an on-going prefetch request on the bus and the request needs to be killed, a one cycle penalty for killing the current fetch request on the bus would probably be introduced. So, killing a prefetch request is not completely free. On the other hand, since the accuracy of the prefetch requests generated by the IAP scheme is very high, killing a prefetch request might imply that a cache miss will be encountered later. This cache miss will issue a demand fetch which may further kill another prefetch request in the memory bus, thus starting a chain of bus collisions.

In the basic IAP scheme, if the address of the prefetched data and that of the current referenced data corresponds the same address, no prefetch will be issued. In this case, actual prefetch requests will be issued only when the current data are located at the ends of the cache blocks. If these prefetch requests are accidentally killed by other demand fetch requests, no further action will be executed by the scheme even if the bus is free after the killing demand fetch requests are finished. With high prefetching accuracy of the basic IAP scheme, the killings of the prefetch requests are almost sure to cause other cache misses and consequently, serious bus contention problems.

However, in the enhanced IAP scheme, the prefetching actions are issued based on the cache block addresses. Prefetch requests are sent out once the data at the beginnings of the blocks are referenced. In previous section, it has been shown that this enhancement offers more time for the prefetch requests to be finished, thus avoiding the partial hit delay problem. In addition to this advantage, the enhanced IAP scheme is also more reliable in prefetching data. For example, as shown in Figure 4.11, when *a[1]* is referenced, a prefetch request to the cache block *i+1* will be sent out. If this request is killed by other demand fetch request, another prefetch request to the same cache block *i+1* will be issued when *a[2]* is referenced. The scheme will keep on retrying to prefetch the block until [1] the block is successful brought into the cache by prefetching, or [2] when *a[5]* is referenced. A real cache miss is occurred in case [2] and the block *i+1* will be brought into the cache by a demand fetch request.

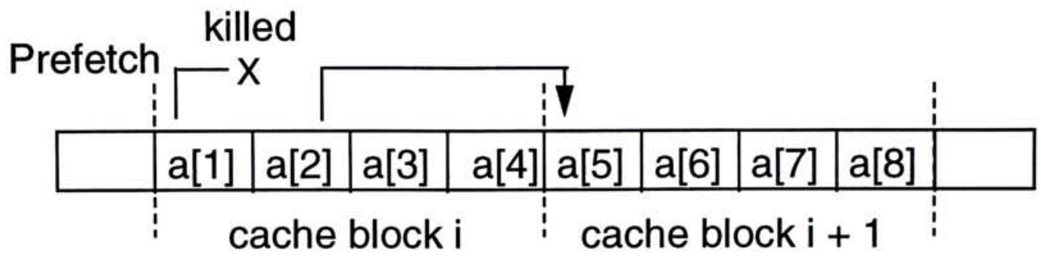
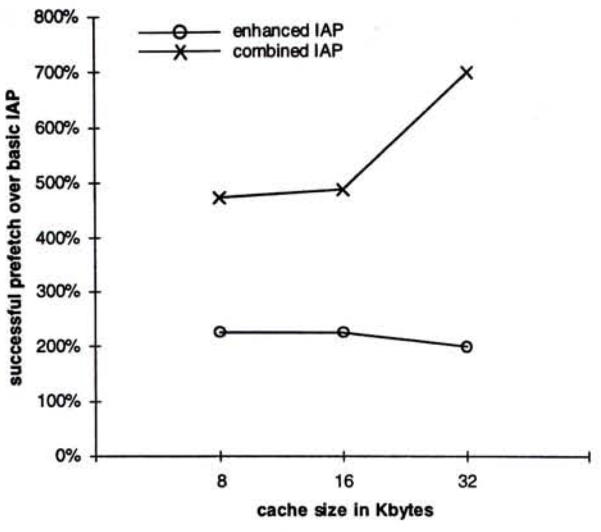
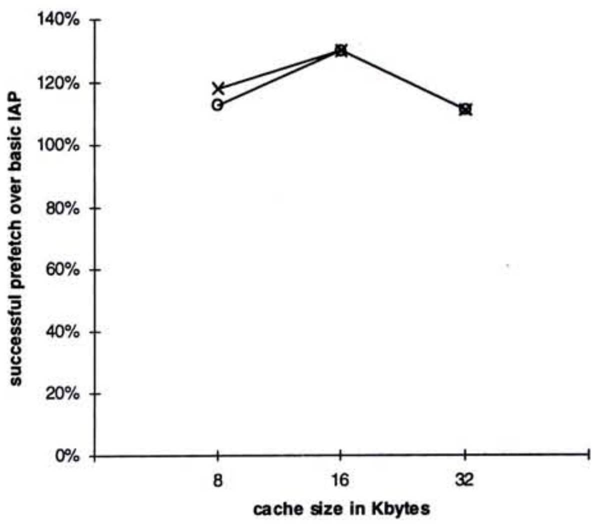


Figure 4.11: Retry after prefetch killed

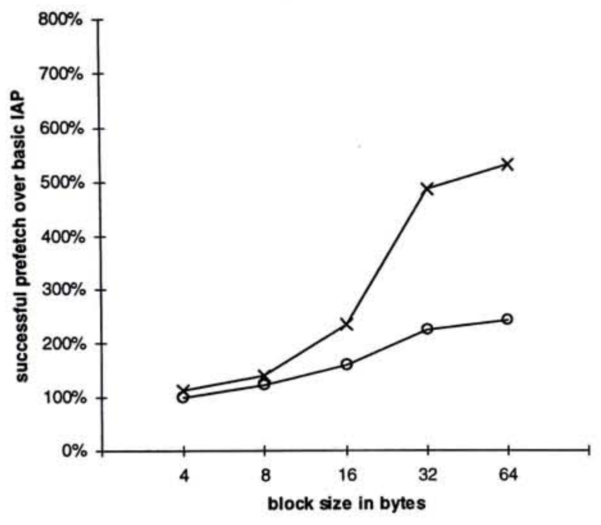
In order to study how this "concept of retrying" helps, the "number of successful prefetch blocks over the basic IAP scheme" is defined. It is the ratio of the total number of the successful prefetch blocks of the enhanced and combined IAP scheme to that of the basic IAP scheme, where successful prefetch blocks refer to those cache blocks brought into the first level cache by prefetch requests and are referenced at least once by the processor before they are replaced. The results for espresso and tomcatv are shown in Figure 4.12.



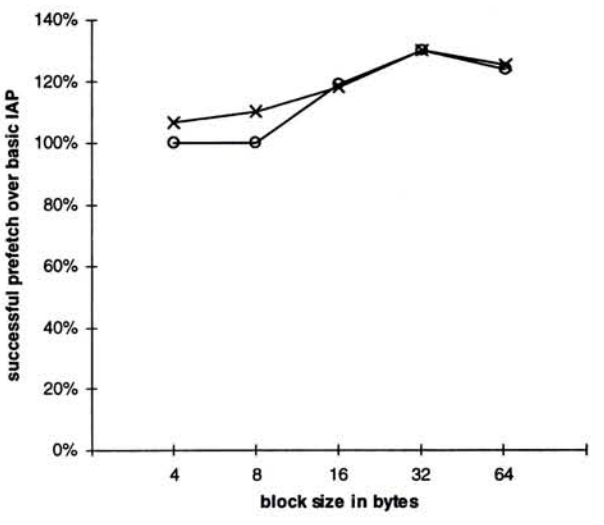
(a) espresso



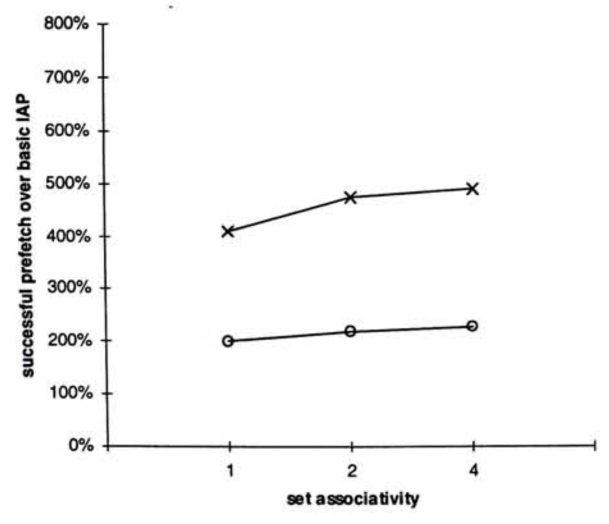
(b) tomcatv



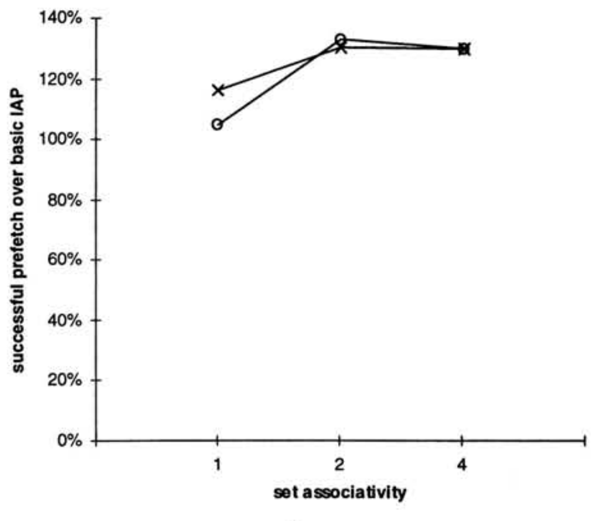
(c) espresso



(d) tomcatv



(e) espresso



(f) tomcatv

Figure 4.12: Number of successful prefetch blocks over the basic IAP scheme

As the results illustrate, the enhanced IAP scheme is more successful than the basic IAP scheme in prefetching blocks into the cache except the cases when the block sizes are small. As explained in last section, it is due to the reason that the enhanced IAP will revert back to a basic IAP scheme when the block sizes are smaller or equal to the size of data used in the programs. The combined IAP scheme, incorporated with default prefetching, attains more successful prefetch blocks than the other two schemes. For expresso, the results for the combined IAP scheme are more significant, as a large portion of *LOAD/STORE* instructions belongs to *non-LOAD/STORE-UPDATE* type (refer to Figure 4.3) and the number of prefetch requests triggered by the default prefetching is large. The results also show that the enhanced and combined IAP schemes are more aggressive in prefetching data and this helps explaining why the enhanced and combined IAP scheme have better performances than the basic IAP scheme even though their accuracy of prefetching are a bit lower than that of the basic IAP scheme.

4.4 Zero time prefetch

Figure 4.13(b,d,f) and Figure 4.14(b,d,f) repeat the experiments for expresso and nasa7 with the assumption of zero time prefetching. That is, data prefetching is assumed to be infinitely fast and no prefetching request will be killed in the middle due to the bus contention with demand fetch requests. The purpose of this study is to find out how effective our IAP schemes can overlap the data fetching time with the processor execution. For the basic IAP scheme, zero time prefetching can help the cache performance significantly. With only one iteration look ahead in the basic IAP scheme, all prefetch requests must be finished in one iteration time in order to have no processor stalls. This bursty traffic requirement significantly increases the bus traffic demand of the basic IAP scheme, resulting in the insufficient bandwidth to allow data to return in time to be used. On the other hand, by comparing the curves of Figure 4.13(a,c,e) and Figure

4.14(a,c,e) with Figure 4.13(b,d,f) and Figure 4.14(b,d,f), we can see that the performance difference between the combined IAP scheme with non-zero prefetch time and the combined IAP scheme with zero prefetch time is very small, only a few percents. The introduction of zero time prefetching to the combined IAP scheme has almost negligible effect to the cache performance. This shows that the combined IAP scheme can prefetch data into the cache accurately as well as overlapping the data fetching time with the program execution time effectively.

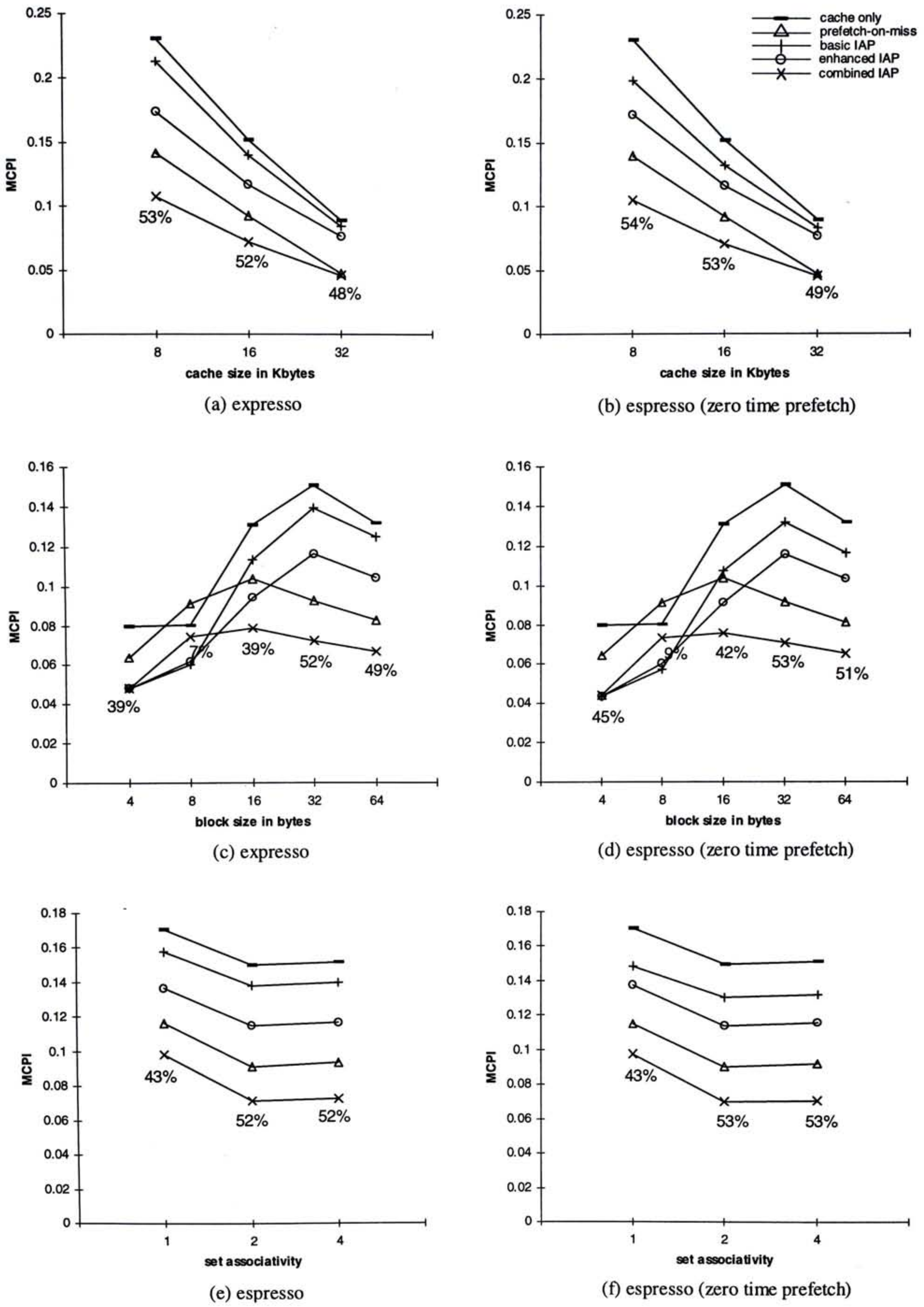
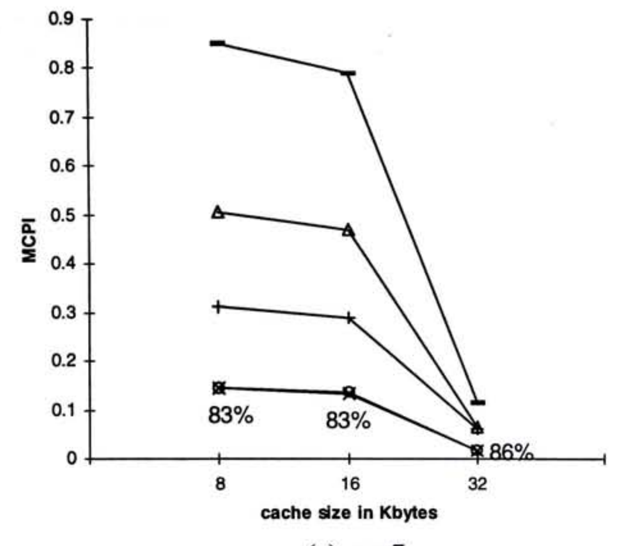
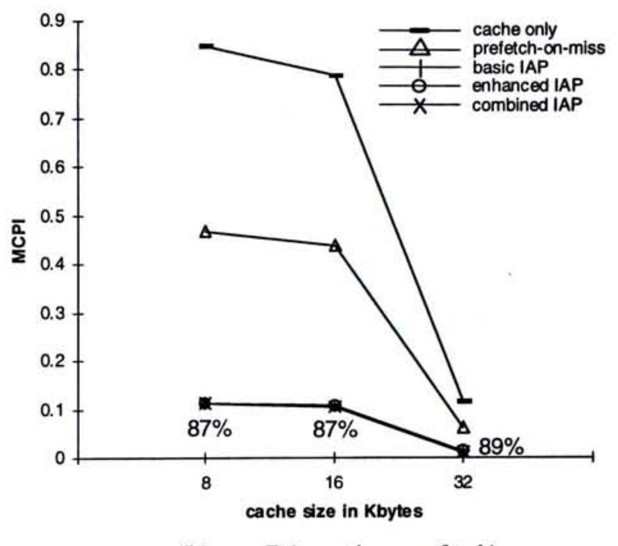


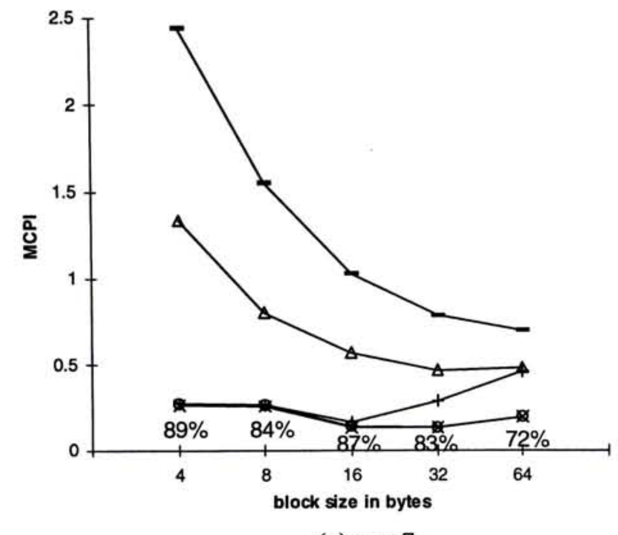
Figure 4.13: Experimental results with zero time prefetch for espresso



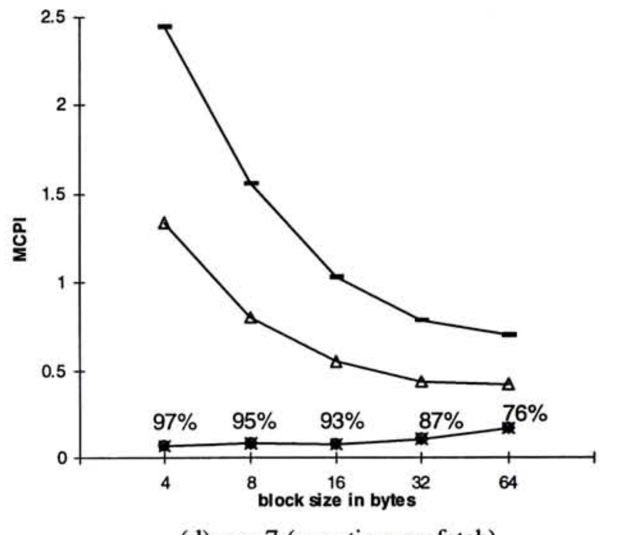
(a) nasa7



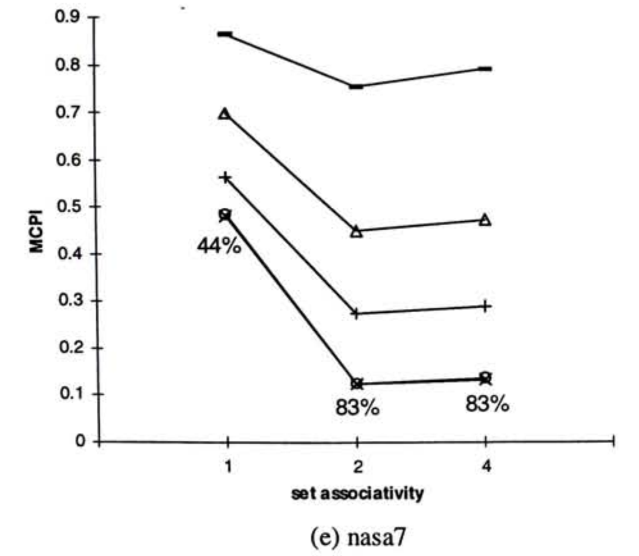
(b) nasa7 (zero time prefetch)



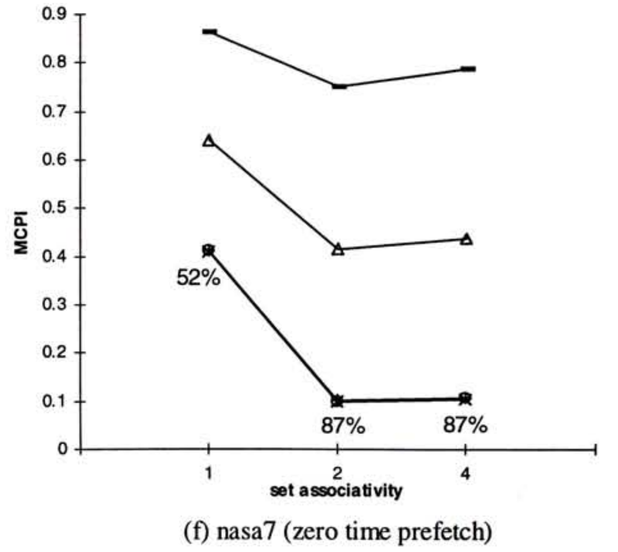
(c) nasa7



(d) nasa7 (zero time prefetch)



(e) nasa7



(f) nasa7 (zero time prefetch)

Figure 4.14: Experimental results with zero time prefetch for nasa7

4.5 Summary

In this chapter, the performance of the basic, enhanced and combined IAP scheme is evaluated using cycle by cycle simulations of the eight SPEC92 benchmarks. For comparison, the performance of a traditional hardware prefetching scheme, prefetch-on-miss, is also included. Cache models with varying cache size, block size and associativity are simulated. Except the slight performance degradation for the benchmark program compress, the results show that the IAP schemes are generally effective in reducing the data access penalty in almost all the other benchmark programs tested. It is observed that the enhanced IAP scheme has moderate improvement over the basic IAP scheme and the combined IAP scheme shows performance improvement over the enhanced IAP scheme in some cases where the spatial locality of data is not fully exploited by the enhanced IAP scheme.

Next, an intensive study of the IAP schemes is carried out using some implicit performance parameters — accuracy of prefetching, partial hit delay time reduction and number of successful prefetch blocks to investigate why and how the IAP schemes work. The results obtained help to illustrate that the IAP schemes can prefetch data accurately and the enhanced and combined IAP are useful in reducing the delay time caused by partial hits and also more aggressive and successful in prefetching data into the cache.

Finally, the experiments are repeated with zero time prefetch, that is, the caching model with the assumption of infinitely fast data prefetching. The results show that the IAP schemes can prefetch data into the cache accurately and as well as successfully overlap the data fetching time with the program execution time effectively.

Chapter 5

Conclusion

5.1 Summary of our research

In this dissertation, three IAP (Instruction Opcode and Addressing Prefetching) schemes, the basic, enhanced and combined, are proposed one by one based on the following observations from various experiments:

- We observe that certain important and very common instruction opcodes and addressing modes actually contain valuable information about what data are expected to be referenced in the near future.
- Most current data prefetching schemes (both hardware and software) only prefetch one cache block, independently of what kind of prefetching strategies they use. That is, they do not have the concept of multiple cache blocks prefetching. However, the next datum which consists of multiple bytes (depending on whether it is single, double, or quadwords), might fall into multiple blocks. Thus, if the datum falls into two or more cache blocks and only the first block is prefetched into the cache, the rest of the datum will still cause cache misses.
- In current software assisted prefetching schemes, the cache activities generated by the *PREFETCH* instruction are based on the data address instead of the cache block address. Consequently, the time period between the issue of a prefetch request and the use of the datum might be too short for the prefetching to be finished in time.
- We found that most software assisted data prefetching schemes using *PREFETCH* instructions might have a very important potential problem of request collisions between demand data fetch requests and very accurate data prefetch requests. That is, while a data prefetch request is being served, a data cache miss occurs and both

requests try to use the same data bus. This is usually due to the one iteration look ahead in software assisted data prefetching scheme and the lack of cache block concept in the *PREFETCH* instruction. As a result, even though prefetch requests issued by software prefetching schemes can be very accurate, they might be killed in the middle of their data prefetching by some data demand fetch requests. This introduces more demand fetch requests when the data are actually referenced in the next iteration. The newly introduced demand fetch requests may then kill other on-going prefetch requests. As a result, a chain of demand fetch misses may be caused. This collision problem will be more serious if [1] the speed gap between the instruction execution rate and the data fetching time is large, or [2] the cache hit ratio is low (e.g. less than 70%).

We propose a collection of three new hardware driven data prefetching schemes, called the IAP schemes, to improve data cache performance. From our simulation study, we saw that the potential of these IAP schemes is very good and the processor idle time due to memory accesses can easily be reduced substantially by the IAP schemes. The nice things about these IAP schemes are: [1] the additional hardware required is very simple, [2] no change in the architecture is required, [3] no new compiler optimization technique is required, [4] the IAP scheme can work with other data prefetching schemes to obtain further cache performance improvement, and [5] the potential performance improvement obtained by the IAP is very big. Furthermore, several observations and enhancements that we made for the IAP schemes (as were discussed in Chapter 3) can also be applied to most software assisted cache prefetching schemes to further improve the cache performance.

5.2 Future work

The IAP schemes that we proposed so far requires the definition of *LOAD/STORE-UPDATE* compound instructions in the architecture. An example of such an architecture is the POWER series, as in the IBM/Motorola/Apple PowerPC and in the IBM RS/6000. However, the question that needs to be answered is — can the IAP schemes be extended easily to other architectures without *LOAD/STORE-UPDATE* instructions?

There are two answers to this question. First, some architectures have some compound instructions that are functionally equivalent to the *LOAD/STORE-UPDATE* instructions defined in the IBM PowerPC or RS/6000 series. As is mentioned in Chapter 3, the HP's Precision Architecture (PA RISC) 1.1 has *LOAD/STORE-MODIFY* instructions. Thus the IAP schemes can be extended easily to this type of architectures without any difficulties. Second, for architectures such as SPARC, where no such compound instructions are defined, it is still possible to implement the IAP schemes provided that an update-counter per register is available. The main purpose of this update-counter *UC* is to book-keep if its corresponding register *R(UC)* is an index register used by some *LOAD/STORE* instructions using "index-displacement" addressing mode in a loop. If the answer is yes, the value of the stride used by the "index displacement" *LOAD/STORE* instructions will be learnt during the first iteration of the loop and it will be stored in the update-counter *UC*. After that, very accurate data prefetching similar to the IAP schemes can be carried out in the rest iterations of the loop to improve the cache performance.

Bibliography

[BaC91] Baer, J.L., Chen, T.F., "An effective on-chip preloading scheme to reduce data access penalty," *Proceedings of the 1991 International Conference on Supercomputing,* November 1991, pp. 176- 186.

[Bre87] Brent, G.A., "Using program structure to achieve prefetching for cache Memories," *Ph.D Thesis, University of Illinois at Urbana-Champaign,* January 1987.

[CaK91] Callahan, D., Kerlnedy, K., Porterfield, A., "Software prefetching," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1991, pp. 40-52.

[ChB92] Chen, T.F., Baer, J.L., "Reducing memory latency via non-blocking and prefetching caches," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1992, pp.51-61.

[ChM91] Chen, W.Y., Mahlke, S.A., Chang, P.P., Hwu, W.W., "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," *Proceedings of Microcomputing 24,* 1991.

[FuP91] Fu, W.C., Patel, J.H., "Data prefetching in multiprocessor vector cache memories," *Proceedings of the l8th Annual Symposium on Computer Architecture*, May, 1991 , pp.54-63.

[FuP92] Fu, W.C., Patel, J.H., "Stride directed prefetching in scalar processors," *Proceedings of the 25th International Symposium on Computer Architecture*, 1992, pp. 102-110.

[GiM86] Gibbons, P.B., Muchnick, S.S., "Efficient instruction scheduling for a pipelined architecture," *Proceedings of SIGPLAN Symposium on Compiler Construction*, 1986.

[GoG90] Gornish, E., Granston, E., Veidenbaum, A., "Compiler-directed data prefetching in multiprocessor with memory hierarchies," *Proceedings of the 1990 International Conference on SuperComputing*, 1990, pp.354-368.

[HeP90] Hennessy, J., Patterson, D., *Computer Architecture: A Quantitative Approach,* Morgan Kauffmann, 1990.

[HP94] Hewlett-Packard, Inc., *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, HP Part Number 09740-90039, third Edition; February 1994.

[IBM89] *IBM AIX V3.2 for RISC Systems/6000: Assembler Language Reference* SC23-2197-01, 1989.

[IBM94] IBM, *The PowerPC Architecture,* edited by May, C., Silha, E., Simpson, R., Warren, H., Morgan Kauffmann, 1994.

[Jou90] Jouppi, N.P., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proceedings of the 18th Annual Symposium on Computer Architecture,* May 1990, pp.364-373.

[KlL91] Klaiber, A.C., Levy, H.M., "An architecture for software controlled data prefetching," *Proceedings of the 18th Annual Symposium on Computer Architecture,* May 1991, pp.43-53.

[Lee87] Lee, R.L., "The effectiveness of caches and data prefetch buffers in large-scale memory multiprocessors," *Ph.D Thesis, Department of Computer Science, University of Illnois at Urbana-Champaign*, May 1987.

[MoG91] Mowry, T.C., Gupta, A., "Tolerating latency through software-controlled prefetching in shared-memory multiprocessor," *Journal of Parallel and Distributed Computing,* Volume l, Number 2, June 1991, pp.87-106.

[MoL92] Mowry, T.C., Lam, M.S., Gupta, A., "Design and evaluation of a compiler algorithm for prefetching," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1992, pp. 62-73.

[Mot92] Motorola Inc., *PowerPC 601 RISC Microprocessor User's Manual,* Publication Number MPC601UM/AD,1992.

[Por89] Porterfield, A.K., "Software methods for improvement of cache performance on supercomputer application", *Technical Report COMP TR 89-93, Rice University,* May 1989.

[Sak72] Sakalay, F.E., "Storage hierarchy control system," *IBM Technical Disclosure Bulletin*, Volume 15, Number 4, September 1972, pp.1100.

[SmG83] Smith, J.E., J.R. Goodman, "A study of instruction cache organisations and replacement policies," *Proceedings of the Tenth Annual Symposium on Computer Architecture*, June 1983, pp. 132-137.

[Smi78a] Smith, A.J., "Sequentially and prefetching in data base systems," *ACM Transactions on Data Base Systems,* Volume 3, Number 3, 1978, pp.223-247.

[Smi78b] Smith, A.J., "Sequential program prefetching in memory hierarchies," *IEEE Computer*, Volume 11, Number 12, 1978, pp.7-21.

[Smi82] Smith, A.J., "Cache memories," *ACM Computing Survey,* Volume 14, Number 3, September 1982, pp.473-530.

[Tha81] Thabit, K.D., "Cache management by the compiler," *Ph.D Thesis, Rice University*, November 1981.

[WeS94] Weiss, S., Smith, J.E., *POWER and PowerPC*, Morgan Kauffmann, 1994.