ABSTRACT

LIM, CHUNGSOO. Enhancing Dependence-based Prefetching for Better Timeliness, Coverage, and Practicality. (Under the direction of Dr. Gregory T. Byrd).

This dissertation proposes an architecture that efficiently prefetches for loads whose effective addresses are dependent on previously-loaded values (dependence-based prefetching). For timely prefetches, the memory access patterns of producing loads are dynamically learned. These patterns (such as strides) are used to prefetch well ahead of the consumer load. Different prefetching algorithms are used for different patterns, and different algorithms are combined on top of dependence-based prefetching scheme. The proposed prefetcher is placed near the processor core and targets L1 cache misses, because removing L1 cache misses has greater performance potential than removing L2 cache misses.

For higher coverage, dependence-based prefetching is extended by augmenting the dependence relation identification mechanism, to include not only direct relations ($y = x$) but also linear relations ($y = ax + b$) between producer (x) and consumer (y) loads. With these additional relations, higher performance, measured in instructions per cycle (IPC), can be obtained.

We also show that the space overhead for storing the patterns can be reduced by leveraging chain prefetching and focusing on frequently missed loads. We specifically examine how to capture pointers in linked data structures (LDS) with pure hardware implementation. We find that the space requirement can be reduced, compared to previous work, if we selectively record patterns. Still, to make the prefetching scheme generally applicable, a large table is required for storing pointers. So we take one step further in order to eliminate the additional storage need for pointers. We propose a mechanism that utilizes a portion of the L2 cache for storing the pointers. With this mechanism, impractically huge on-chip storage for pointers, which is sometimes a total waste of silicon, can be

removed. We show that storing the prefetch table in a partition of the L2 cache outperforms using the L2 cache conventionally for benchmarks that benefit from prefetching.

Enhancing Dependence-based Prefetching for Better Timeliness, Coverage, and Practicality

by
Chungsoo Lim

A dissertation submitted to the Graduate Faculty of
North Carolina State University
In partial fulfillment of the
Requirements for the degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2009

APPROVED BY:

_____	_____
Dr. Gregory T. Byrd Chair of Advisory Committee	Dr. Vincent W. Freeh
_____	_____
Dr. Eric Rotenberg	Dr. Yan Solihin

BIOGRAPHY

Chungsoo Lim was born in South Korea in 1971. He received the Bachelor of Science degree in Department of Electrical Engineering from Inha University in 1997, and the Master of Science degree in Computer Engineering from University of Maryland at College Park in 2004. In 2004, he started his work as a doctoral student under the guidance of Dr. Gregory T. Byrd. Chungsoo Lim's research interest includes data prefetching and value prediction..

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support of many people. First of all, I would like to thank my advisor Dr. Gregory T. Byrd for his thoughtful directions and affectionate encouragements. I would like to acknowledge my advisory committee members, Dr. Yan Solihin, Dr. Vincent W. Freeh, Dr. Eric Rotenberg, and Dr. Suleyman Sair for their enlightening comments and constructive suggestions on my work. I would like to acknowledge friends from Duraleigh Presbyterian church. I would like to thank my wife, Eunkyoung, my son, Heejoon, my parents, and parents-in-law for enduring this long process with me, always offering support and love. Last but not lease, I'd like to thank God for his love and guidance.

TABLE OF CONTENTS

LIST OF FIGURES..... vi

LIST OF TABLES..... viii

Chapter 1 Introduction……..... 1

1.1 Motivation…….. 1
1.2 High level view of prefetching…….. 2
1.3 High level view of the proposed prefetching mechanism……...……..….. 5
1.4 Contributions……. 7

Chapter 2 Related Work..... 9

2.1 Alternatives to prefetching..... 9
2.2 Hardware prefetching schemes..... 11
2.3 Software prefetching schemes..... 16

Chapter 3 Timely Dependence-based Prefetching..... 18

3.1 Motivation..... 18
3.2 Dependence-based prefetching mechanism – high level view..... 20
3.3 Proposed dependence-based prefetching mechanism – high level view..... 22
3.3.1 Key differences with existing mechanisms..... 22
3.3.2 Jump pointer creation method..... 24
3.3.3 Filtering mechanism..... 27
3.4 Implementation..... 28
3.4.1 Inserting into CT..... 30
3.4.2 Updating CT..... 31
3.4.3 Looking up CT..... 33
3.4.4 An alternate CT structure…….. 35
3.5 Evaluation..... 36
3.5.1 Experimental Setup..... 36
3.5.2 Effectiveness of hardware-based JPT creation mechanism..... 37
3.5.3 Jump pointer distance..... 40
3.5.4 Jump pointer accuracy..... 42
3.5.5 Sensitivity of the correlation table to ports and access latency..... 44

3.5.6 Sensitivity of the potential producer window to ports and access latency.. 47
3.5.7 Effectiveness of the proposed prefetcher..... 49
3.6 Conclusion..... 54

Chapter 4 Exploiting Linear Dependence Relations to Enhance Prefetching..... 55

4.1 Motivation..... 55
4.2 Linear relations..... 56
4.3 Distributions of direct and linear relations..... 58
4.4 Added hardware complexity..... 60
4.4.1 Arithmetic units..... 63
4.4.2 Dependence chain tracking logic..... 63
4.4.3 Candidate pair table..... 64
4.4.4 Data forwarding logic (direct-relation filtering mechanism)..... 66
4.5 Evaluation..... 67
4.5.1 Speedups..... 67
4.5.2 Sensitivity to the size of the L2 cache……..... 73
4.5.3 Sensitivity of the correlation table to ports and access latency…….. 77
4.5.4 Sensitivity of the potential producer window to ports and access latency.. 80
4.6 Conclusion..... 82

Chapter 5 Using L2 Cache for Storing Prefetch Table..... 84

5.1 Motivation..... 84
5.2 Reconfigurable cache..... 87
5.3 Implementing prefetch table in L2 cache..... 88
5.4 Evaluation..... 91
5.4.1 Impact of the L2 cache size on overall performance..... 91
5.4.2 Performance with varying the partition size for JPT..... 92
5.4.3 The effectiveness of the hash function..... 98
5.5 Conclusion..... 99

Chapter 6 Conclusion..... 100

REFERENCES..... 104

LIST OF FIGURES

Figure 3.1 Schematic of DBP..... 21
Figure 3.2 Prefetching schemes for LDS..... 25
Figure 3.3 Organization of the Proposed Prefetcher..... 29
Figure 3.4 Example of Correlation Identification Unit..... 31
Figure 3.5 Example of Jump Pointer Creation Unit and Jump Pointer Table..... 33
Figure 3.6 Example of Correlation Table..... 34
Figure 3.7 The effectiveness of JP creation mechanisms in terms of IPC..... 40
Figure 3.8 Prefetch reference breakdown with varying jump pointer distance..... 41
Figure 3.9 Influence of jump pointer distance on IPC..... 42
Figure 3.10 Accuracy with varying jump distance..... 44
Figure 3.11 Sensitiveness of the CT to the number of ports..... 46
Figure 3.12 Sensitiveness of the CT to its access latency..... 47
Figure 3.13 Sensitiveness of the PPW to the number of ports..... 48
Figure 3.14 Sensitiveness of the PPW to its access latency..... 49
Figure 3.15 Breakdown of prefetch hits: stride-based prefetch, jump pointer-based prefetch, and chain-prefetch..... 50
Figure 3.16 Performance comparison..... 53
Figure 4.1 Code segments from *mcf* and *bzip2*..... 57
Figure 4.2 Extra hardware for linear mechanisms..... 62
Figure 4.3 Candidate pair table implementation..... 66
Figure 4.4 Performance comparison with 1MB L2 cache..... 71
Figure 4.5 Performance comparison with 2MB L2 cache..... 71
Figure 4.6 Performance comparison with 4MB L2 cache..... 72
Figure 4.7 Performance comparison with 8MB L2 cache..... 72
Figure 4.8 Impact of the L2 cache size on the performance..... 73
Figure 4.9 Performance improvements for *treeadd* and *swim*..... 74

Figure 4.10 Performance improvements for *bzip* and *art*…. 76
Figure 4.11 Sensitiveness of the CT to the number of ports. 79
Figure 4.12 Sensitiveness of the CT to its access latency. 80
Figure 4.13 Sensitiveness of the PPW to the number of ports. 81
Figure 4.14 Sensitiveness of the PPW to its access latency. 82
Figure.5.1 Implementation of a reconfigurable cache. 88
Figure.5.2 Implementing a jump pointer table in the L2 cache. 89
Figure.5.3 The impact of the L2 cache size on overall performance. 92
Figure.5.4 Performance with varying the partition size for the jump pointer table in L2 caches: 1MB L2 Cache (top), 2MB L2 cache (middle), and 8MB (bottom) for *mcf*. 95
Figure.5.5 Performance with varying the partition size for the jump pointer table in L2 caches: 1MB L2 Cache (top), 2MB L2 cache (middle), and 4MB L2 cache for *health*. 97
Figure.5.6 The effectiveness of the hash function. 98

LIST OF TABLES

Table 3.1 Simulation parameters..... 37
Table 3.2 The number of jump pointers created for *mcf* and *health*..... 39
Table 4.1 Frequency of occurrence (%) of dependence patterns..... 59
Table 5.1 Design Choices for Reconfigurable Cache..... 91

Chapter 1

Introduction

1.1. Motivation

During the past decades, the gap between processor speed and memory speed widened, and this has been the most serious impediment to achieving higher performance. This phenomenon is due to trends in processor and memory designs. The trend in processor design is that processor clock period is reduced to achieve higher throughput, but the trend in memory design is in the direction of higher densities rather than in the direction of lower access latencies.

Due to this wide gap, current processors need more instruction-level parallelism to overcome the gap, but finding enough parallelism is not easy in most applications. Consequently, processors are likely to stay idle for many cycles, waiting for memory to provide requested data.

To remedy this problem, a lot of proposals have been presented, and some of them were actually implemented in commercial processors [8, 35]. Many of them are based on speculation. Techniques based on speculation are value prediction [27, 55], prefetching [1, 2, 7, 8, 11, 17, 18, 20, 21, 22, 24, 25, 26, 28, 29, 30, 32, 33, 34, 35, 36, 38, 43, 44, 46, 47, 51, 53, 56, 57, 58, 59, 60], checkpoint-assisted schemes [52], and speculative multi-threading [36, 50, 60]. There are non-

speculative techniques such as decoupled access/execute architecture [49], various cache designs [1, 8, 41, 54], and cache-conscious structure allocation [9, 10].

This dissertation focuses on prefetching mechanisms. There are three reasons for this. First, the potential of prefetching is very high, and none of the existing prefetching schemes is able to achieve the full potential. If prefetching can eliminate all cache misses, all load instructions can be considered as instructions that have execution latency of L1 cache hit latency. Second, prefetching doesn't require a recovery procedure when a mis-prefetch occurs. Some performance degradation may occur, if a useful block is evicted by a prefetched block or if contention happens at the memory interface. However, this degradation can be minimized by giving priority to demand fetches over prefetches. Third, a prefetch mechanism can be easily combined with other techniques. For example, if prefetching is combined with value prediction [14], higher performance may be obtained. This is because value prediction is more effective for reducing dispatch-to-issue latency and prefetching can reduce issue-to-writeback latency. Prefetching makes value predictions validated earlier than the case without prefetching.

1.2. High level view of prefetching

There are numerous prefetching mechanisms, but they can be classified by two aspects: how to generate prefetch addresses, and where to place prefetched data. The latter can be level 1 (L1) cache, level 2 (L2) cache, or a dedicated prefetch buffer that is usually placed at the same level as L1 cache. If L1 cache is chosen for the storage of prefetched data, extra care is required, because L1 cache is susceptible to cache pollution by prefetched data due to its small capacity. If prefetch buffer is chosen, hardware complexity is elevated, because the buffer should be accessed in parallel with L1 cache and data movement should be supported from the buffer to L1 cache. Many prefetching

schemes advocate using L2 cache, even though getting prefetched data from L2 cache takes time, because L2 cache doesn't suffer cache pollution as much as L1 cache because of its big capacity. Also, a modern out-of-order pipeline may be able to hide L2 cache latency.

How to generate prefetch addresses can be further divided into two things: what to look at and what to look for in order to generate prefetch addresses.

Prefetching mechanisms rely on address streams. Streams can be either whole address streams from the processor, or miss address streams from L1 or L2 cache. By observing only miss address streams, some patterns cannot be discovered, but hardware complexity can be decreased and important patterns can still be observable. These streams can be localized by using program counter (PC) [33] or cache set numbers [16]. Localized stream may lead to finding new patterns and more efficient utilization of resources due to sharing.

In the streams described above, prefetchers try to find patterns that can be utilized to predict future addresses. The two main patterns prefetchers look for are stride pattern [15, 57] and repeated pattern [7, 16, 18, 33, 34, 47]. Stride is a difference between two consecutive elements in a stream; having a stable stride pattern means differences between each consecutive pair in a stream are the same. Stride pattern is very effective in two aspects. First, it requires modest space budget, which makes it easy to implement. Second, it can predict unseen addresses, which is not possible with repeated patterns. However, stride patterns cannot be a panacea for all applications because they are not prevalent in some applications. Repeated patterns can be used for prefetching by correlating an element in a sequence with a few numbers of elements that precede it in the sequence. One special case of repeated pattern is jump pointers [44]. In a linked data structure (LDS), each object generally has pointers to its adjacent objects. Jump pointers refer to non-adjacent objects. When an object is accessed, the object referenced by its jump pointer is prefetched. Since LDS is likely to be traversed in the same order, jump pointers can prefetch objects that will be accessed in the near future.

Repeated pattern-based prefetching requires substantial amount of space and it takes time to learn repeated patterns, but it can cover more loads than stride-based prefetching.

One of more general prefetcher utilizing repeated pattern is distance prefetcher [22, 33]. Distance prefetching is a generalization of Markov prefetching [18]. Distance prefetching was originally designed for prefetching TLB entries, but the mechanism can be readily adapted to data prefetching. Distance prefetching uses the distance between two consecutive global miss addresses to index the correlation table. Each correlation table entry holds a list of deltas that have followed the entry's delta in the past. One useful property of this prefetching scheme is that it can predict unseen addresses, because a sequence of delta can be shared by multiple address streams.

There are some mechanisms that have different ways of generating prefetch addresses. One such mechanism is dependence-based mechanism [43]. It identifies a set of producer instructions for a consumer load, and issues a prefetch request based on the value produced by the producer instructions. This mechanism is able to issue prefetches for almost all candidate loads, but may not be able to fully hide miss latencies.

Ideal prefetching eliminates all cache misses. In order to achieve this goal, a prefetch must be performed for every miss (coverage), each prefetched block should arrive at L1 cache or prefetch buffer before it is actually accessed by processor (timeliness), and every prefetched block has to be accessed by the processor (accuracy). In reality, these three requirements are hard to satisfy at the same time. If we want to increase coverage, more prefetches need to be issued. Among them, unnecessary prefetch requests exist inevitably, lowering accuracy. If we focus on timeliness, addresses of loads far away from the current instruction need to be predicted. This may decrease accuracy. If accuracy is the target, prefetches that have high probability of success are selectively issued, impairing coverage.

More problems arise when actual implementation is considered. Since demand fetches and

prefetches share resources, such as buses or cache, too much contention between the two may end up decreasing performance. This problem should be resolved to enjoy full potential of prefetching. Another practical problem is the implementation cost of prefetching. In order to achieve high coverage, accuracy, and good timeliness, different prefetchers sometimes need to be combined, or prefetcher size is required to be huge in order to keep more history information. However, if prefetcher size is too big, it is not feasible to implement. Getting ample performance boost with reasonable silicon budget is extremely difficult.

Pure software prefetching is typically highly accurate, but incurs runtime overhead and cannot issue prefetches sufficiently far in advance of a load to hide main memory access latencies [56]. Hardware only prefetching is more versatile at targeting various access patterns, hiding much of the main memory access time. However, hardware approach is inferior to software approach in terms of resource utilization and power consumption.

1.3. High level view of the proposed prefetching mechanism

To cope with the issues mentioned in the previous section, we proposed a prefetching mechanism that performs well in all three categories. For coverage, dependence-based scheme [43] is chosen for the base mechanism because it usually has good coverage. For a large portion of loads, it is possible to find their producer loads. This dependence-based framework facilitates combining different prefetching techniques, also improving prefetch coverage. For timeliness, producer loads' memory access patterns are utilized, because these patterns make generating future addresses possible. Storage issue is also addressed by using a portion of the L2 cache for storing correlations.

The proposed prefetching mechanism sits near the processor core, and receives information about committed instructions from the core. From this flow of information, prefetching mechanism

predicts future addresses and deposits prefetch requests into the prefetch request queue. Prefetch request queue sends the requests to L2 cache, and L2 cache provides prefetched blocks to the prefetch buffer, which is accessed at the same time as the L1 cache.

From the information provided by the processor core, the prefetching mechanism first extracts dependence relationships. It is similar to the ones described in the previous section, but it is different in some ways. Previous works identify either a producer load that produces the base effective address of a consumer load or a set of instructions that are involved in producing the effective address of a consumer load. The proposed prefetching mechanism tries to find an instruction that has linear relationship with the effective address of a consumer load. By extending direct relation to linear relation, more relation can be discovered. Note that direct relation is just a subset of linear relation.

To be a consumer load, for which a dependence relation is identified, a load should have some qualifications. First, a load has to miss a lot in the L1 cache. If a load misses occasionally, prefetching for it is likely to be of little help for improving overall performance. Second, a load has to be a part of a cluster of cache misses, because clustered cache misses contribute to severe performance degradation. In the proposed mechanism, a novel filter mechanism is implemented for identifying loads that satisfy these two qualifications.

Once a dependence relation is revealed, the proposed prefetching mechanism tries to find a pattern in the output values of each producer instruction. If a pattern is found, it is used for predicting future output values of the instruction. Patterns the mechanism looks for can be any pattern discussed in the previous section. The reason why dependence-based scheme is augmented this way is that the dependence-based scheme alone cannot hide cache miss penalty completely. Because the producer instruction is usually positioned close to a consumer load, dependence-based prefetch scheme manages to hide only fraction of the cache miss penalty. By combining dependence-based

mechanism and pattern-based prefetching mechanism, both mechanisms are getting benefit from each other. Dependence-based mechanism can be enhanced in such a way that timeliness of prefetching is improved. The space overhead of the pattern-based mechanism can be decreased, because only the information about producer instructions needs to be stored in their tables. Moreover, clever use of chain prefetching can further reduce the space overhead.

Space overhead for combined pattern-based prefetching schemes can be resolved by storing prefetcher tables in reconfigurable L2 cache [41]. Instead of a dedicated prefetch table, part of the L2 cache is used for prefetcher tables. There are two motivations for this approach. First, L2 cache hit ratio is usually not as high as that of L1 cache, meaning there is ample potential for utilizing the cache more effectively. One way of doing so is to use it for different processor activities such as prefetching. Second, since a prefetching mechanism is unlikely to work well with all applications, it can be a waste to have a big prefetch table close to the processor core.

1.4. Contributions

Contributions of this dissertation are listed as follows:

- We propose using a dependence-based prefeteching scheme as a general prefetcher framework, in which different prefetching schemes are readily combined. The purpose of the prefetchers built on the dependence-based scheme is to improve the timeliness of dependence-based scheme by predicting future addresses of producers. The role of dependence-based prefetching mechanism is two-fold. First, it provides a foundation for integrating multiple prefetchers together. For this, producers are classified into different groups and each group is associated with a different prefetcher. As well as the selection mechanism for multiple

prefetchers, dependence-based prefetching scheme can also reduce the number of correlations that should be stored to make prefetches. Compared to a conventional correlation-based prefetching mechanism, correlation-based prefetching mechanisms built on top of dependence-based prefetching scheme requires only a fraction of correlations due to two reasons. First, the latter focuses only on producers' stream, which is a subset of the entire stream. Second, the number of correlations can be reduced by virtue of chain prefetching, which is enabled by dependence relations learned by the dependence-based mechanism.

- We propose extending the dependence identification mechanism of dependence-based prefetching mechanism to include linear relations as well as direct relations. With this method, a dependence relation is expressed as a linear equation, anda larger number of important relations can be identified. These new relations generate more useful prefetches, resulting in higher performance. Additional hardware complexity and space overhead are studied, and ancillary filtering mechanisms are also proposed.

- We propose using a portion of the L2 cache for storing the prefetcher table. In the case of a correlation-based prefetcher, its table size is generally too large to implement in pure hardware. To remedy this problem, and to manage the table more efficiently, an existing structure, L2 cache, is chosen to store correlations. Since this approach requires dividing up L2 cache into two partitions, a reconfigurable cache is leveraged. For the benchmarks used for this study, it is shown that storing prefetch table in L2 cache is feasible and has significant potential.

Chapter 2

Related Work

2.1. Alternatives to prefetching

As mentioned in section 1.1, there are many alternatives to prefetching. Some of them are based on speculation and some of them are not. In this section, exemplary works are briefly presented.

One speculation-based alternative is load value prediction [27, 55]. Load value prediction is proposed to break data dependence between load instructions and their dependent instructions. By doing so, dependent instructions can be executed speculatively without waiting for memory to provide the data. But the amount of parallelism uncovered by value prediction is limited by the scheduling window size, and in case of misprediction, the pipeline has to recover from it, wasting tens of cycles.

There are many other speculation-based schemes such as checkpoint-assisted schemes and speculative multi-threading. Checkpoint-assisted schemes [52] relieve pressure on the instruction window by allowing instructions to speculatively retire from the instruction window, creating an effect of having bigger instruction window. When a misprediction occurs, a checkpoint is used to

recover processor states to their most recent non-speculative states. With speculative multi-threading [50], contiguous sequences from the dynamic instruction stream of a program – called speculative threads – are distributed across processing elements within a processor or processor cores within a chip-multi-processor. This approach is effective, if data dependence between threads is minimal and there is no cache miss. If not, this approach fails to achieve any benefit.

There are non-speculative techniques such as decoupled access/execute architecture and various cache designs. Decoupled architecture [49] separates accesses to memory to fetch operands and store results from operand execution to produce results. The communication between two parts is performed by using a buffer. High performance can be obtained by exploiting the fine-grain parallelism between the two and it effectively hides all memory latency from the processor perspective.

Various cache designs have been proposed to improve cache performance. Victim cache [54] is a buffer to hold discarded blocks from L1 cache. Processor core can access victim cache in parallel with L1 cache, and if a requested block is found in the victim cache, the block is moved into L1 cache. This plays a role of an extra associativity to each set in L1 cache, even though the size of it is typically much smaller than the size of a way.

A non-blocking cache [8] allows execution to proceed with one or more cache misses until an instruction that depends on a pending miss is reached. In order to accommodate multiple pending misses, miss status holding registers were introduced. By allowing multiple pending misses, multiple memory accesses are overlapped, exploiting memory level parallelism.

Cache-conscious structure allocation [10] is a technique to place data structures in memory in a manner that increases a program's reference locality and improves cache performance. This technique clusters temporally related objects into the same cache block or into non-conflicting blocks. It works well with structures smaller than a cache block. For structures bigger than a cache block,

cache-conscious structure definition [9] was proposed. If a structure is about the same size of a cache block, splitting the structure is employed. It splits a structure into a hot and cold portion and put the hot portion into a cache block. For a structure much bigger than a cache block, field reordering is applied. It reorders structure fields to place those with high temporal affinity in the same cache block.

More complex yet more effective replacement policies than conventional least recently used (LRU) policy have been proposed. Among them, shepherd cache [38] tries to emulate optimal replacement. The optimal candidate for replacement is the one accessed farthest in the future. Due to temporal locality abundant in L1 cache, LRU policy can approximate this optimal replacement fairly well. Hence LRU replacement policy is the obvious choice for L1 cache. However, as only memory accesses that miss at L1 cache go to L2 cache, the temporal locality is filtered out. Consequently, LRU replacement policy is not effective for L2 cache. Shepherd cache provides a lookahead window for main cache, as optimal replacement requires a lookahead into future accesses to be able to make an optimal decision.

2.2. Hardware prefetching schemes

Hardware prefetching schemes can be classified according to how to generate prefetch addresses.

The simplest way of prefetching is prefetching the next cache block of the block that causes a cache miss. This is based on spatial locality of data access. If one block is accessed, the next block will be accessed in the near future. Stream buffer [35] utilizes this locality. The stream buffers follow multiple streams and each of them are prefetched in parallel. They are designed as FIFO buffers that hold consecutive cache blocks, starting with the one that missed in the L1 cache. On subsequent misses, the head of the stream buffer is looked up. If the head matches the requested block, the block

is transferred to L1 cache.

Another simple but effective way of prefetching is stride-based prefetching scheme. This scheme allows prefetcher to eliminate cache misses that follow a regular stride pattern. This type of accesses is frequently found in applications that manipulate arrays. Palacharla and Kessler [34] proposed a non-unit stride detection mechanism to improve stream buffer. In this scheme, stride is determined as the minimum signed difference between the past N miss addresses.

Whereas global miss address stream is used in Palacharla's work [35], Farkas et al. [13] proposed a PC-based stride prefetching scheme. They looked at each load's access stream and determine a stride for each load.

More recent work on stride-based prefetcher was presented by Hariprakash et al. [15]. They look at L1 miss address stream to detect strides. Their motivation for using global miss stream is that PC-based stride prediction schemes can only identify strides that are enclosed in loop. They compare each recent miss address with several previous miss addresses to calculate all possible strides. Among them, steady strides are selected to compute future addresses. L1 cache miss or prefetch buffer hit triggers prefetching.

The most commonly used prefetching scheme along with stride-based prefetching scheme is correlation-based prefetching scheme [7, 16, 18, 24, 33, 34, 47, 51]. Correlation-based prefetchers are based on address streams from processor or L1/L2 cache, and correlate addresses with their preceding addresses. If a sequence of addresses repeats, correlation-based prefetchers are able to predict future addresses effectively.

Charney and Reeves [7] first proposed a correlation prefetching scheme that took L1 miss address stream as input. They tried to pair two addresses not close to each other, so as to improve prefetch lead time at the expense of lower prefetch accuracy. They also showed that stride-based prefetching could be combined with correlation-based prefetching to improve prefetch coverage.

Markov prefetcher [18] was proposed by Joseph and Grunwald. They found that the miss address stream can be approximated by a Markov model. They correlate multiple addresses with an address in order to handle the case where a given miss address may be followed by one of several different addresses depending on the control flow.

Bekerman et al. [4] presented an efficient correlation prefetcher table structure that reduces space requirement. They observed that there existed some global correlation among static load instructions, i.e. address sequences belonging to different static loads, which only differ by some constant offset. This correlation exists among loads associated to different fields of an LDS. Since all loads associated with the same LDS share the same set of base addresses, they recorded base addresses instead of real addresses. There are two tables in their prefetcher: load buffer and link table. Each entry in load buffer records the history of recent addresses exhibited by the associated load. This history is used to index into link table that stores predicted addresses. Since a link table entry can be shared by multiple entries in load buffer due to global correlation, overall prefetcher size can be reduced.

Sherwood et al. [47] proposed an enhanced version of stream buffer called predictor-directed stream buffer. One drawback of stream buffer is its limited applicability. It works well only with stride intensive applications. Instead of prefetching sequential blocks, they use a hybrid stride filtered Markov predictor to generate the next address to prefetch. They claim that any address predictor can be used to guide the predicted prefetch stream.

Lai et al. [24] published another correlation-based prefetcher, dead-block prediction and dead-block correlating prefetchers. This prefetcher predicts two things: whether current cache line is dead and what data will replace the current line. In other words, it predicts both “when” and “which”, while most of the prefetchers predict only “what”. Predicting when a cache line is dead enables placing prefetched data directly into L1 cache. Unlike miss correlating prefetchers, this prefetcher

primarily capitalize on repetitive instruction sequences rather than memory address sequences to predict memory access behavior. This prefetcher produces timely prefetches because the time during which a cache line is dead is quite long and maybe longer than the time required for prefetching a cache block from memory.

Later, Hu et al. [16] proposed tag correlating prefetchers. It was based on the observation that per-set tag sequences exhibit highly repetitive patterns both within a set and across different sets. Because single tag sequence can represent multiple address sequences spread over different cache sets, significant space saving can be achieved. In addition to the size reduction, this prefetcher improves prefetch timeliness, because the time interval between consecutive misses in a cache set is typically larger than the memory access latency, providing sufficient time for a timely prefetching.

Sharma et al. [46] borrowed the idea of partitioning memory into zones from the tag correlating prefetcher. They examined the correlation of the tags and strides in each zone and used them to make prefetches. The main contribution of the paper is the filtering mechanism that filters out the random noise from the miss stream based on frequency of individual pattern. Also, they considered not only absolute values (addresses) but also differential values (stride). By using differential values, the prefetcher became more efficient in terms of performance and space overhead.

Nesbit et al. [33] proposed a prefetching mechanism using a global history buffer. Prefetching with global history buffer has two significant advantages over conventional table-based prefetching mechanisms. First, the use of a FIFO history buffer can improve the accuracy of prefetching by eliminating stale data from the table. Second, the global history buffer contains a complete cache miss history, enabling more effective prefetching mechanisms. History information in the global history buffer is maintained in linked lists. This prefetching mechanisms targets L2 cache misses because global history buffer needs to be large in order to store L1 cache misses. They tried several combinations of different prefetching mechanisms and pattern detecting mechanism. It

is shown that this approach can improve performance with relatively small space overhead.

Recently, Somogyi et al. [51] introduced a prefetching mechanism called spatial memory streaming. They observed that memory access in commercial workloads often exhibit repetitive layouts that span large memory regions (several KB), and these accesses recur in patterns predictable by code-based correlation. The relationships among accesses in a region are called spatial correlation. Although accesses within these structures may be non-contiguous, they exhibit recurring patterns in relative addresses. There are two reasons for spatial correlation. First, spatial correlation exists because of repetition and regularity in the layout and access patterns of data structures. In this case, the spatial pattern correlates to the address of the trigger access, because the address identifies the data structure. Second, spatial correlation can also form because a data structure traversal recurs or has a regular structure. In this case, the spatial pattern will correlate to the code responsible for the traversal. They used a bit vector to encode spatial correlations.

Most recently, Chou [12] proposed low-cost epoch-based correlation prefetching for commercial applications. It targets on-line transaction processing, which have irregular control flow and complex data access patterns. This nature of the applications makes low-cost prefetchers such as stream buffer and stride-based prefetcher ineffective. Correlation-based prefetching has potential but it generally requires a large table that is impractical to implement on chip. Chou proposed two things. First, he proposed to store epoch-based correlation table in main memory. Second, he exploited the concept of epoch to hide the long latency of its correlation table access. He observed that program execution is a recurring periods (epoch) of on-chip computations followed by off-chip accesses. It was also observed that eliminating only part of overlapped misses couldn't improve performance much. In order to achieve the full potential of prefetching, all overlapped misses should be removed altogether. Hence all addresses in an epoch are lumped and prefetched together. When a miss occurs in an epoch, all addresses involved in the next two epochs are prefetched. This way, accessing

latency to main memory can be hidden.

2.3. Software prefetching schemes

Software prefetching schemes usually insert something in the code to initiate prefetching. Many of the schemes insert explicit prefetch instructions, but there is a scheme that modifies object structure in order to add additional information such as jump pointers in the code. Software-hardware cooperating schemes often insert prefetch hints in the code instead of prefetch instructions. Based on the hints, hardware prefetcher generates prefetch requests.

Because loops in scientific applications are highly predictable and easy to analyze in compile time, prefetch instructions can be automatically inserted in such loops. Mowry et al. [29] were the first to propose a compiler algorithm that automatically places prefetch instructions in the code.

In order to gather information usually not available at compile time, such as loads with stride pattern in irregular programs, a new profiling method was proposed by Wu [57]. It is based on the observation that some pointer-chasing references exhibit stride patterns. This stride profiling is integrated into the traditional frequency-profiling pass and the combined profiling pass is only slightly slower than the frequency profiling alone. This profile information guides compiler to insert prefetch instructions.

Instead of inserting specific prefetch instructions, Rabbah et al. [38] introduced inserting a small code segment called load dependence chain. The load dependence chain represents a subset of the program dependence graph pertinent to the address calculation of a specific delinquent load. This approach is called program embedded precomputation via speculative execution. It utilizes any unused instruction slots in Itanium processor to embed speculative address precomputation within program regions that are likely to benefit from prefetching.

There are many software and hardware hybrid prefetchers. Hybrid approach tries to combine the advantages of both prefetching styles. Software prefetching moves the space overhead of hardware prefetching to program code (main memory), and improves the prefetching accuracy. Hardware prefetching reduces the overhead inevitable with software prefetching, and is able to gather and leverage runtime information.

Jump pointer prefetching [44] can be implemented as a pure software mechanism or a hybrid mechanism. Software version inserts additional pointers in the LDS to connect non-adjacent elements in the LDS. These jump pointers allow prefetch instructions to name link elements further down the pointer chain without sequentially traversing the intermediate links. Because jump pointer based schemes create memory level parallelism, they tolerate cache misses even when the traversed loops do not have sufficient work to hide the miss latencies. However, jump pointer installation code increases execution time, and the jump pointers themselves incur additional cache misses. In order to reduce these side effects, modest hardware support is introduced. This hardware is the previously proposed dependence-based prefetching mechanism [43]. This mechanism is responsible for generating chain prefetches. Consequently, the number of artificially created jump pointers embedded in LDS can be significantly reduced. This is a type of integration that hardware mechanism helps software mechanism.

Some hybrid prefetchers [17, 56] have the opposite form: a software mechanism that helps the hardware mechanism. Software provides vital information such as dynamic data structure and offsets to pointers in LDS via instructions. Such instruction may be a new instruction specifically designed for this purpose, or an unused instruction, in which a prefetch hint is encoded.

Chapter 3

Timely Dependence-based Prefetching

3.1. Motivation

Dependence-based prefetching (DBP) [43] identifies the producer-consumer relationship between loads and uses it to prefetch data. When a producer's value is loaded, it is then used to issue prefetches for consumers. There are several advantages of DBP. First, it doesn't require knowledge of past access patterns of consumer loads, eliminating the need for a big correlation table. Once the dependence relation is learned, prefetches can be made. Second, it has good coverage. Most of the loads in the codes that manipulate linked data structures (LDS) can be covered by DBP. Third, it has great accuracy because no prediction is involved. However, this scheme suffers from a severe shortcoming: timeliness. DBP works only when there is enough work between producer and consumer. There is no guarantee that prefetches are issued early enough for data to be available to the processor before it is accessed. Prefetch timing in the schemes depends on available work between producer and consumer.

To resolve this problem, Roth et al. [44] utilize a *jump pointer* on top of dependence-based prefetching. Jump pointers refer to non-adjacent nodes in an LDS, so the prefetch engine can run

ahead of the processor, resulting in timely prefetches. They introduce a way to capture jump pointers by storing past effective addresses of a recurrent load in a queue. Because they advocated software and hybrid approaches using main memory to store jump pointer storage, they didn't elaborate on purely hardware implementation. Pure hardware implementation may be expensive, but it doesn't require any changes in instruction set and memory allocator interface. Roth et al. also did not analyze the relationship between jump pointer accuracy and the distance between the home and target nodes of a jump pointer.

Collins et al. [12] published similar work, using pointer cache. They first identify pointer loads and record these pointers (their effective addresses and loaded values) in the pointer cache. Each pointer is actually a jump pointer with zero distance. They use the pointer cache for value prediction, prefetching, and speculative precomputation. This work has the same problem as dependence-based prefetching. Associating an address of a pointer load with its loaded value doesn't improve prefetch engine or speculative thread's progress. If prefetch engine or speculative thread can hide one miss latency for main thread by a successful prefetch, the main thread can easily catch up because prefetch engine or speculative thread experiences miss latency but main thread does not.

There are a couple of potentials other than the advantages mentioned above that make revising DBP worthwhile. First, it is easy to combine different prefetching schemes on top of DBP. For example, adding a stride prefetcher to a jump pointer prefetcher [44] can be done seamlessly without elevating complexity. By doing so, the prefetching scheme becomes more general because it can work well with more applications. Second, the number of correlations or jump pointers stored in a table can be significantly reduced by leveraging chain prefetching, which is based on dependence information identified by DBP.

To make the most of the newly found potentials, we propose a dependence-based prefetch mechanism that mitigates the prefetch timing issue by exploiting the memory access patterns of

producer loads. This is similar to Roth's work [43], but we use dependence-based prefetching as a framework, on which multiple prefetching schemes such as stride prefetching, jump pointer prefetching, or more general correlation-based prefetching can be integrated. These prefetching schemes are employed to make the prefetch engine progress faster than the main processor. To reduce the space overhead of a big table such as a jump pointer table or a correlation table in pure hardware implementation, a chain prefetch mechanism that prefetches multiple blocks without stored jump pointers or correlations is introduced.

3.2. Dependence-based prefetching mechanism – high level view

Dependence-based prefetching [43] dynamically extracts program segments responsible for generating addresses of linked data structures. It then speculatively executes the segments alongside the original program. These speculatively executed instructions result in data prefetching.

In order to do so, producers need to be first identified for the target loads (consumer loads). These producer-consumer chains constitute the program segment mentioned above. Only a direct relation between producers and consumers is considered. This means that a loaded value of a producer is the base effective address of the corresponding consumer. The hardware structure for storing potential producers (recently committed loads) is called the potential producer window (PPW), shown in Figure 3.1.

Once producer-consumer pairs are identified, consumers' offsets are also needed to generate precise prefetch addresses. As soon as a producer's value is available, the value is added to the corresponding consumer's offset to form a correct prefetch address. Dependence information such as program counters (PC) of producer-consumer pairs and consumers' offsets are stored in the correlation table (CT), as shown in Figure 3.1.

Whenever a load is committed, the PPW is searched for the address of the committed load. PPW holds information about recently-committed loads such as PC and loaded value. If a match is found, it means a producer generating the address for the committed load is identified. Once a match is found, this producer-consumer pair is stored in the CT. The information stored in CT includes PCs of producers and consumers and offsets of consumers.

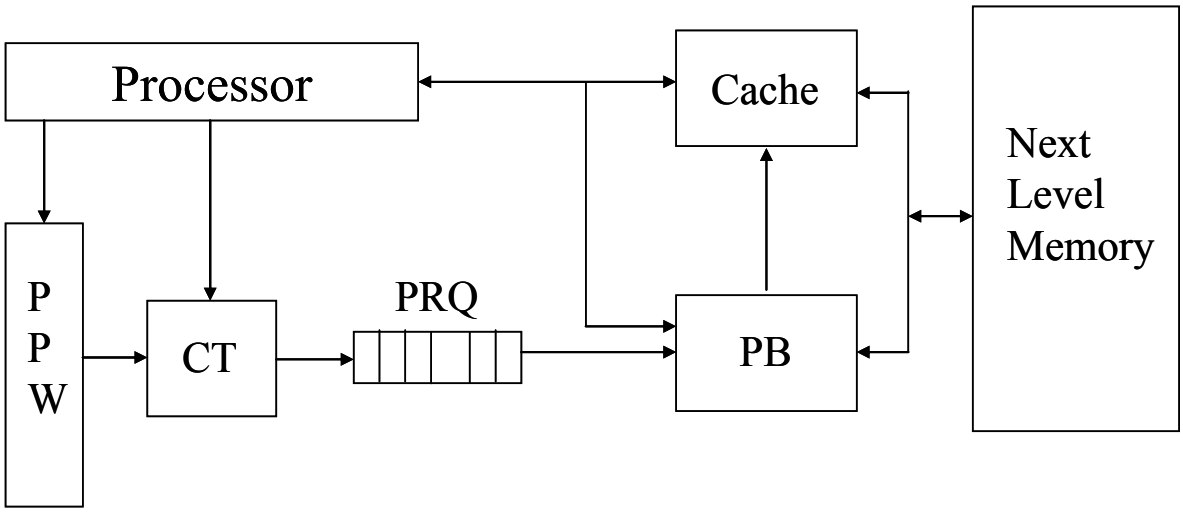


Figure 3.1. Schematic of DBP [43]

In order to issue prefetch requests, CT is looked up by committed loads. If a committed load is found to be a producer for one or more consumers, the loaded value of the committed load is used to computing a prefetch address for each consumer load by adding the offset of each consumer to the loaded value. These generated prefetch requests are queued in the prefetch request queue (PRQ). Depending on the availability of resources such as cache read ports, the requests in the PRQ are serviced in order, and prefetched data is stored in the prefetch buffer (PB).

Though not explicitly shown in Figure 3.1, chain prefetching is possible with minor modification. To support chain prefetching, PB maintains a list of requesting consumers with each

block. When a prefetch block arrives, each consumer on the list plays the role of a producer, looking up CT to generate chain prefetch requests.

3.3. Proposed dependence-based prefetching mechanism – high level view

In this section, we provide a high level view of the proposed prefetcher: timely dependence-based prefetcher. The key differences with existing mechanisms [43, 44] are highlighted and the proposed prefetcher is introduced in high level by explaining the differences.

3.3.1. Key differences with existing mechanisms

The key differences with the original dependence-based prefetcher [43] are the mechanisms that enhance timeliness of the prefetcher. With these additional mechanisms, complexity and space overhead increase but higher performance gain is achieved by converting partial prefetch hits to full prefetch hits.

There are four key differences with the jump pointer prefetcher [44]. The first difference is the way DBP is combined with other prefetchers. In the jump pointer prefetcher, DBP is used only for generating chain prefetches. However, in the proposed prefetcher, DBP plays a more important role as a general framework, on which different prefetchers are combined. In addition to supporting chain prefetching, it learns producers' memory access patterns dynamically, and different prefetchers are assigned to producers according to their patterns. In other words, DBP is capable of classifying producers' memory access patterns, and different prefetchers are combined in such a way that they work for their own set of producers increasing the effectiveness.

The second key difference is how jump pointers are created. In the original jump pointer

prefetcher, there are four prefetching idioms: queue jumping, full jumping, chain jumping, and root jumping. For the hardware implementation, chain jumping is selected in order to decrease the number of required jump pointers, while maintaining prefetching accuracy. With chain jumping idiom, jump pointers are created for all recurrent loads (backbone), and all other loads (rib) are prefetched via chain prefetching. The proposed jump pointer population method leverages chain prefetching even for recurrent loads. The new method will be described in section 3.3.2.

The third difference is the filtering mechanism that eliminates infrequently missed loads. By eliminating such loads, PPW and CT are used only by frequently missed loads improving efficiency of the structures. This filtering mechanism also serves as a mean to combine a stride prefetcher. More detailed description of this mechanism will be presented in section 3.3.3.

The fourth key difference is how to implement the prefetcher. Roth et al. [43] advocated software or hybrid approach to implement jump pointer prefetcher. The advantages of software implementation over hardware implementation are no space overhead and higher prefetch accuracy. On the other hand, there are four disadvantages. First, modifying programs requires human participation. Profiling for identifying loads that manipulate LDS and inserting prefetch instructions are done by human. Therefore, it is hard for software developers to use this approach, and it is even more unlikely for existing programs to get recompiled to incorporate this jump pointer prefetching capability. Second, jump pointer storage consumes user memory and increases the program's data footprint. Third, jump pointer maintenance and prefetching code increases both static program size and dynamic instruction count. Fourth, chain prefetches implemented by software introduce serialization artifacts into the program, clogging the pipeline. To mitigate this serialization problem, hybrid approach is devised. It introduces modest hardware support to allow chain prefetching to be implemented in hardware. The hardware component of the hybrid approach is nothing more than the original DBP.

All the advantages of software approach become the disadvantages of hardware approach, and vice versa. Chapter 5 presents a way to alleviate the space overhead by utilizing a portion of the level 2 cache.

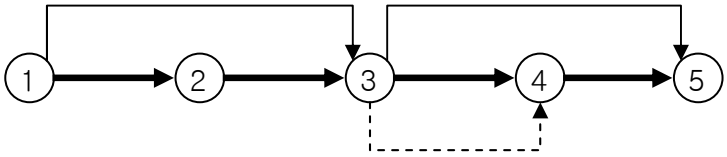
3.3.2. Jump pointer creation method

Jump pointer prefetching [44] targets LDS, and it usually requires a large jump pointer table. Roth et al. present four prefetching idioms, each of which has different space overhead. The difference between idioms is how jump pointer prefetching and chain prefetching are combined. Queue jumping idiom creates jump pointers only for recurrent loads (backbone) and chain prefetching is not used. This is suitable for a simple structure, such as a list. Full jumping idiom creates jump pointers for all loads involved in LDS manipulation. Since loads in LDS manipulation are prefetched using jump pointers, chain prefetching is not used at all. Chain jumping idiom creates jump pointer for recurrent loads (backbone) and other loads in LDS manipulation are prefetched by chain prefetching. Root jumping idiom exclusively relies on chain prefetching.

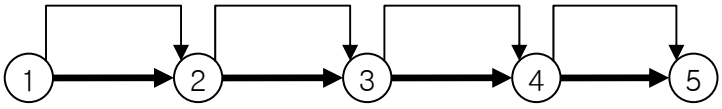
In terms of space overhead, root jumping is the best and full jumping is the worst. In hardware implementation, prefetch accuracy is unavoidably lower than that of software implementation. Chaining prefetching tends to make prefetch accuracy even lower, because one mispredicted prefetch address may result in multiple inaccurate prefetches. Therefore, in hardware implementation, it is important to balance these two requirements: space overhead and prefetch accuracy.

Proposed jump pointer creation method lowers the space overhead of chain jumping by creating jump pointers for only a subset of recurrent loads. All other recurrent loads are prefetched by using chain prefetching, as well as other non-recurrent loads in LDS.

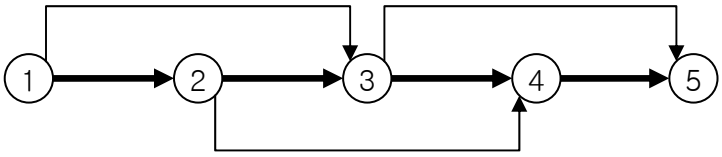
Figure 3.2 shows different jump pointer creation algorithms. A circle represents a dynamic instance of a recurrent load (pointer load). Bold arrows indicate pointer dereferences and all other arrows show prefetches. If it is a dashed line, it means that it doesn't need a pointer stored in a table and that prefetches can be issued after current instance's value is loaded. If it is a solid line, it means that it needs a pointer stored in a table and that prefetches can be issued after current instance's address is available.



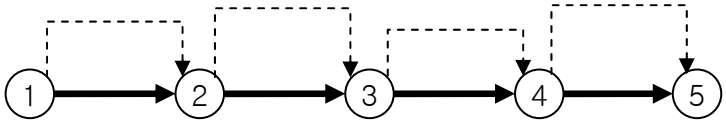
(a) proposed prefetch scheme



(b) pointer cache [12]



(c) chain jumping [44]



(d) dependence-based scheme [43]

Figure 3.2. Prefetching schemes for LDS

A circle represents a dynamic instance of a recurrent load (pointer load). Bold arrows indicate pointer dereferences and all other arrows show prefetches. If it is a dashed line, it means that it doesn't need a pointer stored in a table and that prefetches can be issued after current instance's value is loaded. If it is a solid line, it means that it needs a pointer stored in a table and that prefetches can be issued after current instance's address is available.

Let's look at the diagram in terms of storage requirement and timeliness of prefetches for prefetching nodes 3, 4, and 5. Assume that nodes 3, 4, and 5 incur cache misses.

Pointer cache [12] requires space for three pointers, and prefetches may not be able to fully eliminate the cache misses if there are not enough independent instructions to hide prefetch latencies. Jump pointer [44] also requires three pointers, but prefetches are issued earlier. When node 1's address is available, prefetch for node 3 is issued. In this way, two different accesses are overlapped. With the aid of jump pointers, it is more probable that all misses are eliminated. Dependence-based prefetching [43] requires no space for pointers. Even though it requires space for dependence checking and storing, this is generally smaller than the space for jump pointers. Although this prefetcher is the least demanding in terms of space, it has the worst prefetch timing. Since prefetch for node 3 is issued after node 2's value is available, there is not much chance for the cache miss of node 3 to be completely hidden.

The proposed prefetcher requires space for only two pointers. Nodes 3 and 5 are prefetched in the same way they are prefetched by jump pointers, but node 4 is prefetched differently. Prefetching node 4 doesn't require a jump pointer; instead it is prefetched by a chain prefetching mechanism. When prefetched data for node 3 comes in, prefetch for node 4 is triggered. The address for node 4 is computed from the prefetched data for node 3.

In this example, jump pointers skip just one node, but they can skip more nodes, making it more probable that targeted cache misses are eliminated. This may result in lower accuracy.

3.3.3. Filtering mechanism

The filtering mechanism is placed between the processor and the CT in Figure 3.1. In figure 3.3, it is in the correlation identification unit (CIU). A table plays a role of the filtering mechanism, and it is called frequently missed load table (FMLT). Its main job is filtering out infrequently missing loads, so that the rest of the prefetching mechanism can deal with only frequently missing loads. This mechanism uses a simple counter, in which the number of misses of each static load is recorded. When a counter value is bigger than a predefined threshold, it is identified as a frequently missing load (FML) and sent to PPW to find its producer in the list of preceding loads. The filtering mechanism resets the entire table if a predefined number of FMLs have been identified since the last reset. In this way, only the most frequently missing loads can be identified as FMLs, enabling efficient utilization of limited resources placed after FMLT. There are two thresholds in this mechanism: the number of FMLs that triggers FMLT reset, and the number of misses that draws a line between frequently missing loads and infrequently missing loads. Since each application has its own threshold values that work the best for it, dynamically adjusted thresholds may be necessary. This is left for future research.

The secondary job of the FMLT is filtering out loads with stride patterns. Such loads are direcetly inserted into CT and marked as regular strided loads. This is how the regular stride prefetching is implemented because the loads with stride patterns generally do not have producer loads. The benefit of this mechanism is twofold: to combine stride prefetching with dependence-based prefetching seamlessly and to lessen the burden on CIU. It is a good idea to apply complex prefetching algorithm only to loads or addresses that cannot be prefetched by a stride prefetcher. This is because stride prefetching is very efficient in terms of space requirement. If a stride for a static load is known, its future addresses can be accurately predicted even though the predicted

addresses have not yet been seen.

For dependence-based prefetching scheme, there needs to be a way to combine it with stride prefetching. Since only a load can be a producer, direct-dependence-based prefetching is not able to perform stride prefetching as is. In order to implement the stride filtering mechanism, FMLT has stride field, previous address field, and stride confidence field. Stride field is for storing previous stride and is compared with current stride computed from current address and previous address field. If the current stride is the same as the previous stride, stride confidence is incremented. Otherwise, stride confidence is decremented. If a load is identified as FML and its stride confidence is high enough, instead of being sent to PPW, it is inserted into CT. To indicate such a stride pattern, there is a field for a consumer called 'stride'. If this is set to one, it means the consumer is for stride prefetching.

3.4. Implementation

In this section, hardware implementation details are presented. The proposed prefetcher is composed of four parts: correlation identification unit (CIU), correlation table (CT), jump pointer creation unit (JPCU), and jump pointer table (JPT). The interactions of these components with themselves, with a processor, and with the memory hierarchy are shown in Figure 3.3.

CIU receives committed loads' information, such as effective address, loaded value, and offset, and produces entries for CT. Each entry has producer's information (PC, effective address, and loaded value) and consumer's information (PC and offset). Since entries of CT are directly responsible for prefetches, it is important that every entry in CT must represent stable producer-consumer relationships. Once an entry is inserted into CT, its producer load is then updated by the loads committed from the processor. Through the update operations, patterns associated with every

producer load are dynamically learned by means of incrementing or decrementing counters. When CT is looked up by the processor, these counters are read and an appropriate action is taken. CT is looked up by the processor, whenever a load's effective address is available. If a stride is detected in a producer's effective address, a prefetch request is queued in the prefetch request queue (PRQ). If a producer is confirmed as a recurrent load, JPT entries are generated by JPCU and stored in JPT. Every prefetch request is queued in PRQ and sent to L2 cache. When a prefetched block is returned from L2 cache, it is stored in the prefetch buffer. This buffer is accessed by the processor in parallel with L1 cache.

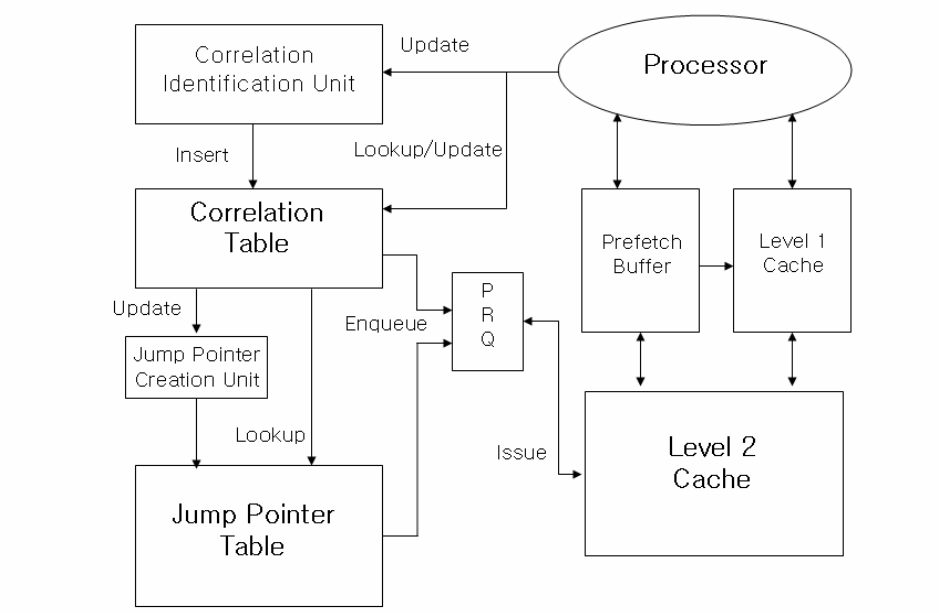


Figure 3.3. Organization of the Proposed Prefetcher

In the following sub-sections, each unit is described by explaining the three basic actions: inserting into CT, updating CT, looking up CT.

3.4.1. Inserting into CT

CIU is responsible for generating CT entries and inserting them in CT. The entries to be inserted into CT must have the following two properties. First, the consumers of the entries should miss frequently in the L1 cache, making the processor stall. Second, consumer-producer relations must be strong. CIU is made up of three tables: frequently missed load table (FMLT), potential producer window (PPW), and candidate pair table (CPT).

FMLT is in charge of pinpointing static load instructions that miss in L1 cache frequently. FMLT is updated by committed loads from the processor, and keeps track of the number of misses per static load instruction. If the number of misses reaches a predefined threshold, it resets the counter and sends a signal to PPW. PPW holds a list of recently committed load instructions' PCs and loaded values in order to facilitate finding producers. The signal from FMLT activates the pairing logic in PPW that tries to find a matching producer for a given effective address from the processor. When the signal is high, both PPW update and pairing are conducted, but when the signal is low, only updating PPW is performed. If a producer-consumer pair is formed, it is then sent to CPT. CPT records the number of times a specific pair has been identified. Only pairs that have strong correlations between their producer and consumer are inserted into CT. FMLT and CPT also reduce the number of CT accesses, reducing conflicts on the write port of CT.

Figure 3.4 shows the three tables. Figure 3.4(a) is the information of a committed load from the processor. As shown in Figure 3.4(b), the load accesses FMLT and increments the corresponding counter to 127, which is the predefined threshold. Then FMLT signals to PPW to activate producer searching. Figure 3.4(c) shows two entries of PPW, one that is just updated by the committed load and the other one that is found to be a producer of the committed load. Since the producer is found, this producer-consumer pair is sent to CPT. CPT already has the pair, and its counter is incremented

to 7, which is the threshold for inserting into CT, as depicted in Figure 3.4(d). This series of actions describes the mechanism of inserting an entry into CT.

Integrating a plain stride prefetcher into CT can be easily done. Even though the producer is not found for a consumer, an entry is created in CT with its PC as the producer's PC. The entry is updated by subsequent instances of the consumer; if the consumer has stride pattern in its address stream, it will be learned and prefetches will be issued based on it.

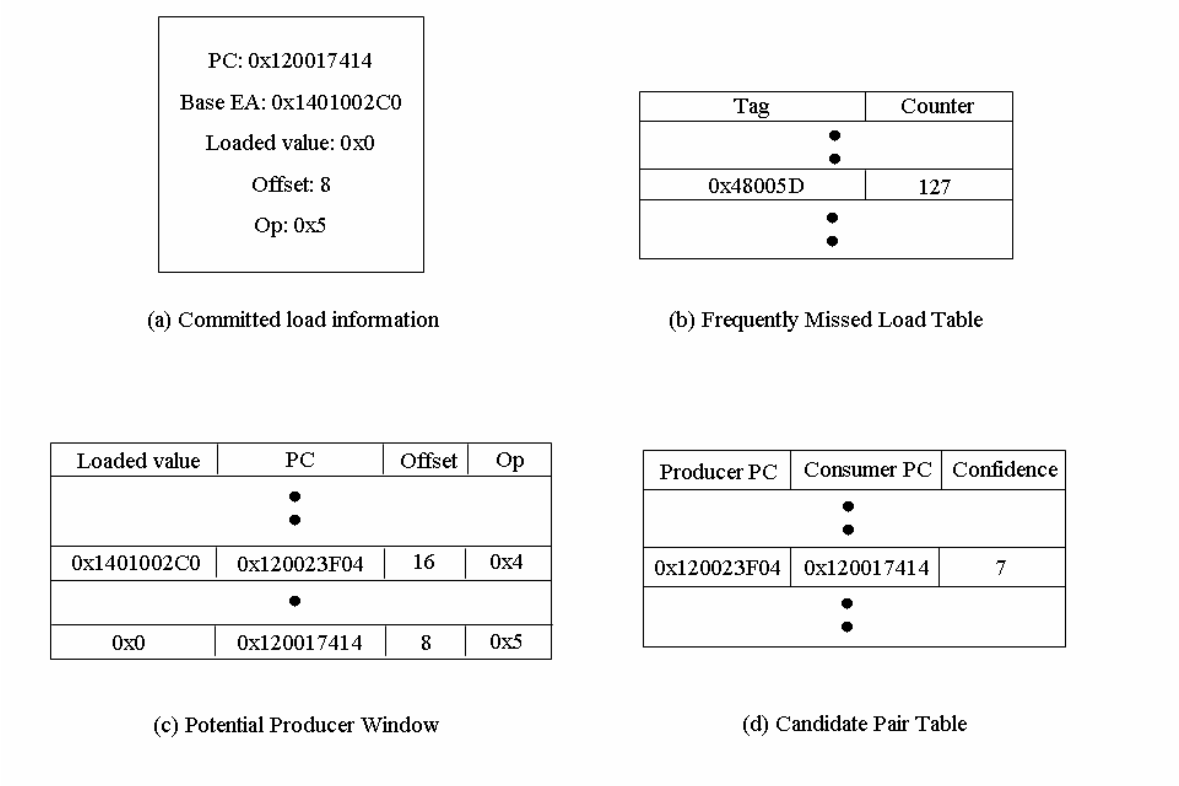


Figure 3.4. Example of Correlation Identification Unit

3.4.2. Updating CT

Once an entry is created in CT, the entry is updated by committed load instructions. Because we want to find a pattern for each producer, only the producer portion of the entry is updated. Once a

pattern, either stride or recurrent, is identified for a producer load, this pattern is utilized to make the prefetch engine run ahead of the processor, making it more probable that prefetched data arrive before they are accessed.

If the updating load experienced an L1 cache miss, consumer loads are searched against its PC. Searching all consumer loads may take many cycles, but JPT update occurs infrequently. For pointer-intensive benchmarks such as health and mcf, JPT update occurs once per 22 cycles for mcf, and once per 100 cycles for health.

If a match is found and corresponding producer has been identified as a recurrent load, JPT can be updated. CT sends the PC of the corresponding producer to JPCU, and JPCU sends two addresses to JPT. In JPCU, there are 16 FIFO queues, each of which corresponds to a load that has been identified as a recurrent load. These queues are used to record effective addresses and loaded values of up to eight most recently committed instances – in other words, for each recurrent load, up to eight most recently committed instances are stored. For a given PC, JPCU finds the right queue that is assigned to the PC, gets two addresses from the queue, and sends them to JPT. Jump pointer distance is a predefined constant, used for picking up the two addresses. Let's assume the distance is set to i. The first address is the loaded value of the most recent element of the queue, and the second address is the address of the i-th element from the tail. The first address is to be stored in JPT, and the second address is used to get an index to JPT. Figure 3.5 illustrates this mechanism. The address of the upper entry (0x1401017D8) of the queue is used for indexing into JPT, and the loaded value of the most recent entry, which is the lower entry (0x1402D87F0) of the queue in the figure, is written to JPT.

When the second address is accessed again, the first address is prefetched if the corresponding confidence is high enough. The first address is the loaded value of the most recent instance of the producer, so that if a miss occurs for a consumer that is just committed, the missed

address can be computed by adding the consumer's offset to the first address. Therefore, prefetching the address computed from the first address can eliminate the cache miss of the consumer.

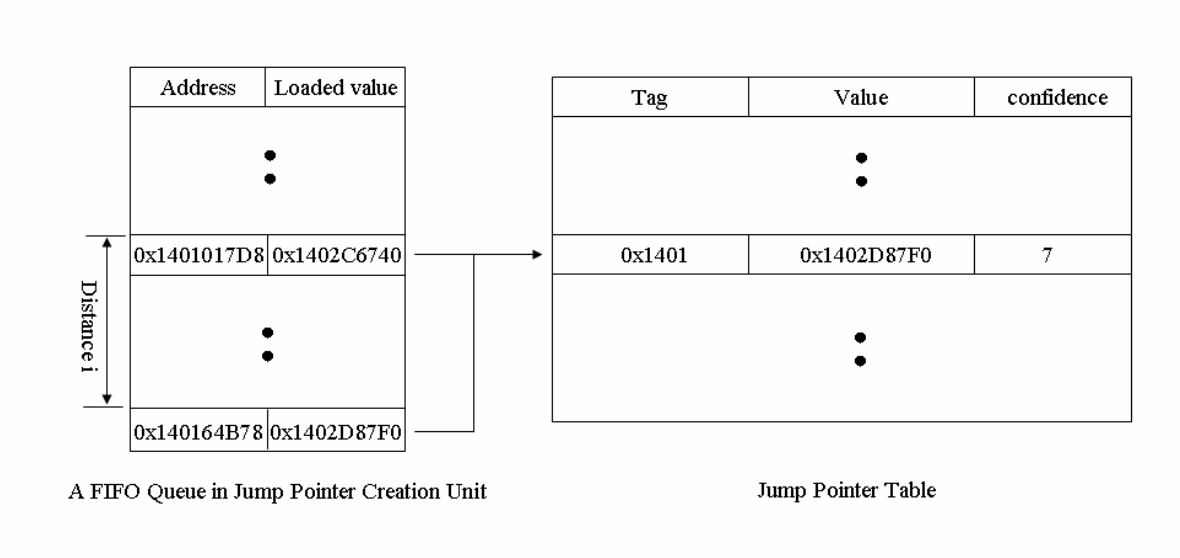


Figure 3.5. Example of Jump Pointer Creation Unit and Jump Pointer Table

3.4.3. Looking up CT

CT plays two roles. It detects producers' patterns, and it works as storage of producer-consumer pairs. One producer can have multiple consumers, which is common in producer-consumer relationships. Figure 3.6 shows two entries of a CT. Counter1 represents the confidence of the stride pattern, and counter2 represents the confidence of the recurrent pattern.

When their effective addresses are ready, load instructions access CT. A load's PC is used to index into CT, and if the indexed entry has a stable pattern, a prefetch request is issued and stored in PRQ. For a stride pattern, the future address is computed by first multiplying the stride by a small constant, and adding the result to the current address. For a recurrent pattern, JPT is accessed with the current address to get a future address.

The upper entry in Figure 3.6 has a stable stride pattern, so a prefetch address is computed from the stride and consumer's offset. The lower entry has a recurrent pattern, so the current effective address will be used to index into JPT. The address returned from JPT is added to a consumer's offset to form a prefetch address.

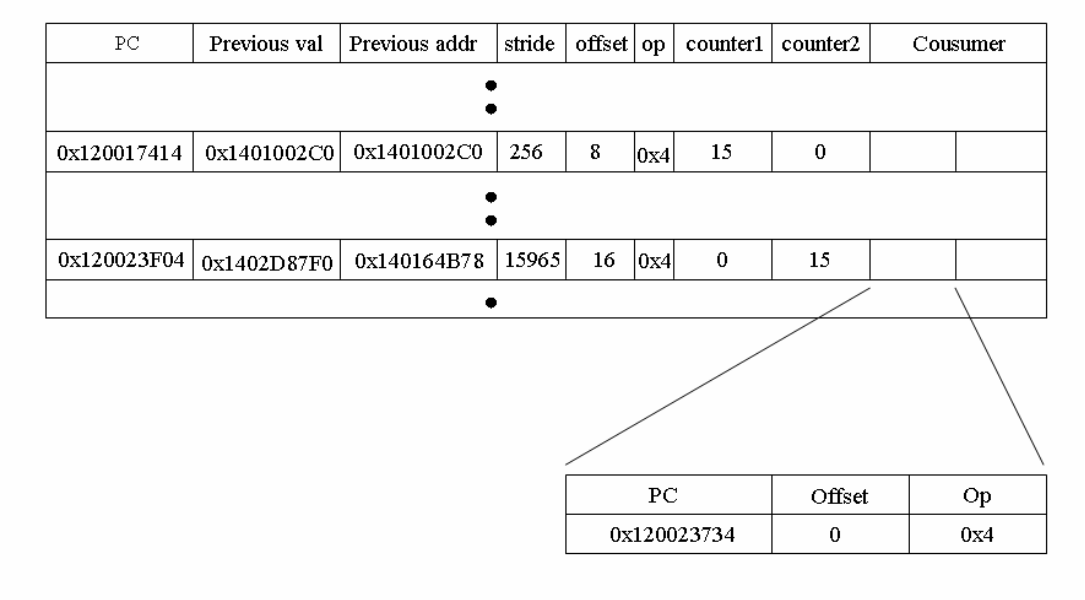


Figure 3.6. Example of Correlation Table

For chain prefetching, CT is looked up too. When a prefetch request is queued in PRQ, corresponding CT entry's index is stored. When the prefetched data arrives, CT is accessed again with the index. Then, prefetch addresses are computed by adding the corresponding consumers' offsets to the prefetched value, because the value is directly related to the effective addresses of consumers.

3.4.4. An alternate CT structure

The structure of the CT presented earlier is ineffective for storing consumers per producer. From experiments, it is found that only a few producers have more than two consumers, and this finding suggests that allowing a producer to have more than two consumers is wasting precious silicon estate. However, it should be noted that those producers with more than two consumers often contribute to performance improvement and that to harvest the full potential of them, all of their consumers should be stored. Therefore, all consumers should be stored in more effective way.

One such way is to divide the CT into two tables: producer table and consumer table. In this way, searching the consumers for creating jump pointers can make more efficient. In the producer table, each producer has a set of indexes to the consumer table. Each index points to a consumer of the corresponding producer. Since an index is much smaller than a consumer's information, this approach can reduce space overhead. Each entry in the consumer table also has an index to the producer table. This index points to the corresponding producer. This facilitates jump pointer creation.

When an entry is created in the CT, indexes to the both tables are first determines and the corresponding entries are updated. Indexes to the tables are also stored. When the CT is updated, the producer and the consumer tables are searched in parallel. If a match is found in the producer table, the producer entry is updated. If matches are found in the consumer table, jump pointers are created. When the CT is looked up for generating prefetches, only the producer table is probed. If a match is found, the stored indexes in the producer entry are used to access its consumers.

Looking up the CT is more critical than updating the CT in terms of timing. This is because updated entries are not likely to be used in the near future, and looking up the CT is related to generating prefetch requests. If prefetch requests are made too late, it may decrease the potential

performance improvements. Because accesses to the tables are serialized, a lookup operation may take more cycles with the new CT structure.

3.5. Evaluation

In this section, we experimentally evaluate the effectiveness of the proposed prefetch mechanism. Section 3.5.1 describes benchmarks used for evaluation and simulator architecture such as processor core configuration, memory hierarchy configuration, and prefetcher configuration. In section 3.5.2, we evaluate several hardware-based jump pointer creation mechanisms by comparing them with naïve implementation. Section 3.5.3 contains experimental results for varying jump pointer distance. In section 3.5.7, we analyze our proposed scheme in terms of IPC, and breakdown of memory operation handling.

3.5.1. Experimental setup

Our experiments were performed using the SPEC2K integer benchmark suite, the Olden benchmark suite, and the deltablue pointer-intensive benchmark. From the SPEC2K integer benchmark suite, bzip2, gap, mcf, parser, and twolf were used. From the Olden benchmark suite, bisort, health, mst, treeadd, and tsp were used. All other benchmarks not used in this experiment don't show noticeable improvement, so the results for them are not presented. Olden and deltablue benchmarks were simulated to completion. For SPEC2K benchmarks, a representative 100 million instructions were simulated, which were identified by SimPoint [48]. SPEC2K benchmarks use reference inputs.

For our experiments, SimpleScalar simulator is used [6]. The memory interface has been rewritten to model cache hierarchy, bus occupancy, and limited bus bandwidth, similar to Sharma et

al. [46]. Simulated processor configuration is summarized in Table 3.1.

Table 3.1. Simulation parameters

Processor core configuration	5 stage pipeline; 4 way superscalar; 128 entry Reorder buffer; 32 entry load-store queue, hybrid (bimodal + 2 level) branch predictor; 4 integer units, 2 multiplication units, 1 division unit;
Memory hierarchy configuration	64K, 2way, 32B, 16 MSHR, 2 cycle L1 data cache; 64K, 2way, 32B, 16 MSHR, 1 cycle L1 instruction cache; 1M, 8way, 64B, 16 MSHR,15 cycle L2 cache; 100 cycle memory latency; 2:1 frequency ratio, 32B bandwidth L1-L2 bus; 4:1 frequency ratio, 16B bandwidth L2-mem bus;
Prefetcher Configuration	Direct-mapped, 128 entry FMLT; Direct-mapped, 64 entry CPT; Fully-associative 128 entry PPW; Fully associative 64 entry CT; 32 entry PRQ; 128 entry(4KB), 32B Prefetch buffer;

3.5.2. Effectiveness of hardware-based JPT creation mechanism

In a pure hardware implementation of JPT, the space requirement is the most critical factor. In this section, several mechanisms designed to reduce the space requirement are evaluated.

The first mechanism is to create a jump pointer only for a missed dynamic instance. Since not all instances incur cache misses, it can save some space by not creating unnecessary jump pointers for instances that hit in L1 cache.

The second mechanism is the chaining mechanism explained in section 3.3.2. If two or more consecutive dynamic instances miss in L1 cache, one jump pointer is created for the first instance, and the following two instances can be prefetched by means of chain prefetching. Additional complication for this approach lies in keeping the savings achieved by chain prefetching. Assume a

jump pointer of distance n is created, and the subsequent n-1 instances are prefetched via chain prefetching. Since the following n-1 instances are already covered, new jump pointers for them shouldn't be created. Therefore, there should be a mechanism that distinguishes covered instances from non-covered instances. This can be achieved by recording the chain length of each pointer in the JPT, and using that information when the same jump pointer is encountered again. If a jump pointer is about to be created for an instance whose position is less than the stored chain length from the jump pointer that has already been learned and was encountered again, jump pointer creation is canceled.

The third mechanism is a jump pointer blocking mechanism. There are some static loads whose effective addresses don't repeat. Since the jump pointer approach relies on repeated patterns of address streams, it is not able to obtain any benefit from this kind of load. In addition, these loads waste space, preventing useful jump pointers from being inserted into JPT. This mechanism effectively blocks creation of jump pointers for non-repeating loads by keeping track of the number of jump pointers created and the number of times prefetch requests are made for each CT entry. If the number of created jump pointers is large and the number of prefetch requests made for the pointers is small, the corresponding CT entry is blocked from generating jump pointers. As a side effect, this may end up blocking creation of useful jump pointers. This happens for loads that have a large number of elements in their repeating sequences. To prevent this, the block is removed when the number of prefetch requests exceeds a certain threshold. This way, loads that don't contribute to prefetch are blocked, and loads that have potential for performance improvement are allowed to create jump pointers.

Table 3.2 shows the number of jump pointers created for mcf and health, which get the most benefit from jump pointers. For this experiment, the JPT is unlimited, and the jump pointer distance is four.

Table 3.2. The number of jump pointers created for *mcf* and *health*

	JP created for cache miss	JP created with chain mechanism	Unconstrained JP creation
mcf	69,257	66,917	82,735
health	92,569	72,490	131,001

For mcf, the first and the second mechanisms create 16% and 19% fewer jump pointers, respectively, than the case where jump pointers are naively created for all dynamic instances. For health, the two mechanisms achieve 30% and 45% space savings, respectively. The difference between the cache miss and chaining mechanisms grows if there are many consecutive misses, allowing chain mechanism to take advantage of them. Sometimes prefetch chains are broken for two reasons. First, recurrent loads often load zeros, which happens when the control flow changes. Second, another instruction in a seldom-used different control path produces an effective address that interrupts the stable pointer chasing pattern. If a prefetch chain is broken, the chain ends there and a new jump pointer is created. Since health has longer and fewer prefetch chains than mcf, space saving is greater than that of mcf.

Figure 3.7 shows IPCs of different mechanisms, normalized to the IPC of the no-prefetch case. For unlimited JPT, naïve creation is the best, followed by missed-instance-only creation and chain-prefetch-aware creation. Naïve creation beats the other two simply because it has more jump pointers. For 32k JPT, chain-prefetch-aware creation performs the best. Even though the number of jump pointers created for missed-instance-only creation and chain-prefetch-aware creation are similar (mcf), chain-prefetch-aware creation actually prefetches more because of chain prefetching.

The blocking mechanism described earlier doesn't seem to have big impact on performance. This is because in mcf and health, there is no such load whose effective address stream doesn't

repeat. For parser, mst, tsp, the blocking mechanism reduces space requirements dramatically with little performance degradation, but they don't benefit much from jump pointers.

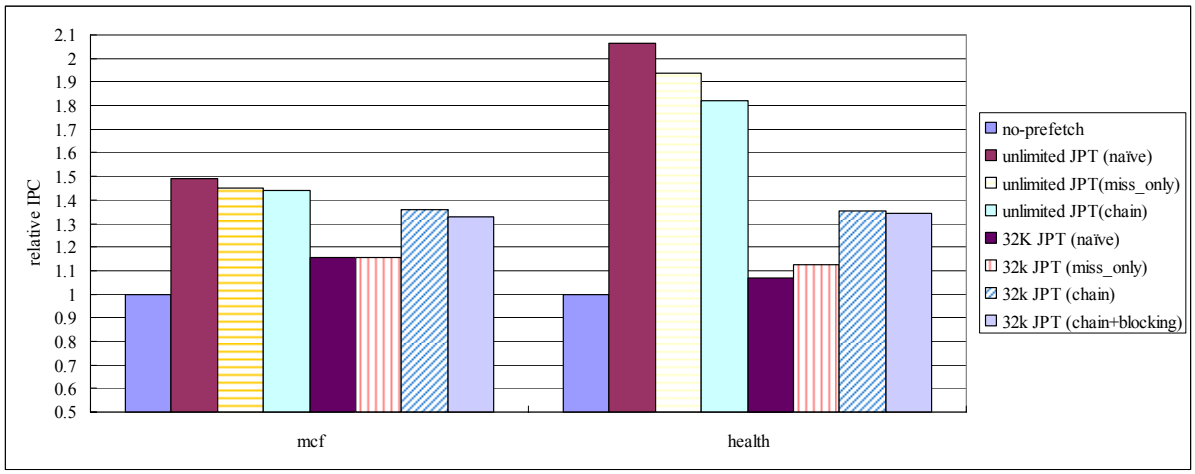


Figure 3.7. The effectiveness of JP creation mechanisms in terms of IPC

3.5.3. Jump pointer distance

In this section, we examine how effectiveness of jump pointers changes as jump pointer distance is varied. Jump pointer distance is the number of instances between the instance for which the jump pointer is created and the instance the pointer points to. Figure 3.8 shows prefetch reference breakdown: full hit, partial hit, miss. Full hit means the L1 cache miss is completely hidden, and partial hit means the L1 cache miss is partially hidden. Mcf, health, and deltablue are selected to show the breakdown, because they benefit from the jump pointer prefetching scheme. For this study, JPT is unlimited, only jump pointer-based prefect is used, and perfect branch prediction is assumed.

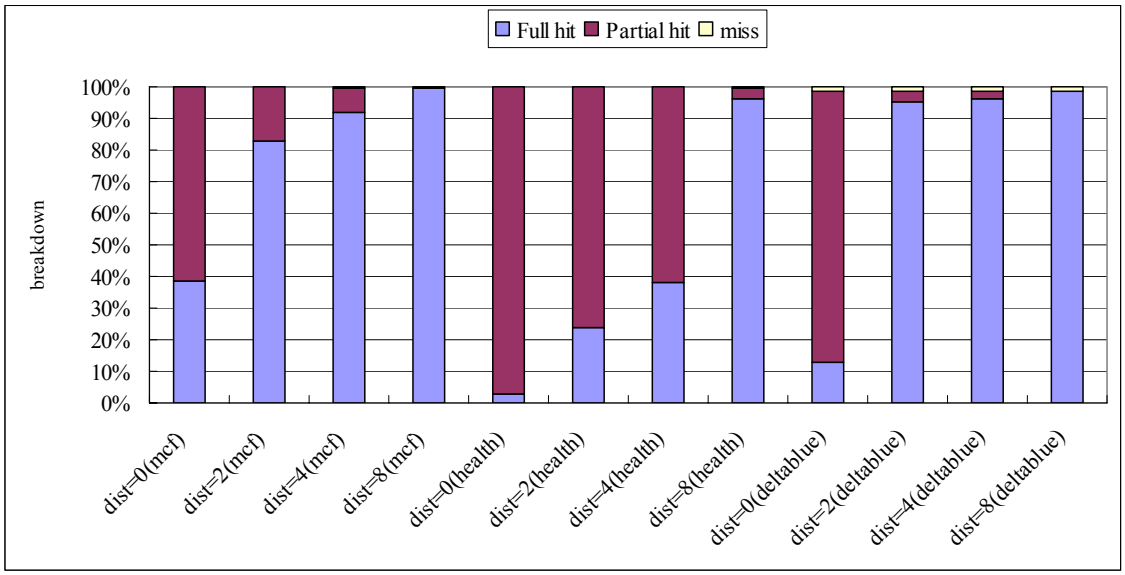


Figure 3.8. Prefetch reference breakdown with varying jump pointer distance

As jump pointer distance increases, the number of full hits also increases and the number of partial hits decreases. This is because prefetches can be issued much earlier, making the prefetches arrive at the prefetch buffer before they are used. Note that the rate at which the number of full hits grows is different for each benchmark. For mcf and deltablue, the number of full hits grows fast but for health, it grows slowly.

Figure 3.9 shows the IPC. At jump pointer distance two, IPCs of mcf and deltablue already saturate, whereas IPC of health saturates at distance four. This is in accordance with what we found in figure 3.8.

Inaccuracy that might be manifested at large jump pointer distance doesn't appear in the figures. Not shown clearly in figure 3.8, the number of misses increases as the distance increases. However, any negative impact of the inaccuracy is overrun by the IPC boost from converting partial hits to full hits.

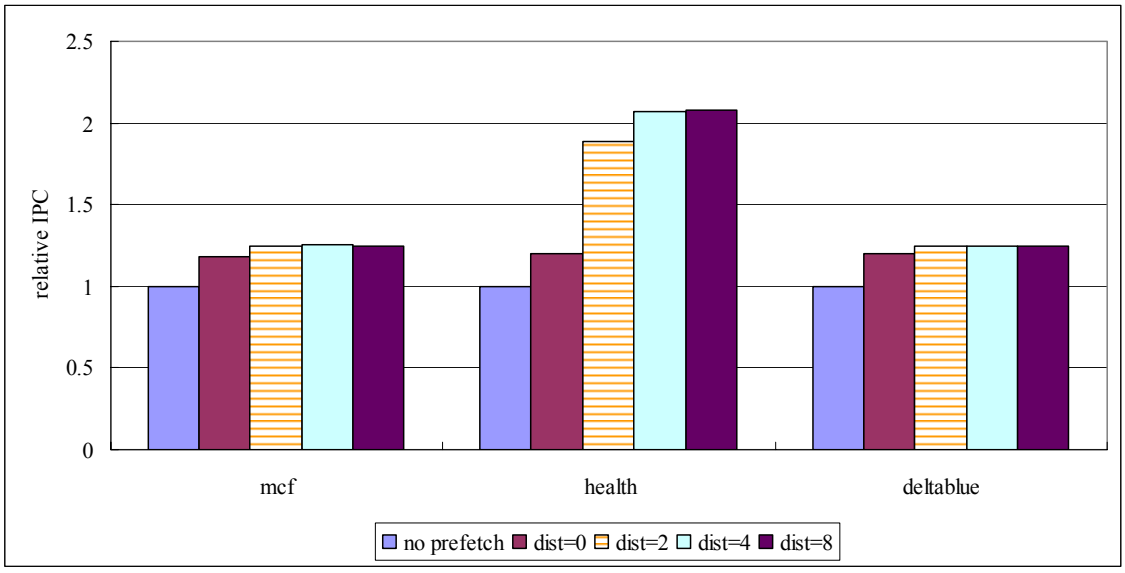


Figure 3.9. Influence of jump pointer distance on IPC

3.5.4. Jump pointer accuracy

Jump pointers are used to predict future addresses that a processor is likely to access. Therefore, accuracy of jump pointers is an important factor that determines prefetcher efficiency. If the accuracy is low, the cache could be polluted with useless data, and demand fetches may be delayed due to resource contention with useless prefetches.

Figure 3.10 shows the accuracy of jump pointers with varying jump distance. A jump pointer is a pointer that points to the address of a future dynamic instance, and it is associated with the address of the current dynamic instance. Jump distance is the number of dynamic instances between the instance that the pointer is associated with and the instance the pointer points to. The x-axis represents jump distance and the y-axis shows jump pointer accuracy. Each line indicates a load that jump pointers are created from. The numbers of the lines are the last four decimal digits of the loads.

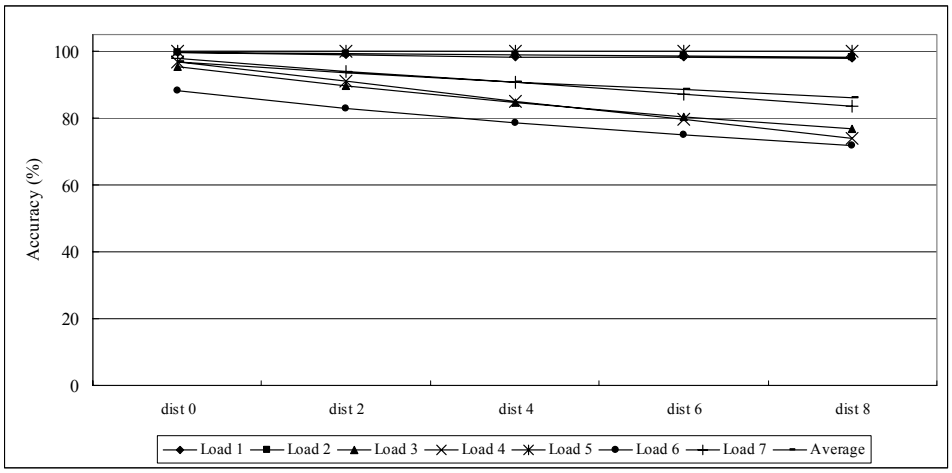
For benchmarks mcf and health, recurrent loads with a lot of dynamic instances are selected

and their accuracies are measured.

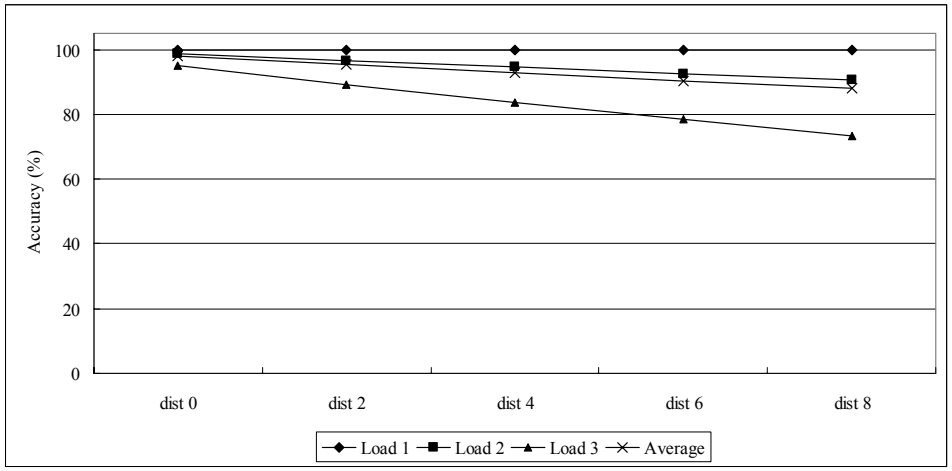
To measure the accuracy, the stream of effective addresses and loaded values for each load instruction is generated and analyzed. Jump pointers for a given distance are created and stored in a table. For each jump pointer, two addresses are stored: the address the pointer is created from, and the address that the pointer references. When the address the pointer references is seen again, the stream of address/value pair is checked with the given jump distance. If the same addresses are paired again, the hit counter is incremented. Each effective address and loaded value pair updates the jump pointer table to learn changes in the LDS.

The graph shows that accuracy drops linearly as jump distance increases. On average, 86% - 87% accuracy is achieved with distance 8, but for some static load instructions, accuracy drops below 75% with distance 8. This leads us to conclude that jump distance needs to be as small as possible, as long as it is still large enough to hide cache misses.

Some of the useless prefetches could be avoided if store instructions that change pointers in LDS are allowed to update jump pointers. Pointer cache [12] employs a store teaching technique to handle this problem. However, jump pointers are hard to update, because the address of the pointer itself is not recorded.



(a) *Mcf*



(b) *Health*

Figure 3.10. Accuracy with varying jump distance

3.5.5. Sensitivity of the correlation table to ports and access latency

The correlation table (CT) is the center of all prefetching activities: learning producers' patterns (update) and generating prefetch requrest (lookup). It stores producer-consumer pairs, learns producers' memory access patterns, and generates prefetch requests. When an entry is created in the CT, it requires a write operation. When the CT is updated in order to learn producers' memory access patterns, it also requires a write operation. When the CT is probed to generate prefetch

requests, a read operation is performed. Since the CT is accessed quite a lot due to these updates and lookups, the impact of the number of ports on the CT and access latency of the CT need to be studied.

Figure 3.11 shows the impact of the number of ports on the CT to overall IPC. The number of ports is varied from one read and one write port up to eight read and eight write ports. To measure the influence of modeling the ports, a result without modeling the ports is included. Also, a case with a set of small buffers, which temporarily stores accesses to the CT, is presented. The y-axis represents relative IPCs normalized to the IPC of the case without modeling the ports.

For twolf, the number of ports doesn't matter at all. But for mcf and health, when only one read and write port are available, significant IPC decrease is observed. This is because some of the crucial producers are not identified due to contention on the ports. If a set of small buffers are placed to temporarily store read and write operations respectively, one read port and one write port are turned out to be sufficient.

The sensitivity of the CT to access latency is shown in Figure 3.12. The access latency of the CT varies from two cycles to eight cycles. Y axis represents relative IPCs normalized to the IPC when access latency is set to 2 cycles. It is observed that all of the benchmarks are not sensitive to the access latency of the CT. This means that the access latency of the CT can be tolerated by the out of order processor core and the look-ahead capability of the proposed prefetching scheme makes the CT insensitive to its access latency.

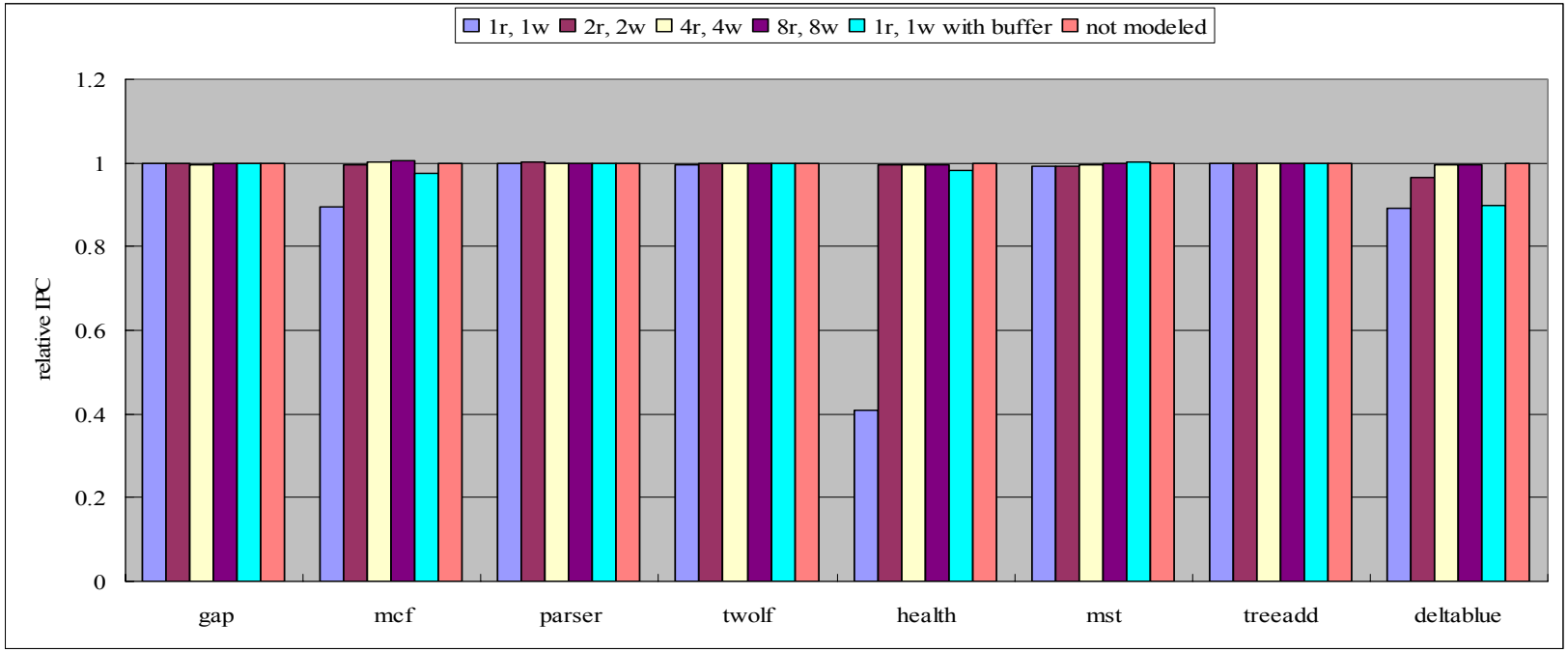


Figure 3.11. Sensitiveness of the CT to the number of ports

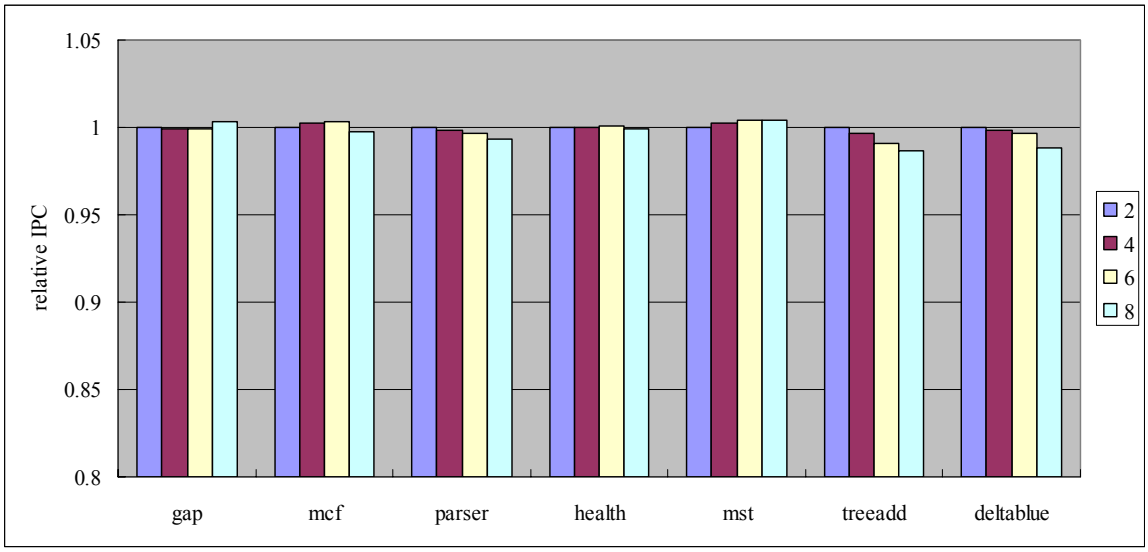


Figure 3.12. Sensitiveness of the CT to its access latency

3.5.6. Sensitivity of the potential producer window to ports and access latency

The potential producer window (PPW) is a FIFO queue, in which committed loads' information is stored. consumer to producer matching is performed in the PPW, and this action requires traversing the queue until a corresponding producer is found. To accurately implement the PPW, ports and access latency of the PPW are modeled. For example, if the length of a traversal is three PPW entries and access latency is two cycles, the corresponding port will not be ready in six cycles. If there is no available port, the corresponding request is simply discarded. For this experiment, write port is assumed to be sufficient. This is because writing multiple entries per a cycle can be implemented relatively inexpensively. Therefore, only the read ports are considered.

Figure 3.13 shows the sensitiveness of the PPW to the number of read ports on it. Y axis represents relative IPCs normalized to the IPC of the case where the number of read ports is set to one. X axis is the number of ports on the PPW, and the number varies from one to four. As shown in the figure, none of the benchmarks is sensitive to the number of ports. This is because consumer to

producer matching doesn't occur frequently due to the filtering mechanism and updating the CT is not in a critical path.

Figure 3.14 presents the sensitiveness of the PPW to its access latency. Y axis represents relative IPCs normalized to the IPC of the case where the access latency is set to one. X axis represents the access latency of the PPW, and the latency varies from one cycle to four cycles. All benchmarks are not sensitive to the access latency of the PPW. The same reasons as the number of port experiment apply. Comsumer to producer matching doesn't occur frequently enough to have the access latency influence overall IPC, and updating CT is not in a critical path.

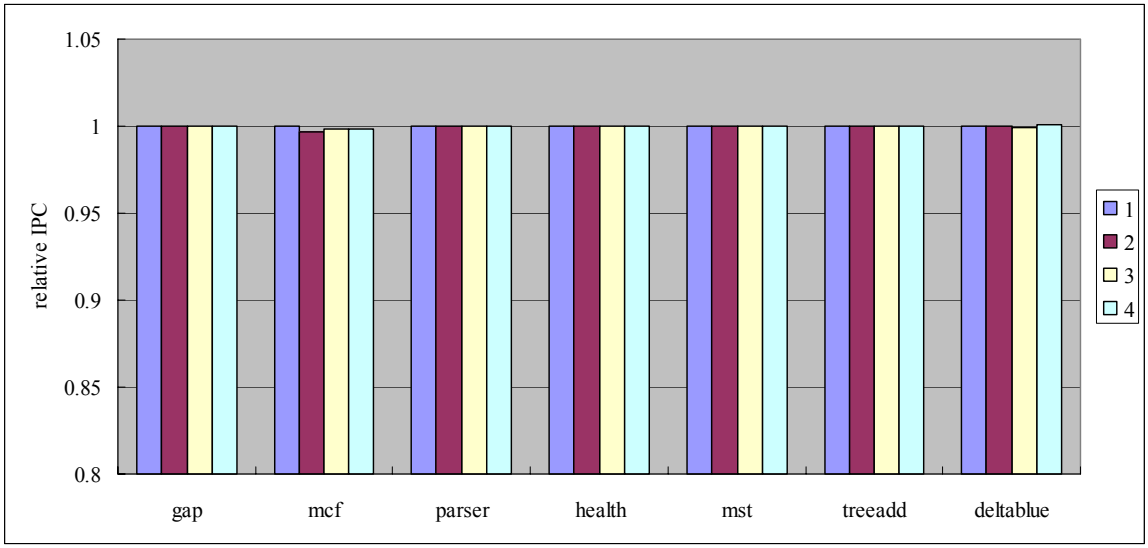


Figure 3.13. Sensitiveness of the PPW to the number of ports

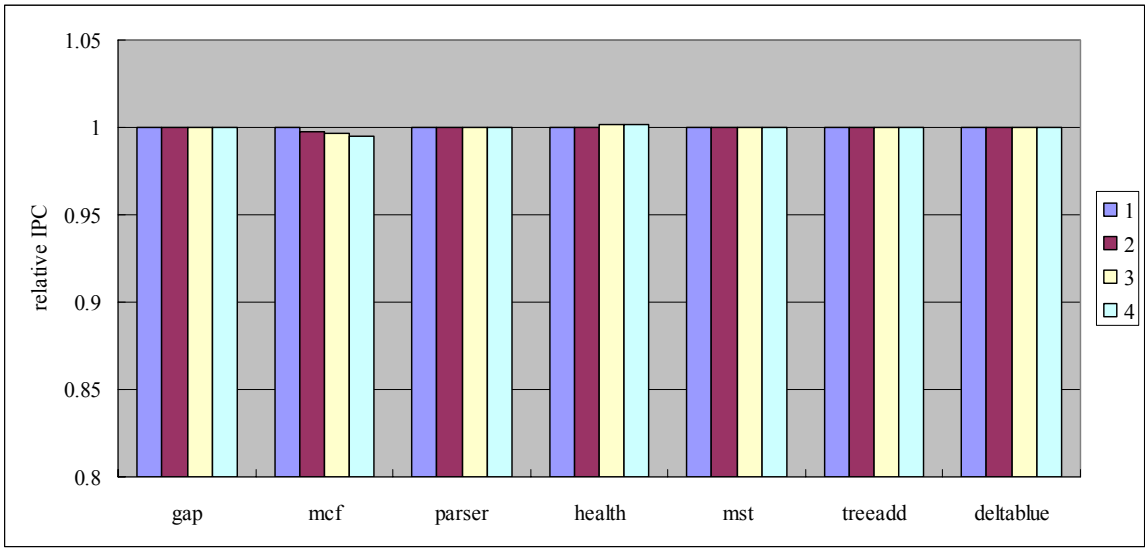


Figure 3.14. Sensitiveness of the PPW to its access latency

3.5.7. Effectiveness of the proposed prefetcher

To evaluate the effectiveness of the proposed prefetcher, systems with 1M L2 cache and 2M L2 cache are compared with the system with 1M cache and the prefetcher. Two versions of the prefetcher are presented: one with 32K JPT, the other with 64K JPT. The prefetcher with 32K JPT takes about 512KB, and the one with 64K JPT consumes about 1MB. The prefetcher has chain mechanism described in section 3.3.2, jump pointer distance is set to four, and JPT access time is set to 18 cycles for both sizes.

Figure 3.15 presents breakdown of prefetch hits: stride-based prefetch, jump pointer-based prefetch, and chain-prefetch. Each bar has three regions, each of which indicates a mechanism that contributes to prefetch hits. Note that each benchmark has a different number of prefetch hits.

Jump pointer-based prefetching is dominant for mcf, twolf, bisort, health, and tsp. Stride-based prevails in gap, parser, mst, and treeadd.

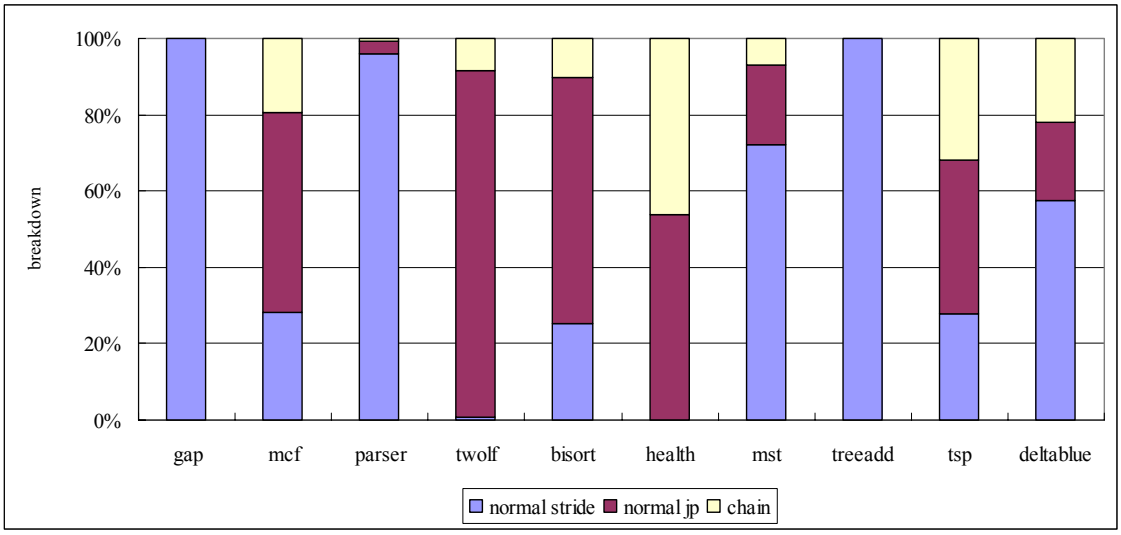


Figure 3.15. Breakdown of prefetch hits: stride-based prefetch, jump pointer-based prefetch, and chain-prefetch

Figure 3.16 shows the IPCs of seven different configurations. The first two bars represent the IPCs of two different L2 cache sizes: 1MB and 2MB without prefetching. The next three bars are IPCs of three prefetching schemes: he proposed prefetching with 64K-entry JPT, original dependence-based prefetching (DBP) [43], and jump-pointer prefetching (JPP) with 64K-entry JPT [44]. The last two bars show IPCs of perfect L2 Cache and perfect L1 Cache, respectively. In this section, all prefetching mechanisms are evaluated with dedicated prefetch tables and 1MB L2 cache. (Storing the prefetch table in L2 is evaluated in the next section.) The IPCs are normalized to that of 1MB L2 cache without prefetching.

For some benchmarks such as *gap*, *treeadd*, and *equake*, going from 1M L2 cache to 2M L2 cache doesn't improve performance more than 2%. This means that the advantage of the bigger L2 cache size is not effectively utilized to improve performance.

To quantify the potential of prefetching, the performances of perfect cache at L1 and L2 cache are measured. Since the proposed scheme eliminates L1 cache misses, the IPC of perfect L1 cache indicates its full potential. The performance of prefetching mechanisms that remove L2 cache

misses is bounded by the performance of perfect L2 cache.

As clearly shown, perfect L1 cache has much more potential than perfect L2 cache. This observation, along with the first observation about L2 cache size, suggests a possible way to use the L2 cache differently. The first observation shows that for some benchmarks, L2 cache is not effectively utilized, meaning some portion of L2 cache can be used for a different purpose such as prefetching. The second observation hints that even though perfect L2 cache yields benefit, there is still much room for performance improvement.

The proposed prefetching mechanism is implemented with 64k entry-JPT, consuming about 1MB of space. The JPT is implemented separately from 1MB L2 cache. If we compare the bar for the proposed mechanism with the one for 2MB L2 cache without prefetching, we can see how efficient it is to have both prefetcher and L2 cache, given a certain silicon estate. Except for *bzip2*, *twolf*, *vpr*, and *art*, the combination of 1MB prefetcher with 1MB L2 cache outperforms the 2MB L2 cache without the prefetcher. For *health*, the bigger cache improves the performance quite a lot, but the prefetching mechanism is able to improve the performance more than the bigger cache can.

If we compare the proposed prefetching scheme with the existing two schemes (DBP and JPP), it is observed that the proposed prefetcher outperforms the older ones. The proposed prefetching scheme is better than the original dependence-based prefetching, for two reasons. First, the proposed prefetching scheme uses producers' memory access patterns to predict future addresses of producers. This enhances the timeliness of the prefetching scheme, and therefore converts many prefetch partial hits to full hits. For the original dependence-based prefetching, 95% of the prefetch hits are partial hits, but with the proposed prefetching, only the 28% of the prefetch hits are partial hits. Partial hits occur, when prefetched blocks do not arrive before they are demanded by the processor. The second reason is that the proposed mechanism utilizes producers' patterns other than recurrent pattern, while the original dependence-based mechanism is solely dependent on recurrent

pattern. With the original dependence-based prefetching, prefetch partial hits constitute 64% of all prefetch hits, but with the proposed prefetching, prefetch partial hits are only 22% of all prefetch hits.

The proposed prefetching mechanism is better than original jump-pointer prefetching for three reasons. First, the proposed scheme selectively stores jump pointers by using chain prefetching and FMLT. Second, the proposed scheme uses not only recurrent pattern but also other patterns, such as stride pattern. Third, the proposed jump pointer creation mechanism is more space-efficient than the original mechanism. Hence, the proposed mechanism results in higher performance if the size of the JPT is limited.

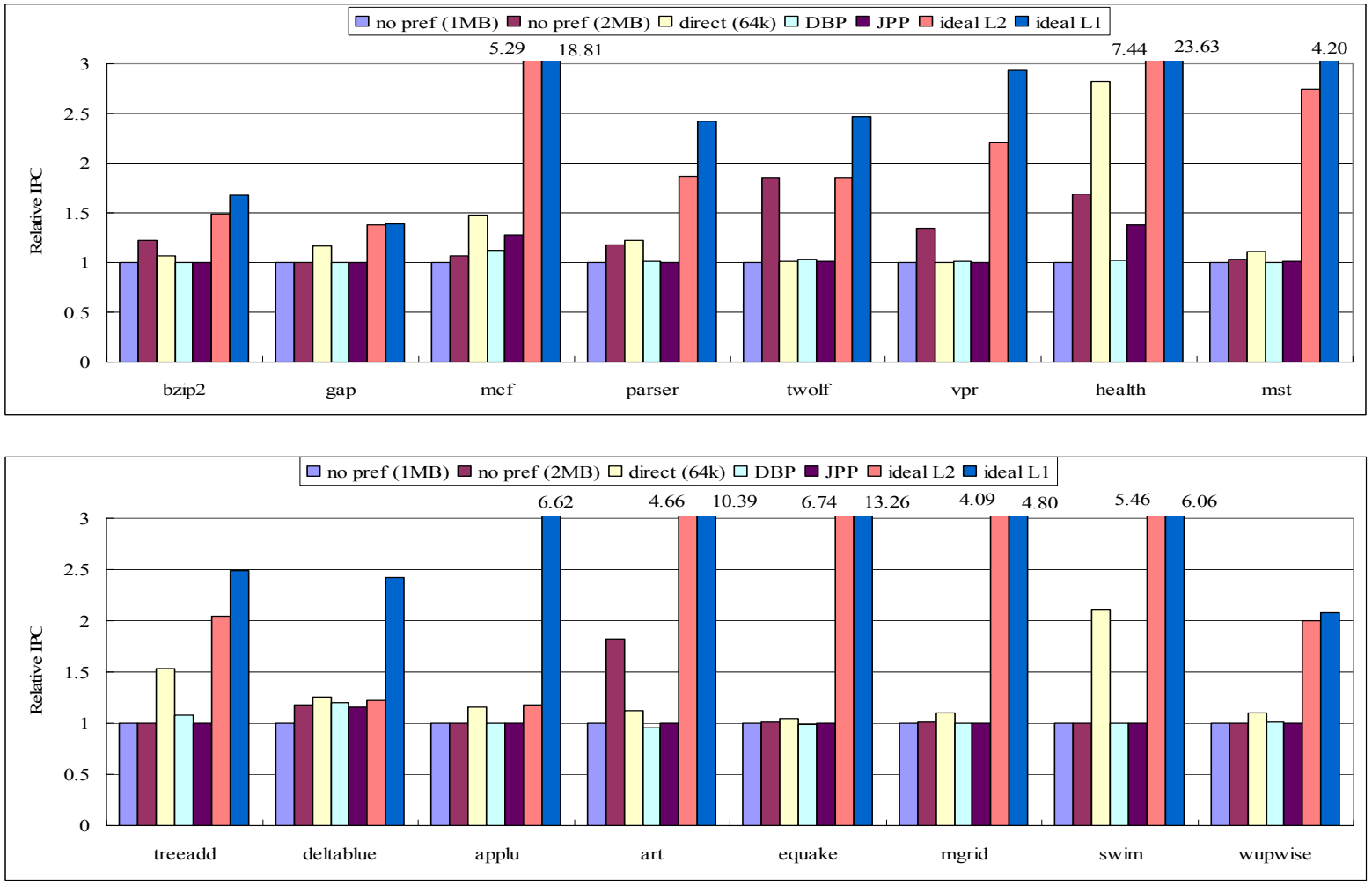


Figure 3.16. Performance comparison

3.6. Conclusion

This chapter has presented two enhancements to traditional dependence-based prefetching. First, once producer-consumer relationships are discovered, patterns in the producer's address or value stream are exploited to prefetch further into the future. For this study, we have used stride and recurrence patterns, commonly found in pointer-intensive programs. It is shown that this improved timeliness results in higher performance in terms of IPC.

Second, our hardware-based jump pointer mechanisms capture the most effective recurrent patterns, reducing the number of jump pointers, compared to traditional schemes. Through experimental evaluation, the proposed prefetcher is shown to be more effective than original dependence-based prefetching and jump-pointer prefetching as well as increasing the size of L2 cache.

Chapter 4

Exploiting Linear Dependence Relations to Enhance Prefetching

4.1. Motivation

The original dependence-based prefetching (DBP) [43] is based on direct relations between producer loads and consumer loads. This means the base effective address of a consumer is the same as the loaded value of the corresponding producer. The coverage of direct relation varies according to data structures used in applications. For applications that manipulate linked data structures (LDS), the direct relation captures most of the producer and consumer pairs. However, for applications that manipulate arrays, the direct relation is not able to capture many pairs because addresses of consumers are typically computed, not loaded.

Even for the cases where addresses are computed, one or more loads are involved in the computations. In this study, only the dependence chains that contain only one source of variability are considered. If two loads in a dependence chain produce different values every time they are executed, neither load can be a producer because one producer is not able to predict addresses of the corresponding consumer. In most cases, a load is at the top of dependence chains if we select dependence chains the way just described. Judging from the instructions in a chain, the relation

between corresponding producer and consumer doesn't look simple. This is because multiple instructions are involved in a chain and even load is involved in the chain. Therefore, we investigate the relations between producers and consumers for the cases where direct relation fails. If it is possible to represent those relations with a simple equation, the coverage of DBP can be improved.

We first tried curve fitting algorithms, and found that linear equations are enough to represent most of the relations. Since linear equations are relatively simple and linear relation is a superset of direct relation, linear relation has potential to replace direct relation.

4.2. Linear relations

Usually, a load's effective address can be computed in two different ways. The first way is that a base effective address of a load is directly loaded from memory into a register, and the contents of the register is added to the offset of the load to produce the effective address. This type of computation prevails in applications that manipulate linked data structures.

The other way involves more than one instruction: at least one load instruction for loading base address, and a set of instructions that produce an offset value to be added to the base address. The instructions for the offset value may or may not contain a load instruction but do contain arithmetic instructions. By utilizing linear relations, more producers can be identified in addition to producers identified with direct relations.

Some example code segments are shown in Figure 4.1. The code from *mcf* shows direct relations. Suppose instructions 4 and 6 miss at L1 cache frequently, and we want to identify a producer load for them. It is easy to see those instructions 4 and 6 depend on instruction 1, and their relations are direct relations. In other words, the output value of instruction 1 is the base effective

address of instructions 4 and 6. In *mcf*, instruction 1 corresponds to a load for the next object in a linked list.

The code from *bzip2* is an example of a linear relation. Assume instruction 8 is a frequently missing load, and we want to find a producer for it. It is hard to capture the relation by just looking at the code, but it is found that instruction 2 has a linear relation with instruction 8. The loaded value of instruction 2 is doubled by instruction 4, and instruction 6 adds it to r2, which is a constant. This constant is produced by instruction 1, which accesses one memory location and keeps loading the same value. Instruction 7 has a direct relation with instruction 8, but it is likely that output values produced by instruction 8 do not have any pattern. Instruction 7 cannot have recurrent pattern because it is not a load, and if it had a stride pattern, instruction 8 would have been prefetched by stride prefetching. If instruction 1 has a pattern in its address stream, it is possible to predict a future address of instruction 1 and use it to compute a prefetch address for instruction 8, using the expression 2*x*+r2.

```
1: ldq      r27, 24(r27)
2: ldq_u    r31, 0(r30)
3: bne      r27, 0x1200074fc
4: ldq      r8, 56(r27)
5: beq      r8, 0x120007528
6: ldq      r19, 16(r27)
```

Code segment from *mcf*

```
1: ldq      r2, 0(r1)
2: ldl      r4, 0(r4)
3: ldq_u    r31, 0(r30)
4: addq     r4, r4, r5
5: cmplt    r4, 20, r10
6: addq     r2, r5, r2
7: bic      r2, 2 r12
8: ldl      r16, 0(r12)
```

Code segment from *bzip2*

Figure 4.1. Code segments from *mcf* and *bzip2*

There are two main differences between direct and linear relations. The first difference lies in how relations are identified. Direct relations, they can be identified by either comparing the source register number of a consumer with the destination register numbers of preceding instructions, or

comparing the base effective address of a consumer with the output values of preceding instructions. For linear relations, information about at least two dynamic instances of potential producers and a consumer is required. This is because two points are needed to get a linear equation (ax+b) that connects the two points. It is impossible to capture a linear relation by just looking at code. Since any instruction in the corresponding dependence chain can be a producer, we need to observe output values of candidate producers and addresses of the consumer for some number of dynamic instances. The second difference is the number of instructions in relations. Direct relation doesn't have any instruction between a producer and its consumer, while linear relations can have several instructions between them. Compared to the dependence-graph pre-computation approach [3], a linear relation approach compresses a sequence of instructions into a linear expression, which is far more efficient in terms of storage requirement.

There are cases where the linear relation approach doesn't do well. For example, shifting to the left a large number of times truncates bits, making linear dependence identification mechanism unable to catch the relation.

4.3. Distributions of direct and linear relations

Distributions of direct and linear relations depend on the way effective addresses are computed in an application. The distributions of some of the SPEC2KINT benchmarks are presented in Table 4.1.

The statistics are collected only for the frequently missing loads. Those frequently missing loads are first identified, and their producers are manually selected, one for each control path. In the relations, there is no chain of two loads except for the one that involves the consumer load. "No need" means that loads have stride patterns in their address stream, so that searching for their

producers is not necessary. "Direct" and "linear" indicate the type of relations, and "stride", "recurrent", and "others" represent producers' patterns that can be used to predict future addresses of their consumers. "Others" encompasses patterns other than stride and recurrent, such as repeating sequences, or even sequences with no pattern. For the consumers with this type of producer, new ways of prefetching should be devised in order to get prefetched. A couple of potential solutions may be finding a producer of the producer without any usable pattern, or using the output value of such a producer directly for prefetching if it is far away from its consumer. "No producer" represents the case where producers do not exist in the potential producer window (PPW).

Table 4.1. Frequency of occurrence (%) of dependence patterns

	no need	direct-stride	direct-recurrent	direct-others	linear-stride	linear-recurrent	linear-others	No producer
bzip	16.7	0	0	0	79.3	0	4.1	0
gap	86.4	13.6	0	0	0	0	0	0
mcf	10.4	8.9	67.3	13.5	0	0	0	0
parser	46.2	7.8	10.8	35.1	0	0	0	0
twolf	0	27.8	33.9	26.0	0	0	10.5	1.8
vpr	0	0	0	34.5	25.9	0	39.6	0

As it shows, *bzip*, *twolf*, and *vpr* have linear relations, while others mainly have direct relations or "no need". Especially for *bzip* and *vpr*, only the prefetches by linear relations can potentially increase their performances. Plain stride prefetching can do well with benchmarks with high "no need" percentages: *gap* and *parser*. Within linear relations, there is no "linear and recurrent" relation. This is because for recurrent pattern, producers' output values are directly used for consumers' addresses at all times.

There are not many consumers with "no producer", meaning the current size of PPW (256) is big enough for these benchmarks. Even though large portions of frequently missing loads have producers with usable patterns, or are self-prefetchable ("no need" case), there is still a gap between

actual improvements and the improvements expected from the statistics shown in the table. There are three reasons. First, usable patterns are not perfectly stable. For the classification used for the table, not only stable strides but also somewhat weak strides are considered as "stride". Even with stable stride patterns, it is not always possible to prefetch perfectly. For example, depending on how close each dynamic instance of a consumer is to each other, the distance between the current instance and the instance whose address is predicted should be different. For the ones whose instances are close to each other, the prediction distance (the number of dynamic instances between the current one and the predicted one) should be far enough to fully hide the miss latency of the predicted instance. Second, as shown in the table, there are many producers with non-usable patterns. There may be solutions to this, but right now they are not utilizable. Third, control flow variations hinder prefetching. It is possible for a consumer to have multiple control paths to it, and a different producer exists in each control path. If a producer's pattern is used to predict a future address of a consumer but a different control path is taken, prefetching accuracy may drop, resulting in lower performance gain.

4.4. Added hardware complexity

The new mechanisms proposed in addition to the direct-relation-based mechanism are linear-relation-based mechanism and direct-relation-filtered linear-relation-based mechanism. The former exclusively uses linear relations but the latter uses both direct and linear relations. For the latter, the PPW is searched by using both direct-relation-based mechanism and linear-relation-based mechanism. For the direct-relation-based search, loaded values stored in the PPW are compared with current consumer's effective address. For linear-relation-based search, destination register numbers in the PPW are compared with the source register numbers of the current consumer. Only if the

direct-relation-based mechanism fails, candidates identified by linear-relation-based mechanism are used instead. The latter is proposed in order to reduce the space overhead of linear-only mechanism.

Figure 4.2 shows the added hardware (relative to the base mechanism) required for the linear-relation-based mechanism and direct-relation-filtered linear-relation-based mechanism. Shaded units and regions represent extra hardware.

The extra hardware required only for the filtered linear mechanism is the data forwarding logic. For linear mechanism, the data forwarding logic is not necessary because all candidate pairs from the PPW should go through the divider queue. For linear-filtered mechanism, the data forwarding logic delivers data to two different units: the divider queue, and the candidate pair table. The data path labeled '1' is used when the linear-relation-based producer matching succeeds. The path labeled '2' is used when the direct-relation-based producer matching is successfully performed. This data forwarding logic corresponds to the direct-relation filtering mechanism.

All other shaded units are common for both mechanisms. PPW should be updated, not only by load instructions, but also by all output-producing instructions. Because the linear mechanism has to track dependence chains, all instructions should be present in the PPW to track the dependence chain correctly. Direct mechanism requires a simple comparator to identify a producer, but the linear mechanism tracks the dependence chain with source register and destination register numbers. This requires dependence chain tracking logic.

While the direct mechanism finds one potential producer at a time, the linear mechanism finds multiple potential producers. This is because any instruction in the dependence chain can be a producer. Since multiple candidate producers must be kept per consumer load, space requirements in the candidate pair table may be elevated unless special care is taken. We varied the size of the candidate pair table and tried several ideas about how to effectively utilize the candidate pair table.

To compute two coefficients in a linear relation (equation), a divider is required. Since the

divider is not pipelined, input from PPW should be temporarily stored in the divider queue. If the queue is full, inputs to the divider queue are simply discarded. Also, to compute prefetch addresses, a multiplier is needed.

The consumer table, a part of the correlation table (CT) should be modified in order to store the two relation coefficients ‘a’ and ‘b’. The PPW should also need to be augmented to include source and destination register numbers, which are used for linear-relation-based producer pairing.

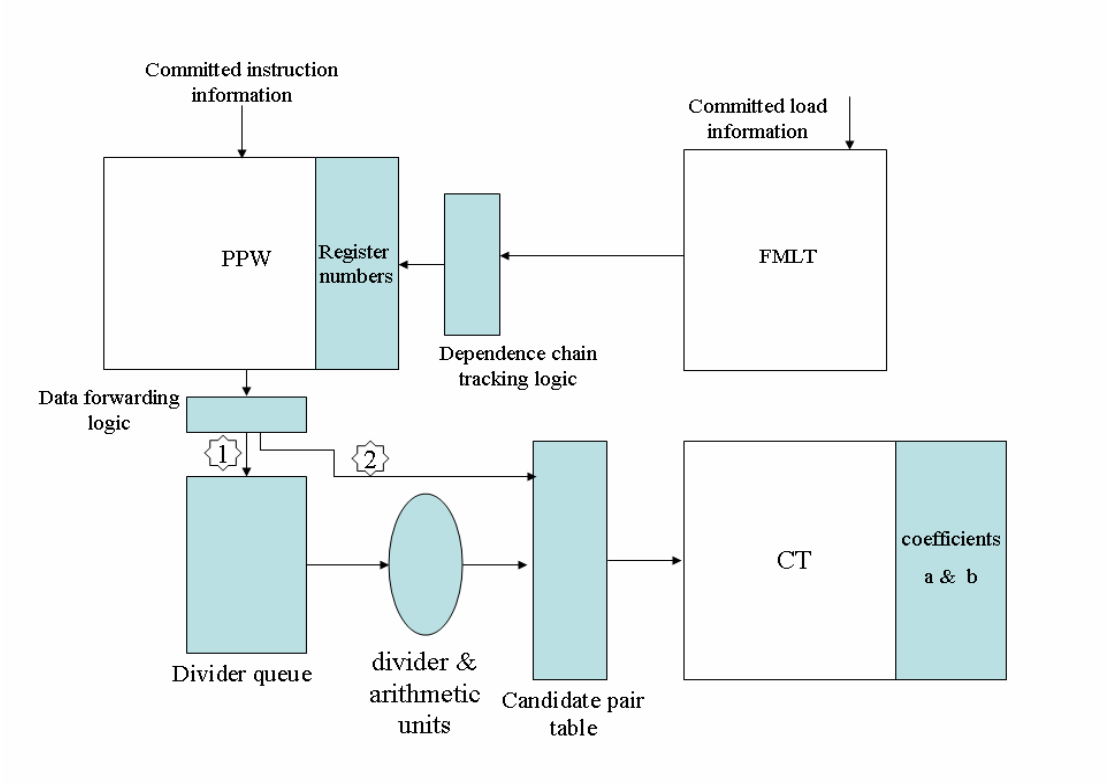


Figure 4.2. Extra hardware for linear mechanisms

The key extra units such as arithmetic units including a divider, dependence chain tracking logic, candidate pair table, and data forwarding logic are explained in more detail in the next subsections.

4.4.1. Arithmetic units

As explained earlier, arithmetic functional units are required for learning coefficients ‘a’ and ‘b’ and generating prefetch addresses. For learning the coefficients, at least a divider, a multiplier, and an adder are required. Since the divider is assumed to be non-pipelined, it is likely to be the bottleneck. This means that having multiple sets of multipliers and adders doesn’t help. Because a division operation takes 20 cycles, requests sent to the divider during a division operation need to be temporarily stored in order not to lose them. This queue is placed between the PPW and the divider, and stores candidate pairs for the consumer load in the candidate pair table. To achieve this, a list of consumers in the candidate pair table is maintained and checked when the divider queue is accessed. Only the candidate pairs for the consumers in the candidate pair table are stored in the divider queue. The purpose of this is to facilitate learning process in the candidate pair table.

For generating prefetch addresses, at least a multiplier and an adder are required. Since multipliers and adders can be pipelined, just a small number of them are enough.

4.4.2. Dependence chain tracking logic

Unlike the direct-relation-based mechanism, which identifies producers by comparing producers’ loaded values with consumers’ effective addresses, linear-relation-based mechanism relies on destination and source register numbers for tracking dependences.

A list of source registers is maintained in order to track dependences. In the beginning of the dependence tracking operation, the source register numbers of a consumer is stored. While traversing the list of preceding instructions in PPW, the list is updated. If the destination register number is the same as one of the source register numbers in the list, the source register number is removed, the destination register number is recorded as a source register number, and the corresponding

instruction is identified as a candidate producer.

If any of the following three conditions is met, traversing is terminated. First, there is a hard limit on the number of candidate producers. If the number of potential producers that have been discovered exceeds a predefined threshold, traversing is terminated. Second, if two instances of the same static producer instruction are encountered, traversing is ended. This is because encountering a static instruction twice means we've crossed into the previous loop iteration. Third, if two instances of a static consumer instruction are encountered, traversing is stopped. The same reason as the second condition applies here. To implement this logic, a structure that records source registers, a structure that keeps track of previously-identified producers, and corresponding control logic are required.

For the direct-relation-filtered linear mechanism, the dependence chain tracking logic searches the PPW for both direct-relation and linear-relation. This doesn't mean two separate searches of the PPW are conducted. When an entry of the PPW is accessed, both loaded value and register numbers of the entry are read and processed together. This search terminates as soon as a direct-relation-based producer is found. If such a producer is not found, search ends according to the three conditions mentioned above.

4.4.3. Candidate pair table

The role of candidate pair table in direct-relation-based mechanism is to filter out unstable candidate pairs; only the stable pairs get written into correlation table (CT). The same role is played by the candidate pair table in linear-relation-based mechanisms. But the difference is that more candidate pairs are generated by the linear mechanism. Moreover, one candidate pair is supposed to be selected among multiple candidates for a consumer. This requires a more tailored structure such

as a set of small tables, each of which is assigned to a consumer. Therefore, a bigger candidate table is not a solution but a partial solution. The small tables do not need to be physically separated. A big table can be logically partitioned and each partition is accessed individually. Figure 4.3 shows how the candidate pair table is implemented.

For direct-relation-filtered linear mechanism, every unit in the figure should be used, but for linear-relation-based mechanism, the shaded units are not necessary. The candidate pair table for linear relation is divided into four partitions, and for indexing into the correct partition, the mapping table has to be accessed first. The mapping table stores consumers' PCs and their partition number, with which indexes to the candidate pair table can be computed. In this way, candidate pairs that belong to a consumer are inserted into one partition making it easier to select a pair out of multiple candidate pairs.

When the direct-relation-filtered linear mechanism is used, candidate pairs that are identified by direct-relation-based mechanism are stored separately from the pairs identified by linear-relation-based mechanism. As explained previously, forwarding candidate pairs to the right candidate pair table is performed by the data forwarding logic.

We found that the number of partitions in the candidate pair table for linear relations can be as few as one. Two achieve this, the candidate table holds candidate pairs of only a static consumer load. It means a dependence relation is learned one after another. In order to prevent dominating (frequently missing) consumers from monopolizing the table, a list of consumers that their producers have been found are maintained in a small FIFO queue. This queue is checked before inserting pairs of a consumer into the candidate pair table. If a match is found in the queue, the consumer is blocked from the candidate pair table.

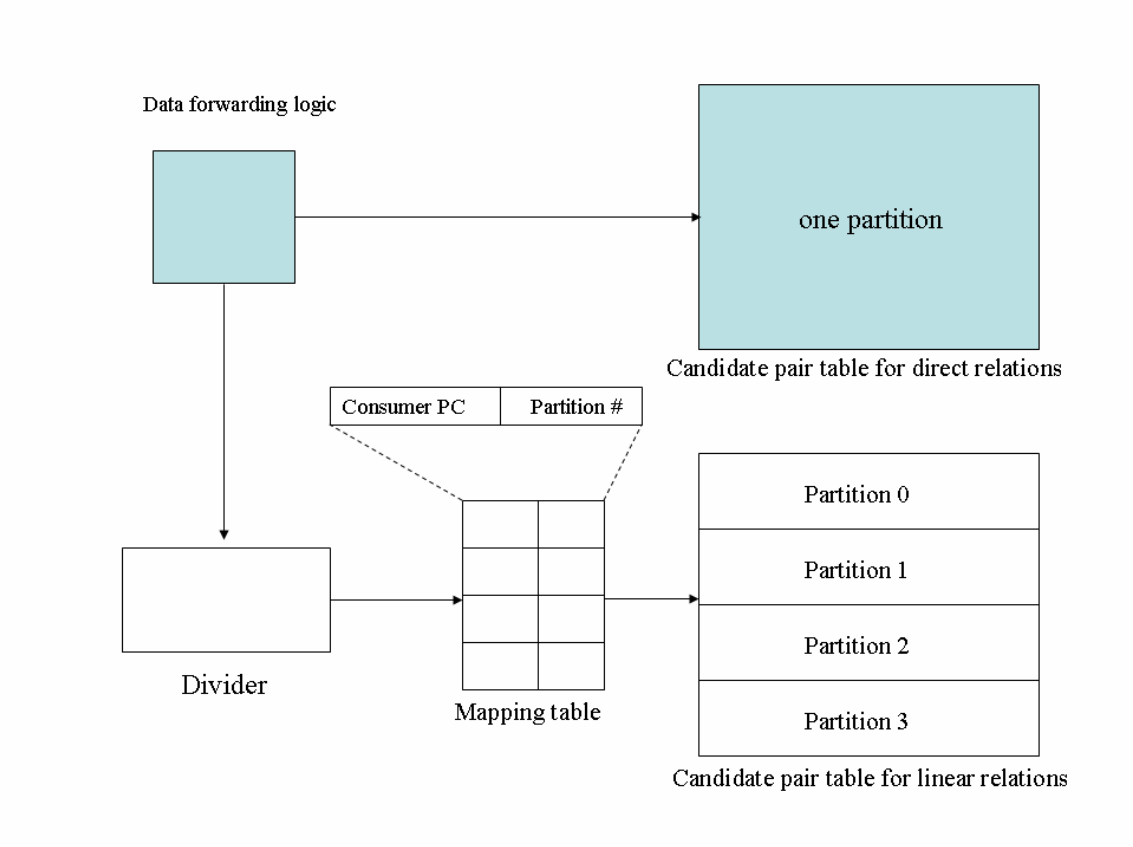


Figure 4.3. Candidate pair table implementation

4.4.4. Data forwarding logic (direct-relation filtering mechanism)

If linear-relation-based prefetching mechanism is implemented naively, the candidate pair table needs to be large to accommodate a lot of candidates from PPW. If direct-relation-based mechanism and linear-relation-based mechanism find the same producers for FMLs, there is no reason to use linear- relation -based mechanism over direct-relation-based mechanism. This is because direct- relation mechanism is relatively cheaper when it comes to space and energy consumption in the candidate pair table.

Both mechanisms access the PPW. Linear-relation-based mechanism inserts all instructions

in the corresponding dependence chain into the candidate pair table, but direct-dependence-based mechanism stops searching if a producer is found and inserts only one producer into the candidate pair table. To combine these two mechanisms, PPW is scanned as described for the linear-dependence-based mechanism, but instructions on the dependence chain are temporarily stored in a small queue. If direct relation is not found, the instructions in the queue are moved to CPT. If direct relation is found, the queue is flushed and only the selected producer is inserted into CPT.

This filtering mechanism also serves as a way to combine the two mechanisms in order to benefit from both, obtaining the better IPC of the two mechanisms.

4.5. Evaluation

4.5.1. Speedups

We now compare the performance of linear-relation-based prefetching, direct-relation-based prefetching, filtered-linear-relation-based prefetching, and perfect prefetching. With the filtered-linear-relation-based prefetching, producers are identified with linear relation mechanism only if direct relation mechanism fails. For this experiment, simulation parameters described in Table 3.1 (page 36) are used, meaning practically achievable performance benefits are presented. As the extra hardware required for the linear-relation-based prefetching mechanisms, a 16-entry candidate pair table specifically for the linear mechanisms and one divider with a 32-entry divider queue are used. The partition size used for JPT in L2 cache varies to meet different requirements of different benchmarks. For example, benchmarks that require big JPT such as *health* use 4 ways of L2 cache, which commensurate 1MB of space. (see Chapter 5 for a discussion of how L2 cache is used to store jump pointers.) For benchmarks that do not need any jump pointers, the whole L2 cache is used

conventionally. Figure 4.4 to figure 4.7 show IPCs of 16 benchmarks with four different L2 cache sizes: 1MB, 2MB, 4MB, and 8MB. The IPCs in each graph are normalized to its base case, where prefetching is not performed.

In each graph, the first bar represents the IPC without prefetching. The next three bars represent the IPCs of the three proposed prefetchings: direct-dependence-based prefetching (direct), linear-dependence-based prefetching (linear), and direct-dependence-filtered linear-dependence-based prefetching (f.linear). The last three bars are for the original dependence-based prefetching (DBP) [43], original jump pointer prefetching (JPP) [44], and tag correlating prefetching (TCP) [16] respectively.

Let's look at the results with 1MB L2 cache first (figure 4.4). For *bzip2*, *vpr*, *health*, *art*, and *equake*, linear-relation-based prefetching outperforms direct-relation-based prefetching. For *bzip2* and *vpr*, some of the producer loads have linear relations with frequently missing loads. Therefore, they are able to issue prefetch requests that cannot be issued with direct-relation-based prefetching, which is not able to identify those producers. For *health* and *art*, both prefetching mechanisms can identify producers but linear-relation-based mechanism finds better producers. For example, with direct-relation-based prefetching some consumers are prefetched via chain prefetching because their producers do not have any usable patterns but the producer of the immediate producer have usable patterns. However, linear-relation-based prefetching identifies producers with usable patterns for these consumers. Since normal prefetching is better than chain prefetching in terms of prefetch timing, this leads to performance increase with linear-relation-based prefetching. For *equake*, linear-relation-based prefetching finds arithmetic instructions as producers that cannot be identified with direct-relation-based prefetching. The producers have linear relations with their consumers and have very stable stride patterns. The consumers' addresses are generated by the producers, and then next a few instances of the consumers use the same addresses before new addresses are generated by

the producers. This behavior makes plain stride prefetching for the consumers impossible.

Except for *swim* and *wupwise*, benchmarks other than the above ones show no difference between direct-relation-based prefetching and linear-relation-based prefetching. For *swim* and *wupwise*, the overhead in computing prefetch addresses and the relatively slow speed of learning producer-consumer pairs outweigh performance gain from linear-relation-based prefetching.

The filtered-linear-relation-based prefetching behaves inconsistently. For some benchmarks, it follows the performance of its linear-relation-based counterpart, but for some other benchmarks, it follows the IPC of its direct-relation-based counterpart. For the benchmarks that benefit from linear-relation-based prefetching, *art* and *health* follow the performances of their direct-relation-based prefetching. This is because the producers found by direct-relation-based mechanism can generate useful prefetches even though they are not as effective as the ones found by linear-relation-based mechanism. It is hard to know whether linear-relation-based mechanism is able to find a better producer if the one by direct-relation-based mechanism performs fine. For the benchmarks that are negatively affected by linear-relation-based prefetching, filtered-linear-relation-based prefetching can reduce the gap between linear and direct mechanism.

If we compare the results from different cache sizes, the relative performances of the prefetching mechanisms do not change significantly. The linear-dependence-based mechanism performs the best and the other proposed mechanisms follow. Tag correlating prefetcher is the best among the existing prefetchers. For some benchmarks such as *bzip2* and *health*, the performance improvement decreases as the L2 cache size increases. On the other hand, for some benchmarks such as *equake* and *swim*, the performance improvement stays the same regardless of the L2 cache size. These behaviors will be explained in the next section.

One anomaly is that the performance of the JPP with 8MB L2 cache is better than those of the proposed mechanisms. This is because the proposed mechanisms use the L2 cache for

implementing the jump pointer table but the JPP has a separate jump pointer table.

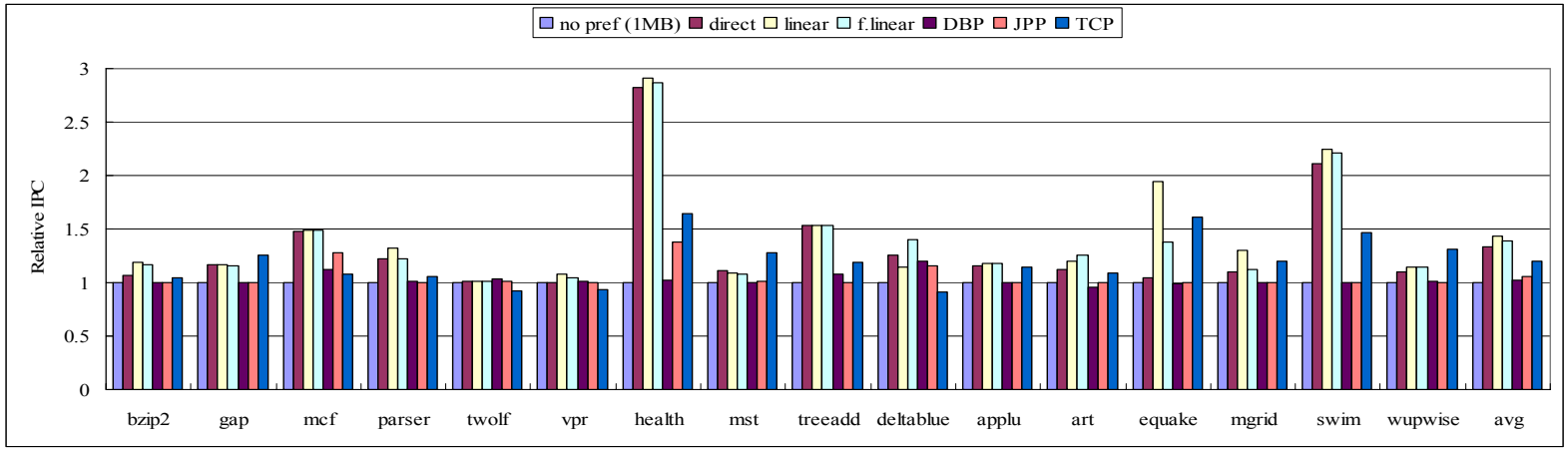


Figure 4.4. Performance comparison with 1MB L2 cache

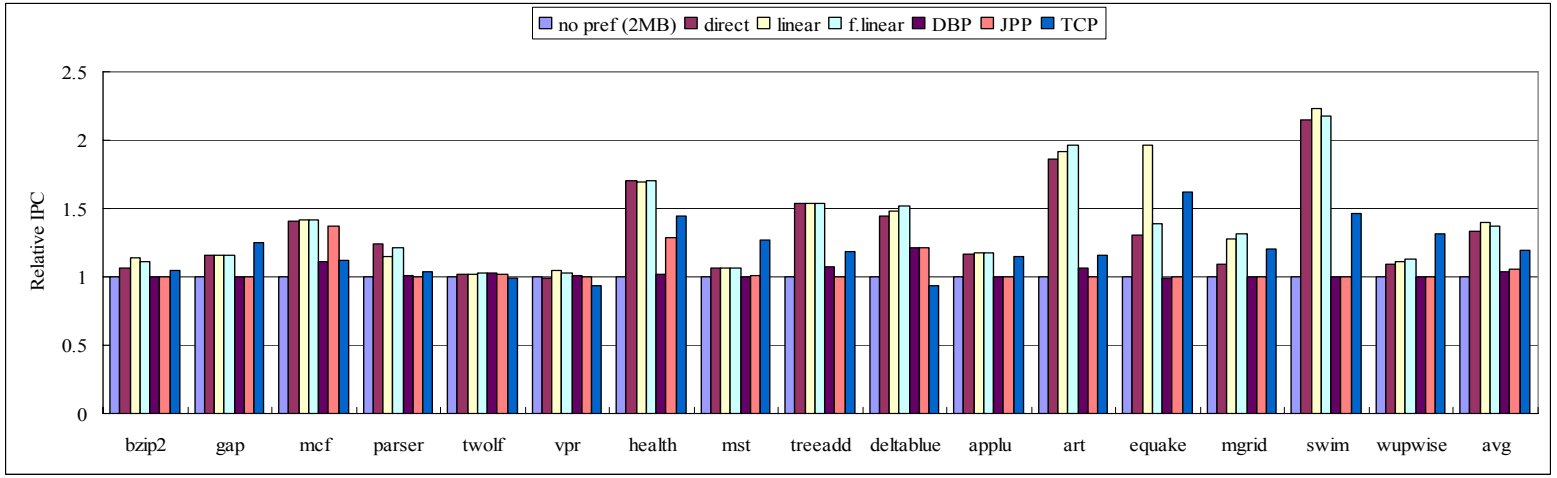


Figure 4.5. Performance comparison with 2MB L2 cache

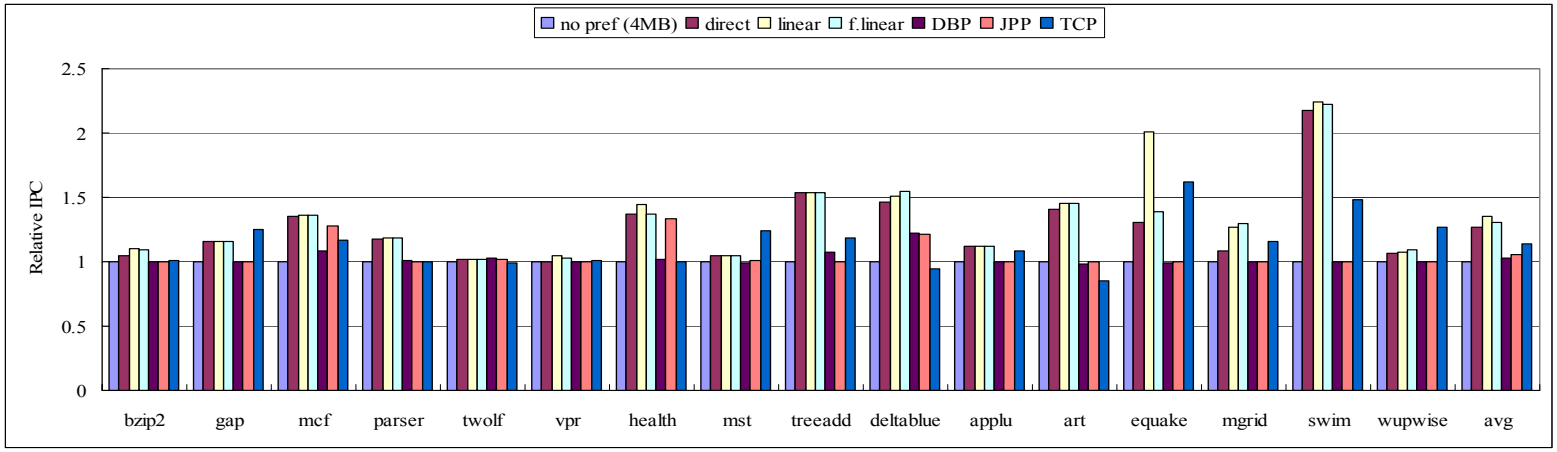


Figure 4.6. Performance comparison with 4MB L2 cache

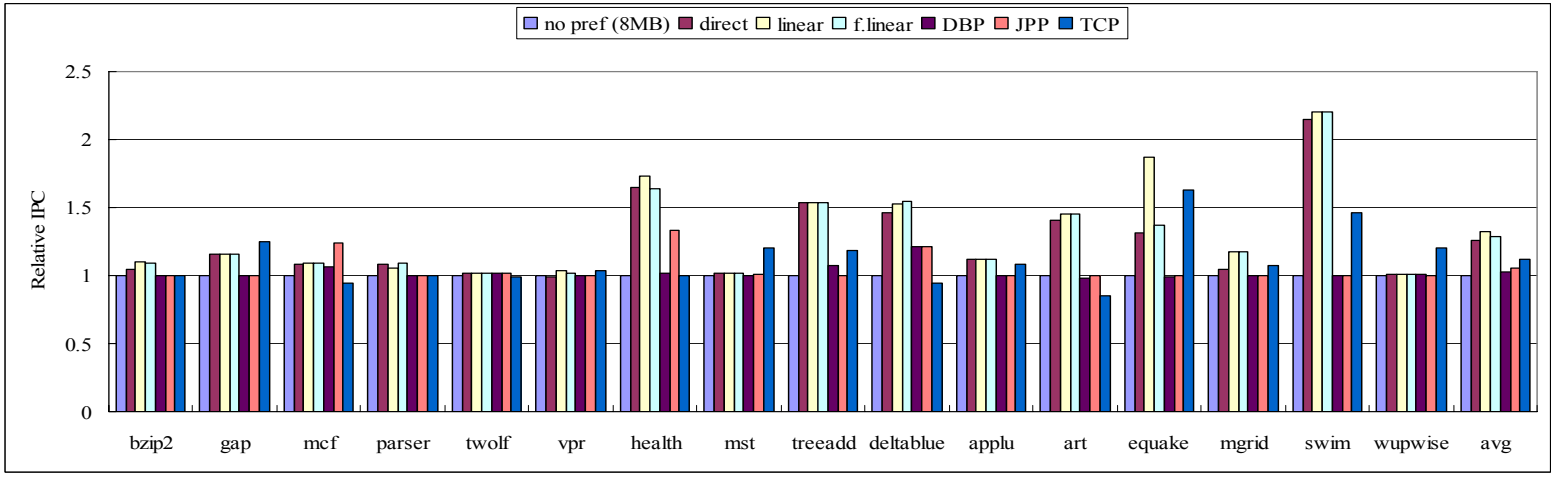


Figure 4.7. Performance comparison with 8MB L2 cache

4.5.2. Sensitivity to the size of the L2 cache

In this section, the sensitivities of the proposed prefetching mechanisms to the size of the L2 cache are presented. As the L2 cache size grows, the base performance without prefetching either improves or stays the same. It is interesting to see how the prefetching mechanisms influence the performance in two different cases.

It is good to see how the L2 cache size affects the base performance first. Figure 4.8 shows the impact of the L2 cache size on the performance. There are two groups, each of which has a different behavior. Except for *gap*, *treeadd*, *deltablue*, and *swim*, all the benchmarks show improvement as the L2 cache size increases. *Deltablue* can fit in 1MB L2 cache, so that no more improvement is obtained with the bigger cache sizes. For gap, treeadd, and swim, their L2 cache miss rates are not low, and do not change as the L2 cache size grows.

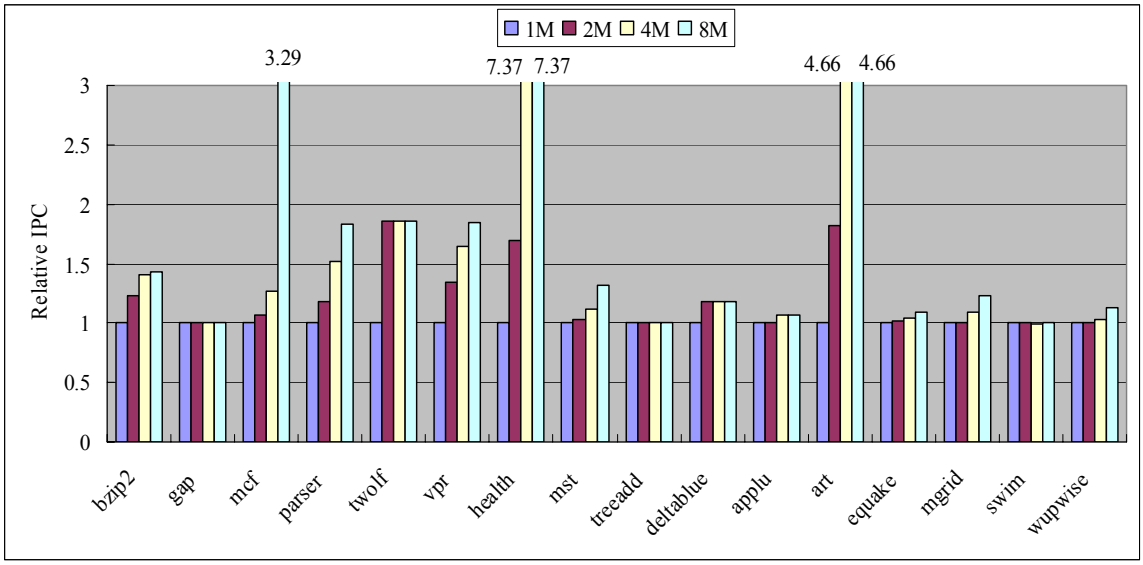
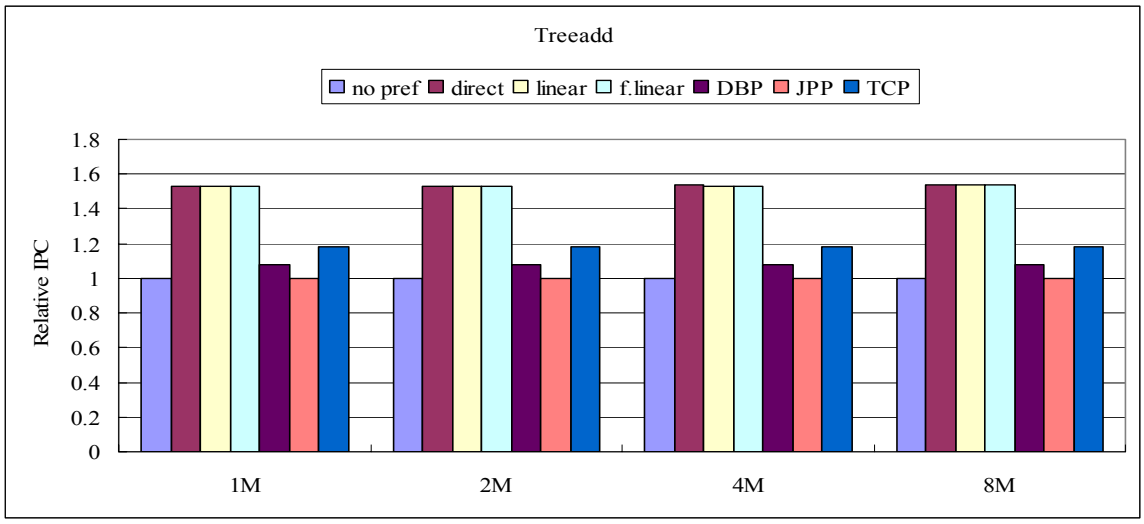
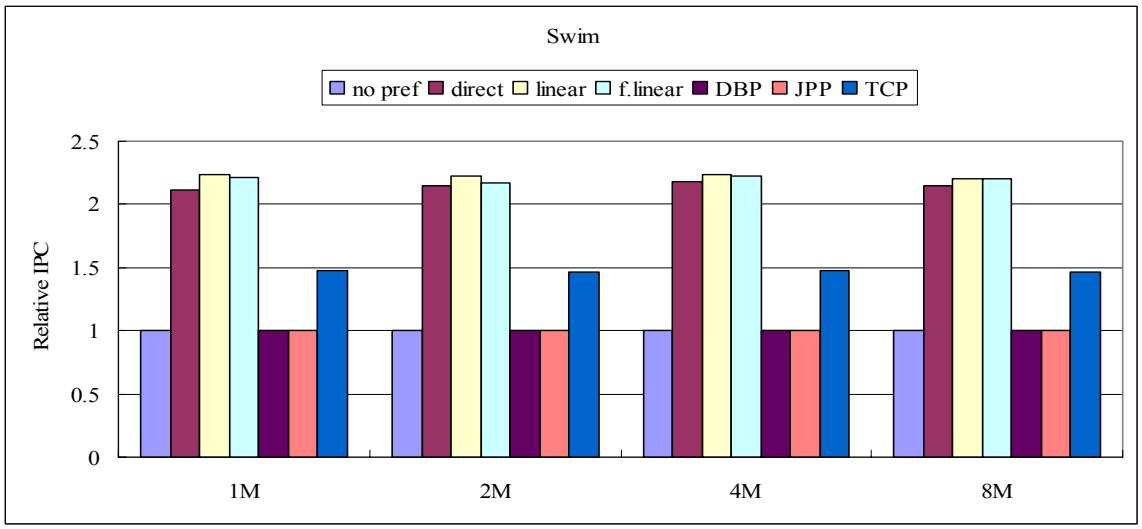


Figure 4.8. Impact of the L2 cache size on the performance

Instead of showing for all benchmarks, only the representative benchmarks from each group are presented. For the group that is insensitive to the L2 cache size, *treeadd* and *swim* are presented. For the other group, *bzip2* and *art* are selected. Figure 4.9 shows the performance improvement results for the first group.



(a) *Treeadd*

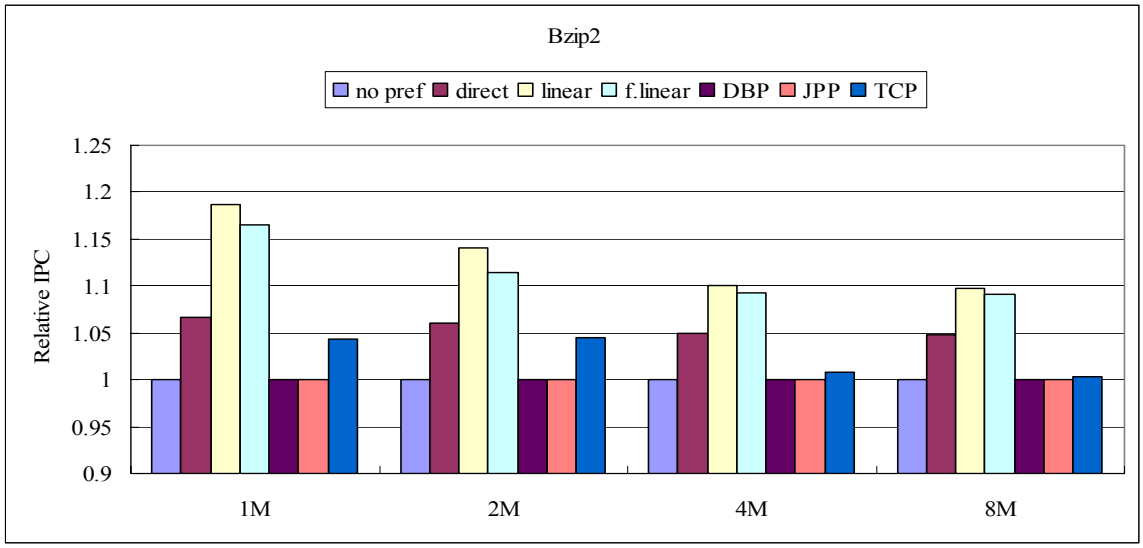


(b) *Swim*

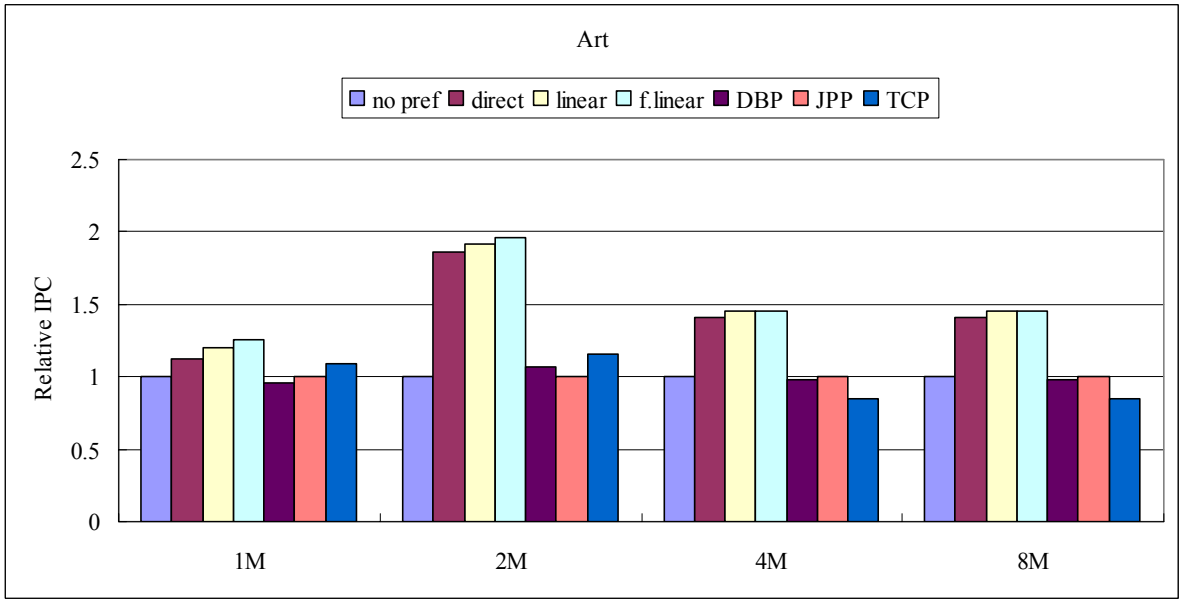
Figure 4.9. Performance improvements for *treeadd* and *swim*

Each graph shows the performance improvements for four different cache sizes. For each cache size, there are six bars, each of which corresponds to a prefetching mechanism. The bars for a cache size are normalized to the IPC of the no prefetching case of cache size. As shown in the graphs, all of the prefetching mechanisms can improve the performance by the same amount for all of the cache sizes. This tells us that a larger cache size is not able to improve the demand fetch as well as prefetch. The effectiveness of the proposed mechanism could have been further enhanced if all of the prefetched blocks are found in the L2 cache. But at least for benchmarks, this is not happening. The other benchmarks in the first group show similar behavior to *treeadd* and *swim*.

Figure 4.10 shows the benchmarks that belong to the other group that can benefit from a larger L2 cache size. The same formatting as figure 4.9 is used. For *bzip2*, all of the prefetching mechanisms show diminishing returns as the L2 cache size grows. This is because prefetching mechanisms cover more L1 cache misses that can be resolved by the L2 cache, and fewer L1 cache misses that are resolved by the main memory. A prefetching mechanism is most effective if it removes a L1 cache miss that needs to travel down to the main memory. If it removes a L1 cache miss that can be satisfied by the L2 cache, the amount of benefit is relatively smaller than removing a L1 cache miss resolved by the main memory. For *art*, a different behavior is observed when the L2 cache size changes from 1MB to 2MB. Even though the effectiveness of the prefetching mechanisms is reduced for the same reason as the one for *bzip2*, the timeliness of the prefetching is also improved. This results in the higher performance gain with 2MB L2 cache than with 1MB L2 cache. In this case, the improved effectiveness due to better timeliness outweighs the reduced effectiveness due to covering more L1 cache misses resolved by the L2 cache. The other benchmarks exhibit the similar behavior to *bzip2*, showing diminishing returns as the L2 cache size increases.



(a) *Bzip2*



(b) *Art*

Figure 4.10. Performance improvements for *bzip2* and *art*

4.5.3. Sensitivity of the correlation table to ports and access latency

A linear-relation-based prefetching is likely to be more demanding on the number of port of the CT, because not only loads but also all output-producing instructions update and look up the CT. Figure 4.11 shows the sensitivity of the CT to the number of ports, when the linear-relation-based prefetching is used.

The number of ports is varied from one read and one write port up to eight read and eight write ports. To measure the influence of modeling the ports, a result without modeling the ports is included. Also, a case with a set of small buffers, which temporarily stores accesses to the CT, is presented. The y-axis represents relative IPCs normalized to the IPC of the case without modeling the ports.

Gap, *parser*, and *twolf* seem to be insensitive to the number of ports. Mst and treeadd shows a little decrease in IPC, when there are only one read port and one write port. But two read ports and two write ports seem to be sufficient to remove the gap with the case where the ports are not modeled (ideal case). *Mcf*, *health*, and *deltablue* are the most sensitive benchmarks to the number of ports. One read and one write port causes significant decrease in IPC, but adding small buffer improve the IPC. Two read ports and two write ports are again enough for these benchmarks.

Read ports are likely to be more critical because issuing prefetches (lookup) is more crucial than learning relations and patterns (update). Once producer to consumer relations and producers' patterns are identified, updates are of no importance unless producer patterns are changing. However, lookup requests are lost, it directly influence the number of prefetches , decreasing the benefits of prefetching.

Figure 4.12 shows the sensitivity of the CT to the access latency, when the linear-relation-based prefetching is used. The latency is varied from 2 cycles to 8 cycles. This is the latency for

reading a producer and its consumers. Y axis represents relative IPCs normalized to the IPC when access latency is set to 2 cycles.

It is observed that all of the benchmarks are not sensitive to the access latency of the CT. This is similar result to the one for direct-relation-based prefetching. Once producer to consumer relations are learned, there is no difference between a direct-relation-based mechanism and a linear-relation-based mechanism.

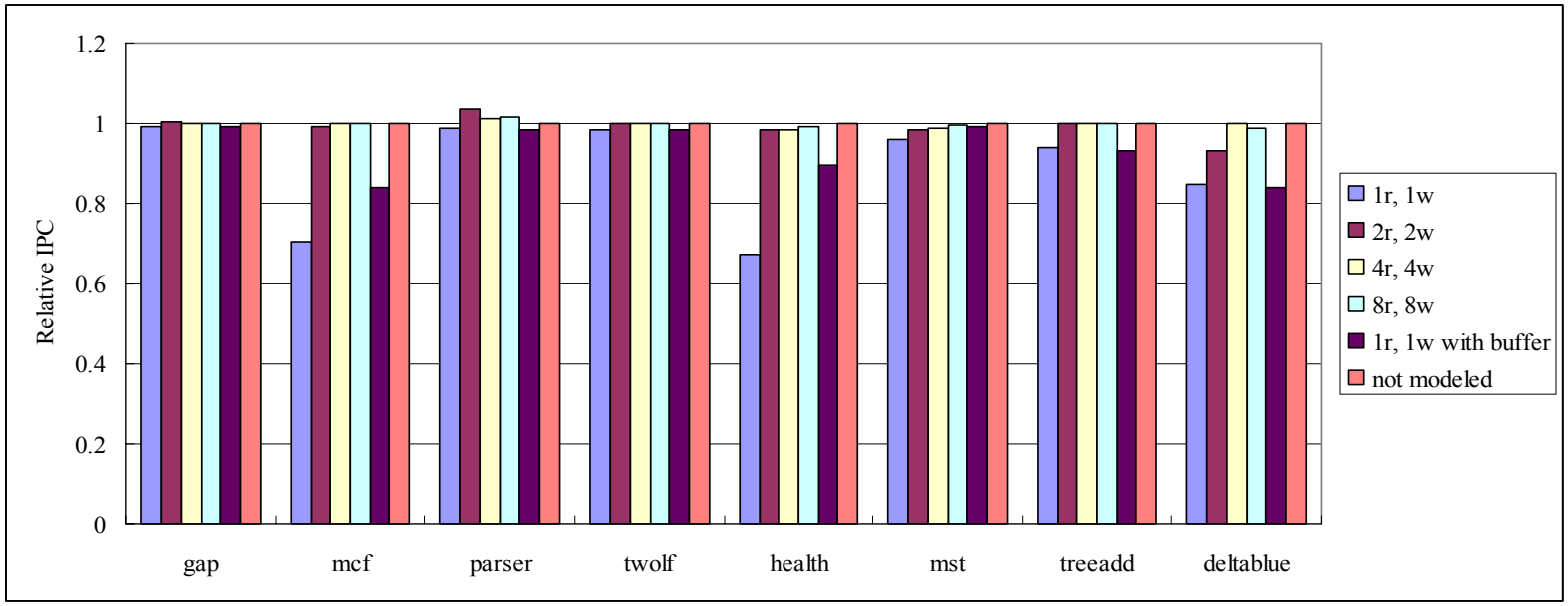


Figure 4.11. Sensitiveness of the CT to the number of ports

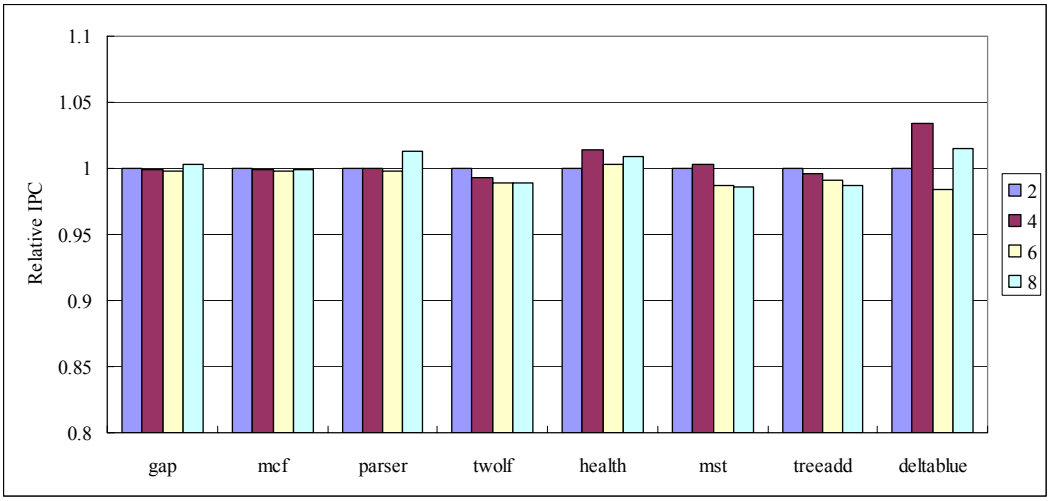


Figure 4.12. Sensitiveness of the CT to its access latency

4.5.4. Sensitivity of the potential producer window to ports and access latency

If only the read ports on the potential producer window (PPW) are concerned, there is no difference with the results for the direct-relation-based prefetching mechanism. This is because the PPW is accessed by consumer loads and the number of consumers with the linear-relation-based prefetching is not significantly different from that with the direct-relation-based prefetching. For more details about how the PPW is modeled, refer to section 3.5.6.

Figure 4.13 shows the sensitivity of the PPW to the number of read ports on it. Y axis represents relative IPCs normalized to the IPC of the case where the number of read ports is set to one. X axis is the number of ports on the PPW, and the number varies from one to four. As expected, none of the benchmarks is sensitive to the number of ports on the PPW. This is because consumer to producer matching doesn't occur frequently due to the filtering mechanism and updating the CT is not in a critical path.

As for the access latency of individual element in the PPW, there is a subtle difference between the direct-relation-based mechanism and the linear-relation-based mechanism. With the

linear-relation-based prefetching mechanism, more PPW entries are likely to be read because the direct-relation-based mechanism stops searching if a producer with the direct relation to the consumer is found. But with the linear-relation-based mechanism, one producer cannot be identified because source and destination register numbers are used for identifying producers instead of producers' loaded values and consumers' addresses. The linear-relation-based mechanism searches the PPW until a predefined number of potential producers are found or another instance of the consumer is encountered.

Figure 4.14 presents the sensitiveness of the PPW to its access latency. Y axis represents relative IPCs normalized to the IPC of the case where the access latency is set to one. X axis represents the access latency of the PPW, and the latency varies from one cycle to four cycles. Unlike the expectation, all of the benchmarks are insensitive to the access latency of the PPW. The same reasons as the number of port experiment apply. Comsumer to producer matching doesn't occur frequently enough to have the access latency influence overall IPC, and updating CT is not in a critical path.

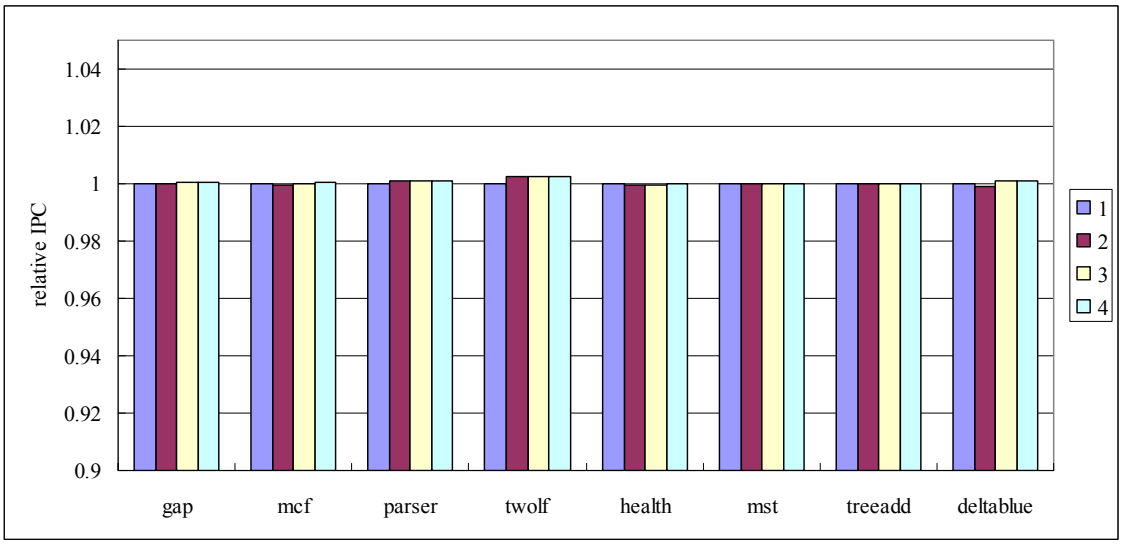


Figure 4.13. Sensitiveness of the PPW to the number of ports

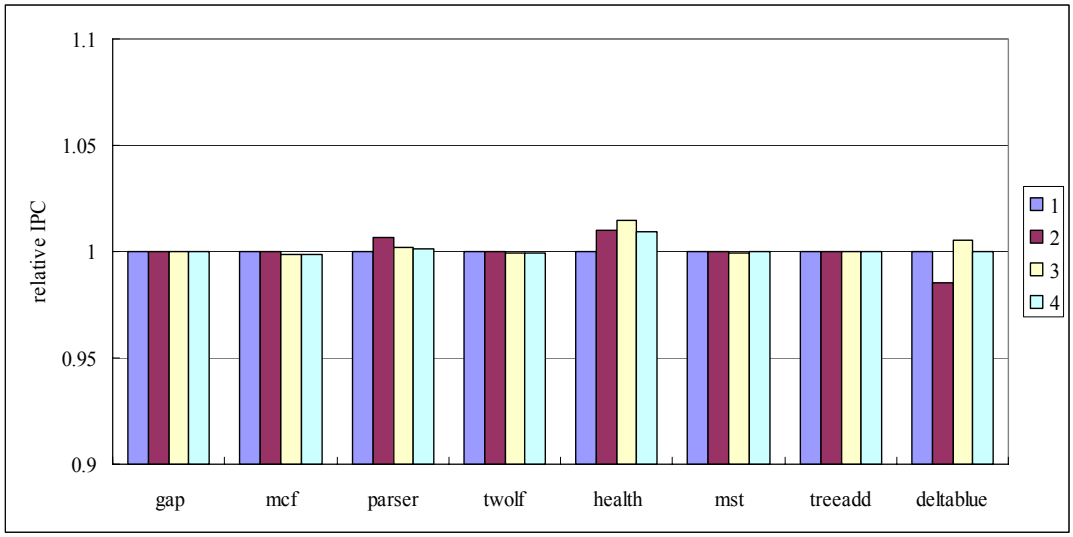


Figure 4.14. Sensitiveness of the PPW to its access latency

4.6. Conclusion

This chapter introduces a new way to identify producers in dependence-based prefetching by using linear relations. Identifying producers is the very core task in dependence-based prefetching, and it determines the overall effectiveness of the prefetching mechanism. By utilizing linear relations between producers and consumers, more consumers can find their producers and this leads to performance improvements. However, in order to support this mechanism, hardware complexity is elevated. In particular, the candidate pair table, where candidate producer-consumer pairs are temporarily stored during learning phase, is likely to be a performance bottleneck. However, we found that a small candidate pair table works as well as much larger table by making only one consumer occupy the table at a time.

To minimize the addition to hardware complexity, a filtering mechanism is proposed. Direct relation filtering mechanism makes linear-relation identification mechanism applied only to the consumers that the direct-relation identification mechanism fails to find producers for. This way

linear relation identification mechanism is used only when it is beneficial, reducing the hardware complexity required for implementing it. But for applications with many linear relations such as *bzip*, the reduction may not be possible in order to fully exploit the potential of linear-relation-based mechanism.

Chapter 5

Using L2 Cache for Storing Prefetch Table

5.1. Motivation

For a hardware prefetching mechanism to be implemented in commercial microprocessors, the mechanism needs to be simple yet effective. Unfortunately, there is no single prefetcher that works well with a wide range of applications. For example, stride prefetcher works well with applications that have regular memory access patterns, but not well with applications that have LDS. Correlation-based prefetchers are effective for applications with LDS, but they usually require large tables to record history information. Combining multiple prefetchers can improve overall prefetch effectiveness, but the high space requirement is hard to overcome. The idea of cramming a large table structure in a processor core may not be appealing to the microarchitect, because of power consumption as well as space consumption. It is possible for a prefetch table to be almost empty for some applications, because the mechanism is not performing well with the applications.

Software-based prefetching [29, 57] is an alternative to hardware-based prefetching, but it has two major drawbacks. First, the compiler lacks run-time information that hardware-based prefetchers are able to extract easily. Second, software overhead is not trivial. Prefetch

instructions must be supported, resulting in modification to instruction set architecture (ISA), and these instructions tend to increase program size.

There are proposals that move the prefetch table into main memory. The most recent work is epoch-based correlation prefetching [11]. At high off-chip latencies, overlappable off-chip accesses to main memory appear to issue and complete together. Due to this behavior, program execution separates into recurring period of on-chip computation followed by off-chip accesses. Each pair of on-chip and off-chip access periods is called an epoch. The correlation table is stored in main memory and the first miss address in the current epoch initiates accessing the table and the table predicts miss addresses in the epoch a few hops from the current epoch. This is unavoidable because accessing the table takes as long as main memory latency. To hide the large access latency to the table, some of the epochs that overlap with table accesses cannot be prefetched, or the distance between current epoch and the epoch the correlation table predicts addresses for should be long. However, longer distance means weaker correlation, resulting in poor address prediction accuracy.

There is also a study that moves the instruction reuse buffer into L1 cache. Ranganathan et al. [41] leveraged a reconfigurable cache to divide L1 cache into multiple partitions, and each partition was used for different purposes. The idea of dividing up a cache is first introduced by Albonesi [1]. A subset of the ways in a set associative cache can be disabled during periods of low cache usage for energy savings. This approach utilizes the subarray partitioning already present for performance reasons, and requires only minor modification to a conventional cache. Therefore, it doesn't compromise performance at all.

Ranganathan et al. used a partition for instruction reuse targeting media processing applications. The motivation for using cache was based on the claim that architectural features that use a significant fraction of the on-chip transistor budget should benefit wide range of applications. They presented two partitioning methods: associativity-based partitioning and overlapped wide-tag

partitioning. They also analyzed the impact of reconfigurability to access time, and claimed that the impact was negligible.

Recently, Burcea et al [5] proposed a technique called predictor virtualization (PV) that used the existing memory hierarchy to emulate large predictor table. Any kind of predictor table can be virtualized, so that this scheme promotes the implementation of such prediction mechanisms by break on-chip resource limitations. To maintain the illusion of a large and dedicated table, PV delivers predictor entries on-demand to a small cache that is tightly-coupled to the processor. They maintain a special per-core register that stores the staring address of the table, so that a memory address for a table entry can be computed based on the value in the register.

In this study, an alternative to both software approach and main memory approach is examined. The alternative is using L2 cache, an already-existing hardware structure, for storing prefetch table [25]. It may sound counter-intuitive to reduce the effective size of L2 cache for better memory performance. Memory intensive applications are likely to benefit more from prefetching, but they also tend to benefit from bigger cache. It sounds like a dilemma because those applications require large prefetch tables, but giving too much L2 cache partition to prefetch table means smaller L2 cache space for conventional L2 cache.

However, after examining the idea by varying L2 cache size and prefetch table size in L2 cache, it is found that it is possible to implement prefetch table in L2 cache. We found a partition size dedicated for prefetch table in L2 cache, where improvement from prefetching outweighs performance loss by reduced effective L2 cache size. One factor that makes this possible is the reduced number of L1 cache misses, which is the outcome of the prefetching mechanism that targets L1 cache misses.

5.2. Reconfigurable cache

Ranganathan et al. [41] propose reconfigurable caches, in which multiple partitions exist, and each partition is used for a different processor activity. Our reconfigurable cache implementation is based on this work. They used associativity-based partitioning, which divides the reconfigurable cache into partitions at the granularity of the ways of the conventional cache, utilizing the conceptual division into ways already present in a conventional cache. For example, a 4-way 1MB cache can be reconfigured to 4 partitions of 256KB each.

Figure 5.1 shows the reconfigurable cache structure. The dark elements are added in order to support multiple inputs and outputs. The reconfigurable cache is able to accept N input addresses and generate N output data elements with N hit/miss signals (one for each partition). A special hardware register called cache status register is employed to track the number and sizes of the partitions and control the routing of the N inputs, outputs and signals.

They also analyze the impact of reconfigurability on cache access time using the CACTI analytical model [42]. They found that the overall microprocessor cycle time and cache access latency are not affected.

A reconfigurable cache requires mechanisms that ensure correct and effective reconfiguration. One such mechanism is the one that maintains data consistency. After reconfigurations, the data belonging to a particular processor activity should reside only in the partition associated with that particular activity. Other mechanisms include how often a reconfiguration occurs and how to initiate a reconfiguration. Repartition may occur either infrequently (e.g., in the beginning of an application) or frequently (e.g., at the beginning of a loop). Repartition may be initiated by software or hardware. A software approach makes the cache status register visible to the compiler, and the compiler makes reconfigurations invoked at the appropriate points in the program. Alternatively, hardware approach

monitors performance, and dynamically decides when and how to change partitions.

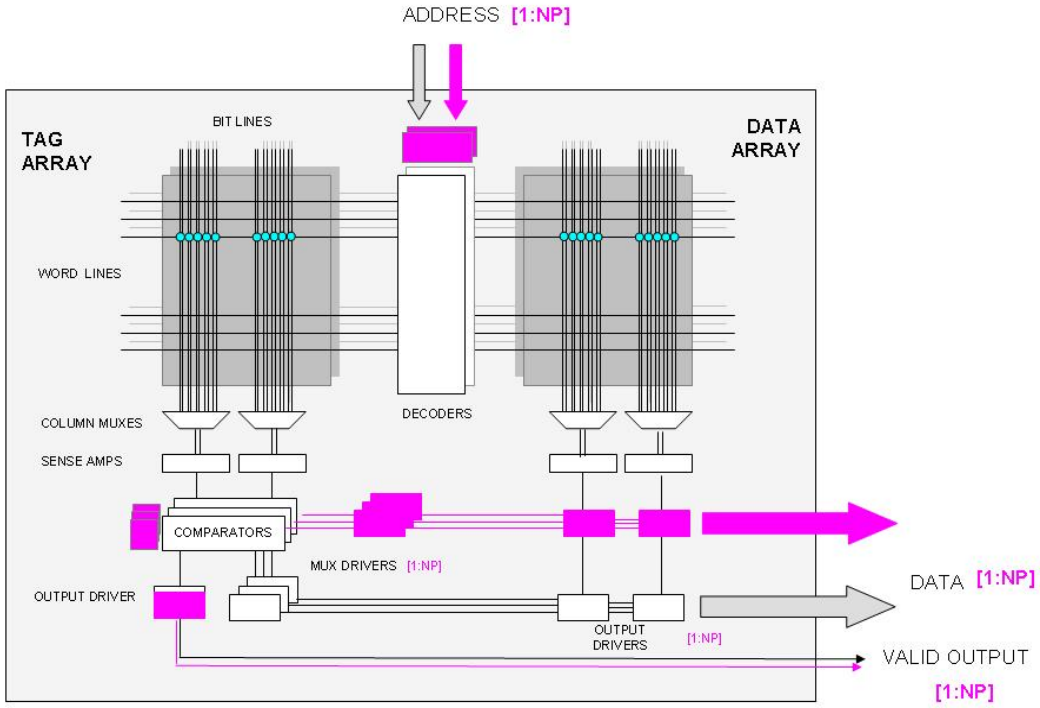


Figure.5.1. Implementation of a reconfigurable cache. [40]

5.3. Implementing prefetch table in L2 cache

With the reconfigurable cache, a space can be reserved for a prefetch table such as a jump pointer table (JPT), but how to use the reserved space also needs to be sorted out in order for a prefetch table to be implemented in the L2 cache. Basically, the following two questions have to be answered. First, how do we pack correlations (jump pointers) in a 64 byte cache line? Second, how do we access an individual correlation (jump pointer) for the given structure of correlations (jump pointers) in the L2 cache? In this section, implementing a JPT in the L2 cache is selected for demonstration.

Storing a jump pointer in a cache line is overly inefficient, because a jump pointer takes about 16 bytes including a tag and a jump pointer, if a 64-bit address is assumed. To make the most of the 64-byte cache line, four jump pointers are stored in each cache line. Since four different jump pointers are stored per cache line, one tag associated with the cache line is not enough. Therefore, a tag for each jump pointer is stored in the cache lines along with the corresponding jump pointer.

Since tags are embedded in cache lines, it is not possible to do tag matching for each jump pointer in a set-associative JPT. Hence, the entire set is read from the L2 cache, and then selecting the right jump pointer is performed outside the L2 cache. Figure 5.2 shows the mechanism for accessing jump pointers in the L2 cache.

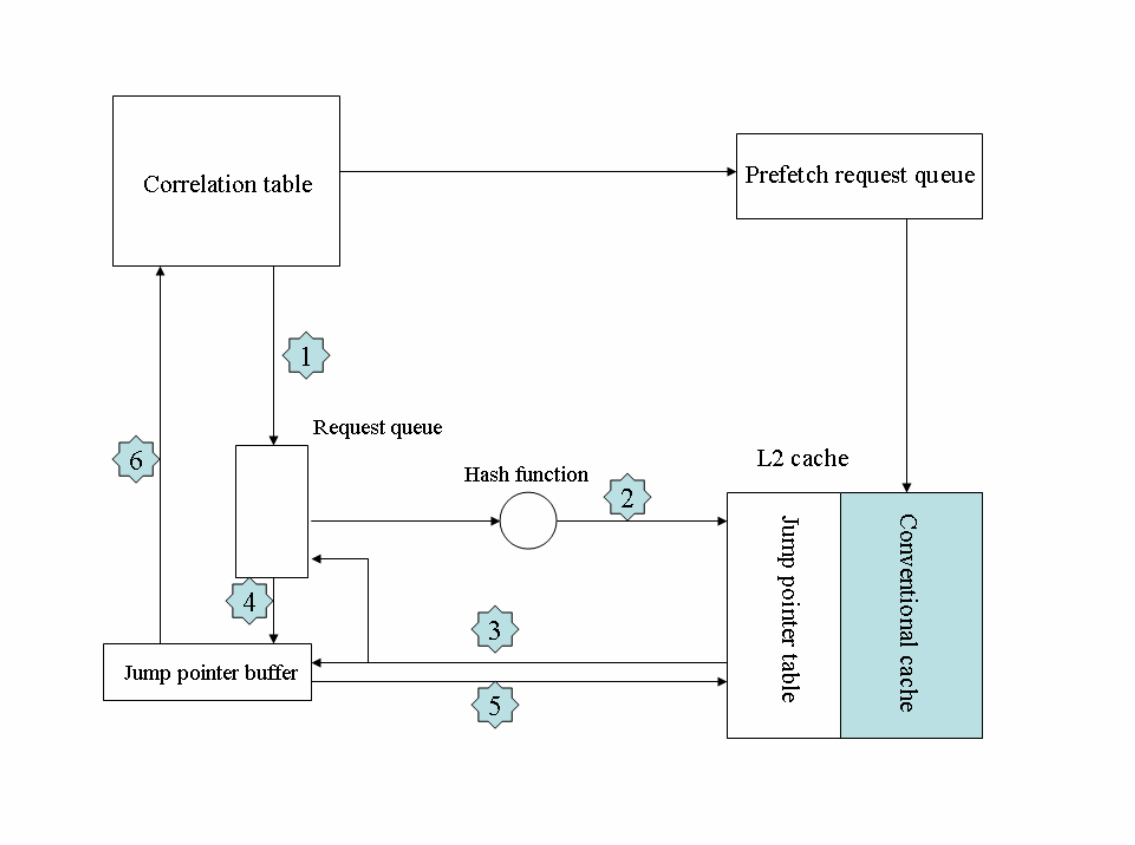


Figure 5.2. Implementing a jump pointer table in the L2 cache

As shown in the figure, a hash function is responsible for mapping addresses from the correlation table (CT) to addresses, with which the JPT is actually accessed. The purpose of the hash function is to uniformly distribute accesses to ways of the JPT. Because of the limited associativity and size of prefetch table implemented in L2 cache, the role of the hash function is critical to prefetching performance.

There are two types of accesses to the JPT: jump pointer update and jump pointer lookup. For each access, a series of actions is performed. Each action is labeled from one to six in the figure. There are two addresses associated with a jump pointer: source address and destination address. For an update access, a jump pointer is first queued in the request queue (action 1). Then hash function generates a new address from the source address of the jump pointer (action 2). The corresponding cache lines are read from the L2 cache and stored in the jump pointer buffer (action 3). When the cache lines arrive, the request queue is also notified of the arrival. With the stored jump pointer in the request queue, an appropriate chunk of the right cache line is updated (action 4). After the update, the update cache line is written back to the L2 cache (action 5).

A lookup access takes the fewer number of actions. The request is stored in the request queue (action 1), and the L2 cache is read by the address generated by the hash function (action 2). The corresponding cache lines are temporarily stored in the jump pointer buffer (action 3), and forwarded to the correlation table (action 6).

Table 5.1 summarizes the design choices that we make for implementing reconfigurable cache. The repartition policy is set to “infrequent”, which means the partitions are set in the beginning of the simulation and never changed. The detection of reconfiguration is assumed to be done by software. Therefore, there is no hardware mechanism that monitors processor or cache status. The reconfiguration point is assumed to be the start of the simulation. Actual implementation of the software-controlled detection and the hardware-controlled detection are not studied, and left for

future work.

Table 5.1. Design Choices for Reconfigurable Cache.

Design issue	Used in this study
Partitioning mechanism	Associativity-based
Address generation	Hardware generated
Data consistency	Cache scrubbing
Repartition policy	Infrequent
Detection for reconfiguration	Software controlled
Reconfigurable cache level	L2 cache

5.4. Evaluation

5.4.1. Impact of the L2 cache size on overall performance

Before implementing a JPT in the L2 cache, it is interesting to know how L2 cache size influences overall performance without prefetching. It gives us the baseline performance for a given L2 cache size before prefetching makes a difference. It may also give us an idea about how to determine the correct partition size for the JPT.

Figure 5.3 shows the impact of the L2 cache size on overall performance. The x-axis represents the L2 cache size, and the y-axis represents IPC. *Mcf* and *health* are selected for this experiment, because they will be used for the later experiments. This is because they are the most demanding benchmarks.

It is observed that as the L2 cache size increases, IPC increases to a point and saturates afterwards. For *mcf*, 8MB is such a point, and for *health*, 4MB is such a point. If we choose two

consecutive points after the saturating point, and assume the smaller one is the size of the partition for the JPT and the bigger one is the size of the L2 cache (e.g., 4MB for the JPT and 8MB for the L2 cache in case of *health*), it is apparent that prefetching will improve the performance. However, if we choose 2MB for the JPT and 4MB for the L2 cache, the benefit from prefetching dependes on the potential of the prefetching mechanism and the IPC decrease due to smaller L2 cache size. To simulate only the interesting L2 cache sizes, we selected 2MB and 4MB L2 cache for *health*, and 2MB and 8MB L2 cache for *mcf*. For these different cache configurations, the partition size for the JPT is varied from 12.5% to 50% of the L2 cache. The results are presented in the next section.

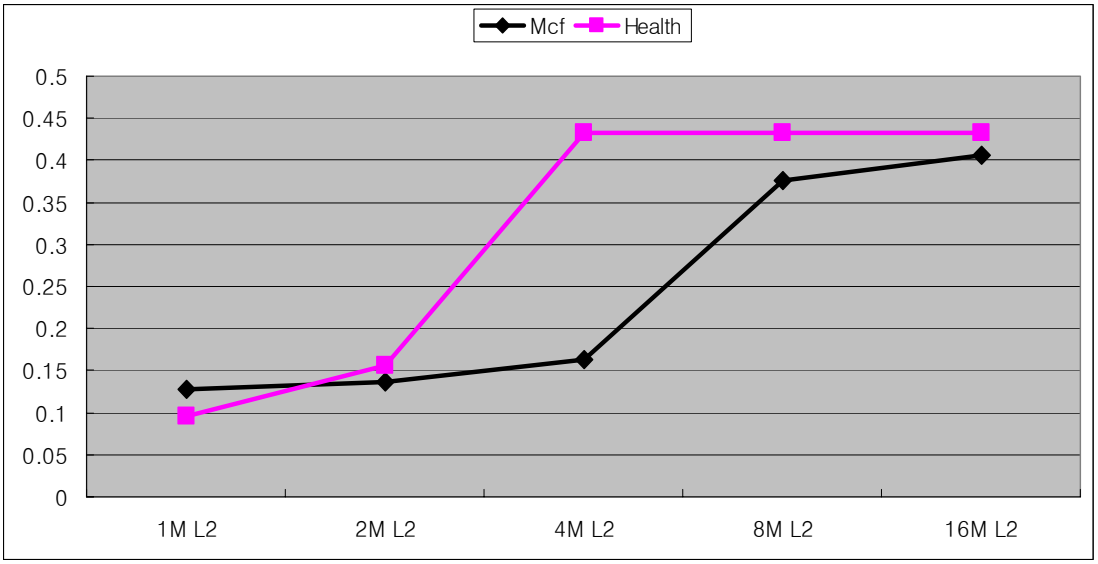


Figure 5.3. The impact of the L2 cache size on overall performance

5.4.2. Performance with varying the partition size for JPT

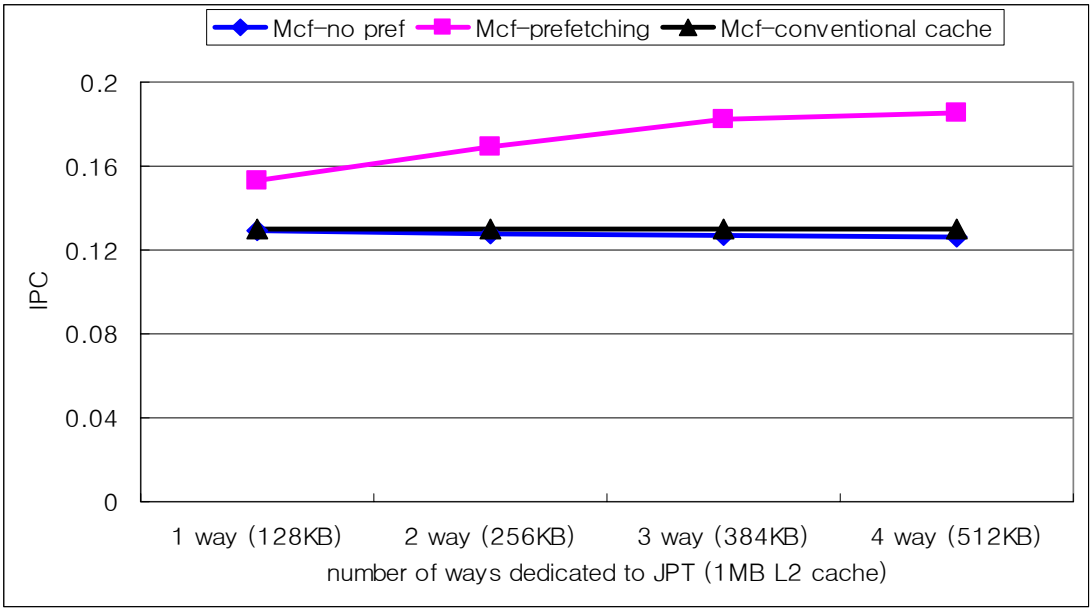
This sub-section shows the results for implementing prefetch tables in L2 cache. Four L2 cache configurations are used: 1MB 8way, 2MB 8way, 4MB 8way, and 8MB 8way. The hit latency of 1MB cache is set to 15 cycles, and all other caches' latencies are set to 18 cycles. In addition to

traditionally cached data, the L2 cache holds the JPT; each 64-byte cache block holds four jump pointer entries. We vary the size of the JPT in L2 cache to see how it influences the overall performance.

In Figure 5.4, there are three graphs. The top one is for *mcf* with 1MB L2 cache, the middle one is for *mcf* with 2MB L2 cache, and the bottom one is for *mcf* with 8MB L2 cache. The x-axis of each graph represents the number of ways reconfigured for storing the JPT, and the y-axis indicates IPC. Since the L2 cache has 8 ways regardless of its size, the 1 way corresponds to 128KB, 256KB, and 1MB for 1MB L2 cache, 2MB L2 cache, and 8MB L2 cache respectively. There are three plots in each graph. The plot labeled "no pref" represents the case where the partition set aside for the JPT is not used and only the other partition of L2 is used conventionally without prefetching. The plot labeled "prefetching" shows the case where the partition for JPT is actually used and prefetching mechanism is put into action. The plot labeled "conventional cache" indicates the case where the whole cache is used conventionally without prefetching.

For 1MB and 2MB L2 cache sizes, IPC drops marginally, as the effective L2 cache size decreases without prefetching. Consequently, if the prefetcher is able to improve the performance more than the IPC decrease, the overall performance increases. This is what is shown in the figure. Note that, in the bottom graph, the IPC increase saturates, when 2 ways (512KB) are reconfigured for storing the JPT. This means that 512KB JPT is big enough to hold all the jump pointers. For 8MB L2 cache, IPC drops significantly, as the effective L2 cache size decreases without prefetching. This is in accordance with the plot in Figure 5.3: there is significant IPC increase in going from 4MB L2 cache to 8MB L2 cache. 8MB L2 cache is selected because of this dramatic behavior. As expected, the IPC improvement due to prefetching is smaller than the IPC decrease owing to the reduced effective L2 cache size except for the case where one way is reserved for the JPT. As mentioned above, there is no more improvement with a bigger JPT size from 512KB JPT. Hence, the

plot for “prefetching” follows the plot for “no pref”.



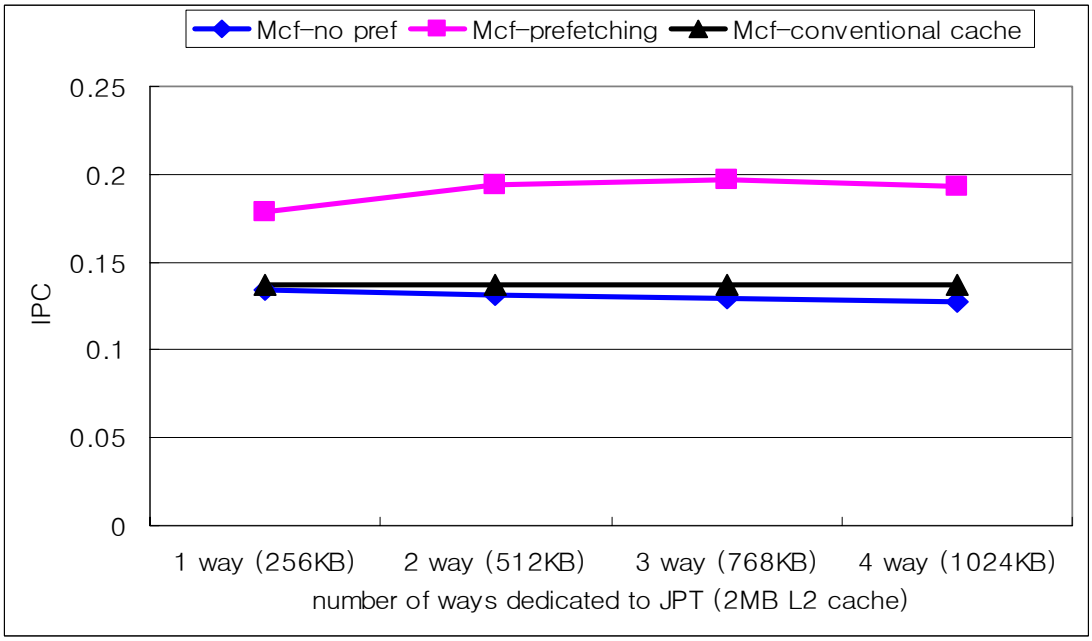


Figure 5.4 continued

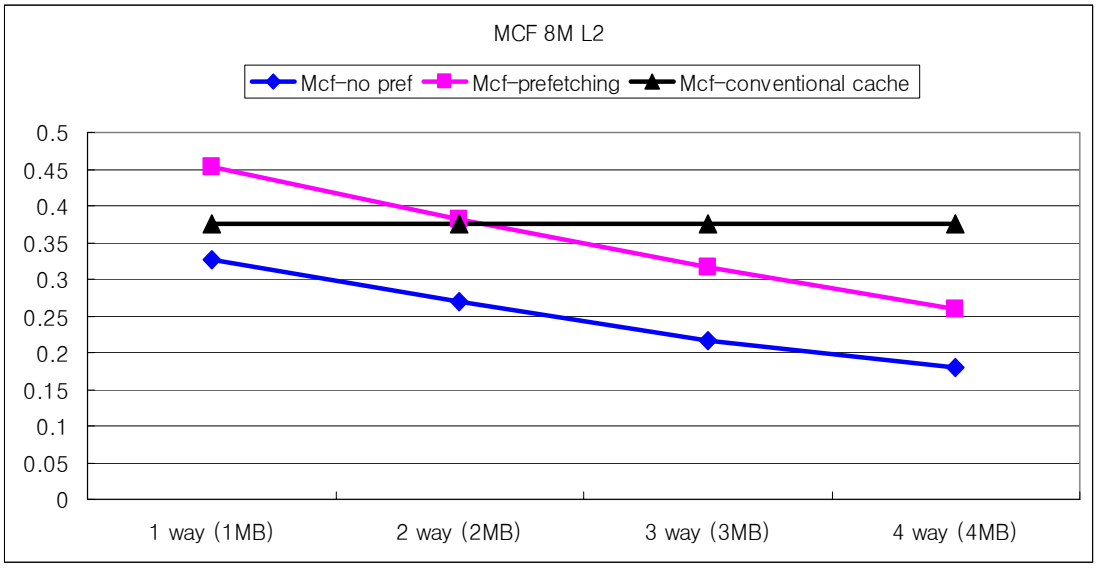


Figure. 5.4. Performance with varying the partition size for the jump pointer table in L2 caches: 1MB L2 Cache (top), 2MB L2 cache (middle), and 8MB (bottom) for *mcf*

Figure 5.5 shows results for *health*. The results for 1MB and 2MB L2 cache are similar to those of *mcf*. IPC increase is much bigger than the decrease due to smaller effective L2 cache size. For *health*, IPC gain saturates when 1MB is used for the JPT as shown in the bottom graph. Therefore, if more than two ways are used for the JPT, overall IPC starts to decreases. However, except for the case where four ways are assigned for the JPT, prefetching improves performance over the whole cache is conventionally used.

Other benchmarks that have recurrent loads are *mst* and *deltablue*. The IPC of *mst* tends to decrease as effective cache size decreases. But 16k jump pointer entries (1 way of 8-way, 2MB cache) is sufficient for it. Consequently, the best partition size for JPT in 2MB cache is 1 way (256KB). IPC with prefetching at the partition size is 1.27, which is just a little bit higher than 1.22, which is achieved when the whole 2MB cache is used conventionally.

For *deltablue*, IPC doesn't decline as effective cache size decreases from 2MB to 1MB. Also, 16k jump pointer entries is enough. Therefore, the best partition size with 2MB cache is any

size between 1M and 2M. Prefetching boosts IPC to 1.59 from 1.12, which is obtained when 2MB cache is used conventionally.

Depending on a benchmark's working set size, the L2 cache reconfiguration point varies. For some benchmarks with limited L2 cache size, it may not be possible to find such a point. However, with the benchmarks used for this experiment, it is possible to identify such points, even though it is impossible to find a reconfiguration point that works well for all benchmarks.

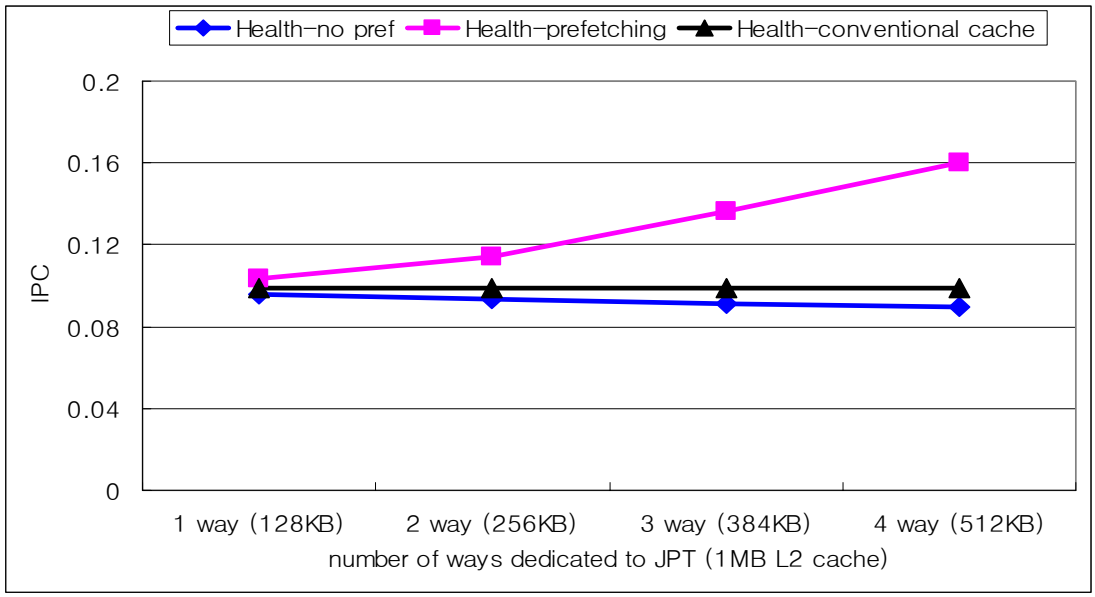
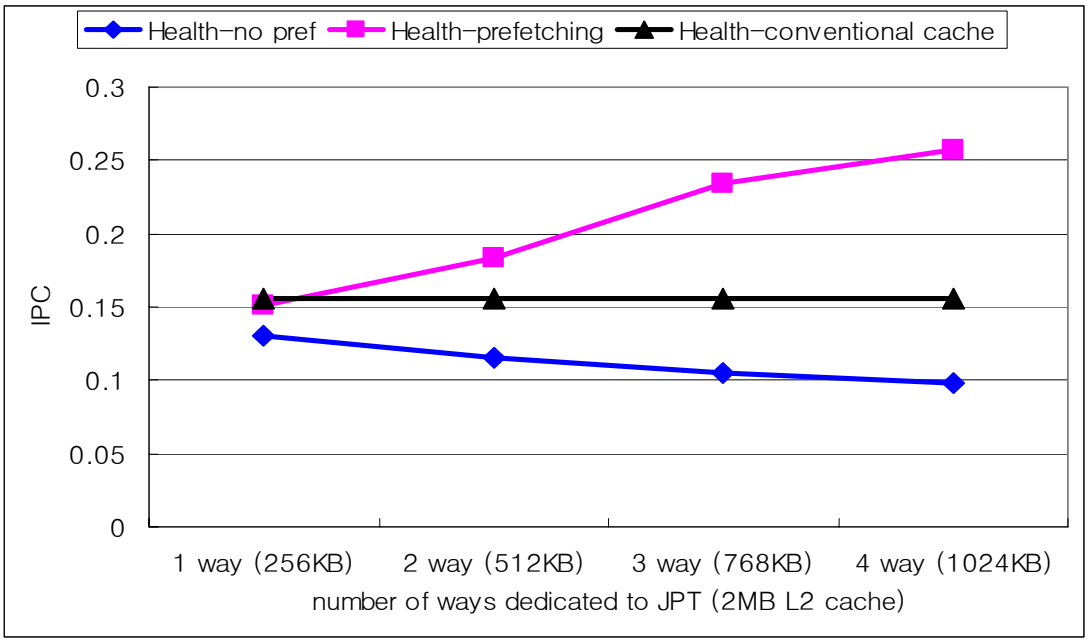


Figure 5.5 continued



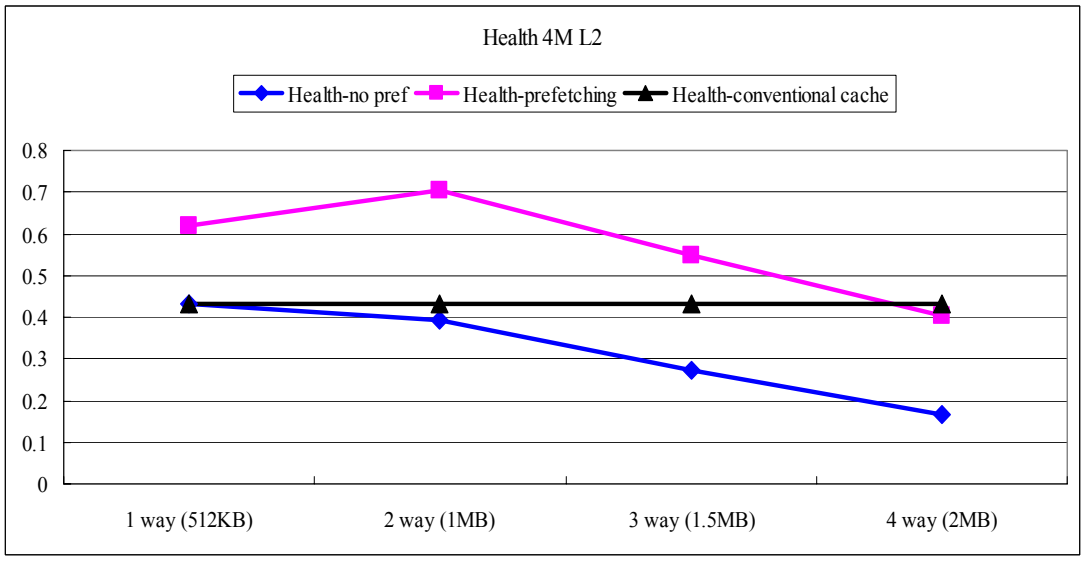


Figure. 5.5. Performance with varying the partition size for the jump pointer table in L2 caches: 1MB L2 Cache (top), 2MB L2 cache (middle), and 4MB L2 cache for *health*

5.4.3. The effectiveness of the hash function

In order to measure the effectiveness of the hash function described in section 5.3., we compare the performance with the hash function and the performance without the hash function. For the hash function, Robert Jenkin's 32bit mix function [19] is used. Figure 5.6 shows the performance comparison.

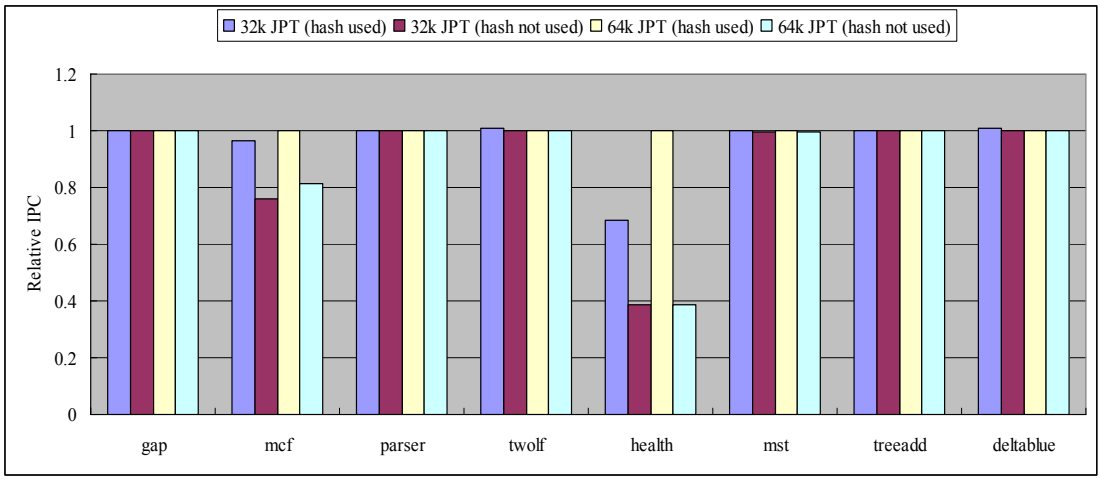


Figure 5.6. The effectiveness of the hash function

There are four bars for each benchmark. The first two bars represent relative IPCs when two way 32k entry JPT is used. The last two bars are for four way 64k entry JPT. The first and the third bars represent the case where the hash function is used, and the others correspond to the case addresses are used to compute indexes. The IPCs are normalized to the IPC of the case where 64k entry JPT and the hash function are used (third bar).

Except for *mcf* and *health*, only trivial differences are observed. *Mcf* and *health* are the ones that most of the benefits are from jump pointer prefetching. For those benchmarks, the hash function plays a crucial role of distributing accesses evenly to the sets in the JPT. Without the hash function,

those benchmarks suffer from a lot of conflicts in the JPT. For *mcf*, going from two way 32k entry JPT to four way 64k entry JPT improves the performance, but for *health*, no improvement is made. This means that *health* has more conflicts in the JPT than *mcf* does.

5.5. Conclusion

Because of the demanding space overhead for the JPT, we investigate using L2 cache to store the JPT. This is based on the observation that having prefetcher is more effective than increasing the size of L2 cache. A certain number of ways are configured to be used for implementing JPT. This way-based approach doesn't require significant additions to the existing cache structure.

Simulation results show that implementing JPT in L2 cache is feasible and beneficial. Reconfiguration decisions, such as how many ways should be assigned for prefetching mechanism, should be based on the target benchmarks' memory access characteristics. It is not true that benchmarks that have high demand on L2 cache are not likely to benefit from prefetching that store JPT in L2 cache. For example, *health* and *mcf* benefit a lot from prefetching but do not fit in L2 cache. Its performance improves because L2 cache is more effectively utilized and L1 cache misses are eliminated. Using L2 cache for storing prefetch tables is a good match with a prefetch mechanism that eliminates L1 cache misses.

Chapter 6

Conclusion

The memory wall problem has been one of the toughest problems that many computer architects try to resolve. Among many attempts to overcome the problem, prefetching has been studied extensively and considered one of the most effective methods. There are many ways to categorize prefetching schemes: complexity, target cache level, and information they utilize. According to the target cache level, prefetching schemes can be classified into two groups: a group that targets L1 cache misses and the other group that targets L2 or lower level cache misses. Between the two groups, there is significant performance gap at least with the benchmarks in SPEC2000 and Olden benchmark suites.

The proposed prefetcher targets L1 cache misses. In order to eliminate all L1 cache misses, a prefetcher should possess good timeliness, accuracy, and coverage. If prefetched blocks arrive too late, there is no gain. If prefetched blocks arrive too early, the L1 cache or a prefetch buffer are not generally big enough to hold them until they are consumed. If future addresses are not accurately predicted and the prediction doesn't cover all of the critical L1 cache misses, full potential of prefetching cannot be achieved.

As a starting prefetch mechanism, dependence-based prefetcher (DBP) [43] is chosen in this dissertation. The primary reason for this selection is that DBP is suitable for targeting L1 cache

misses. By utilizing dependence relations between producer loads and consumer loads, many of the critical misses that have a great impact on the overall performance can be accurately prefetched. Although the relations between loads do not cover for all of the critical misses, this approach is particularly effective for applications with linked-list data structures (LDS). The global history buffer (GDB) [24], considered to be one of the most effective prefetcher for desktop/engineering application [37], is not suitable for targeting L1 cache misses due to its limited applicability only to the L2 cache [51].

However, DBP has poor timeliness, because there is not enough work between a producer and a consumer. This dissertation proposes a way to overcome the poor timeliness of DBP by leveraging memory access patterns of producer loads. Different patterns are handled by different prefetchers and they are combined on top of DBP to be a more general solution. The prefetchers used are stride prefetcher, and jump pointer prefetcher. Better timeliness results in better performance, but it is a tradeoff between performance and accuracy. While DBP doesn't contain any speculation, the proposed prefetcher utilizes speculation (address prediction) to achieve better timeliness.

Although better timeliness is obtained through predictions, the coverage of the proposed prefetcher hasn't been improved. Since a consumer load needs to have a producer load that has a direct relationship with it in order for the prefetcher to work for it, the coverage is limited for applications that do not contain LDS. By adding a stride prefetcher, the problem can be alleviated, but there are consumer loads that still suffer from long access latencies. Ramos et al. [40] identified this type of load, whose relationships with its producers are linear, not direct.

This dissertation extends the dependence-identification mechanism by incorporating linear relations. In this way, more consumer loads are covered by the prefetcher, and consequently higher performance is obtained. Experimental results show that the extra hardware overhead is not prohibitively high, making the new linear-dependence-based mechanism a good candidate that can

potentially replace direct-dependence-based mechanism.

In terms of practicality, a big table introduced by the correlation-based prefetching scheme in order to improve the timeliness is an issue to be taken care of. It is too big to be implemented on chip, and the access latency will be too long if it is implemented off chip. An on-chip structure that has potential for storing the big table is the L2 cache. In order to use the L2 cache for two different purposes, a reconfigurable cache is required. In such a cache, a cache can be partitioned into multiple regions and each can be used for a different purpose. Ranganathan et al. [41] proposed a reconfigurable cache, and they found that a reconfigurable cache doesn't incur any extra access latency. In this dissertation, the reconfigurable cache is utilized to move a big prefetch table into the L2 cache.

Simulation results in Chapter 5 show that this approach is both feasible and beneficial. Even when the working set size of a benchmark is much bigger than the size of the L2 cache, implementing a prefetch table in the L2 cache (decreasing the effective size of the L2 cache) outperforms using the whole L2 cache conventionally. This is not always true, but at least such a partition size exists.

The prefetching mechanism proposed in this dissertation has room for improvement. For timely dependence-based prefetching, correlation-based prefetching such as jump pointer prefetching as well as stride prefetching are used together. For the correlation-based prefetching, the correlation found in producers' addresses are utilized. For gathering more insight, the correlations in producers' addresses should be analyzed and compared to those in miss streams. If using only producers' address streams is too limited, a different way of combining DBP and correlation-based prefetchers should be devised. This means that instead of finding a way, finding a good way has to be found.

While DBP is free of control flow variations, the new timely dependence-based prefetcher is not. This is because of the speculation introduced for better timeliness. If control flow information

can be learned and can aid the prefetcher, the accuracy of the prefetcher can be less susceptible to control flow variations.

For implementing a prefetch table in the L2 cache, there are assumptions that need to be verified and made concrete in order to be practical. Such assumptions contain how to detect reconfiguration points and how to determine a right partition size. While L2 cache size increases gradually, the working set sizes of applications may increase much faster. In this case, scalability of this approach may be an issue. In order to cope with the issue, using main memory as well as the L2 cache may be necessary. As a result, the access latency to the prefetch table in main memory becomes quite long. In this case, it is necessary to leverage spatial locality of accesses to the prefetch table. First the availability of such a locality should be checked. If the locality exists, an indexing scheme that can utilize the locality should be thoughtfully designed.

REFERENCES

[1] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Proceedings of International Symposium on Microarchitecture*, pages 248-259, 1999.

[2] Hassan Al-Sukhni, Ian Bratt, and Daniel A. Commors. Compiler-directed content-aware prefetching for dynamic data structure. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* , pages 91, 2003

[3] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of International Symposium on Computer Architecture*, pages 52-61, 2001.

[4] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, U. Weiser. Correlated load-address predictor. In *Proceedings of International Symposium on Computer Architecture*, Pages 54-63, 1999.

[5] Ioana Burcea, Stephen Somogyi, Andeas Moshovos, and Babak Falsafi. Predictor virtualization. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157-167. 2008.

[6] D. Burger, and T. Austin. SimpleScalar Toolset, Version 3.0 Tech. rep., University of Wisconsin, Madison, 1999.

[7] Mark J. Charney and Anthony P. Reeves. Generalized correlation-based hardware prefetching. *Technical Report EECEG-95-1,* School of Electrical Engineering, Cornell University, February 1995.

[8] T. F. Chen and J. L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA. ACM Press, New York.

[9] Trishul M. Chilimbi, Bob Davidson, James R. Larus. Cache-conscious Structure Definition. In *ACM SIGPLAN*, Pages 13-24, 1999.

[10] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[11] Yuan Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *Proceedings of International symposium on Microarchitecture,* pages 301-313, 2007.

[12] J. Collins, S. Sair, B. Calder, and D.M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of International Symposium on Microarchitecture*, pages 62-73, 2002.

[13] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of International Symposium on Computer Architecture*, pages 133-143, 1997.

[14] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *Preceedings of International Conference on Supercomputing*, pages 196-203, 1997.

[15] G. Hariprakash, R. Achutharaman, and A. R. Ornondi. DStride: data-cache miss-address-based stride prefetching scheme for multimedia processors. In *Proceedings of Australasian Computer Systems Architecture Conference*, pages 62-70, 2001.

[16] Z. Hu, M. Martonosi, S. Kaxiras. TCP: tag correlating prefetchers. In *Proceedings of*

International Symposium on High-Performance Computer Architecture, Pages 317-326, 2003.

[17] Christopher J. Hughes, Sarita Adve. Memory-side prefetching for linked data structures. In *Technical report:UIUCDCS-R*. pages 2001-2221 , 2001.

[18] D. Joseph, D. Grunwald. Prefetching using Markov predictors. In *IEEE Transactions on Computers*, pages 121-133, 1999.

[19] Robert Jenkin. Hash functions for hash table lookup. In http://burtleburtle.net/bob/hash/evahash.html.

[20] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of the small fully associative cache and prefetch buffers. In *Proceedings of International Symposium on Computer Architecture*, Pages 364, May 1990.

[21] M. Karlsson, F. Dahlgren, and P. Stemstrom. P prefetching technique for irregular accesses to linked data structure. In *Proceedings of International Symposium on High-Performance Computer Architecture*, Pages 206-217, 2000.

[22] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for TLB prefetching. In *Proceedings of International Symposium on Computer Architecture*, May 2002.

[23] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. Multi-chain prefetching: effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 268-279, 2001.

[24] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of International Symposium on Computer Architecture,* pages 144-154,

2001.

[25] Chungsoo Lim, Greg T. Byrd. Exploiting producer patterns and L2 cache for timely dependence-based prefetching. In *Proceedings of International Conference on Computer Design*, October 2008.

[26] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. SPAID: software prefetching in pointer- and call-intensive environments. In *Proceedings of International Symposium on Microarchitecture*, pages 231-236, 1995.

[27] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[28] Yue Liu, Mona Dimitri, and David R. Kaeli. Branch-directed and pointer-based data cache prefetching. In *Journal of Systems Architecture: the EUROMICRO Journal*, pages 1047-1073, June 1999.

[29] Chi-Keung Luk, Todd C. Mowry. Compiler-based prefetching for recursive data structure. In *ACM SIGOPS Operating Systems Review*, Pages 222-233, 1996.

[30] Aleksandar Milenkovic, and Veljko Milutinovic. Lazy prefetching. In *Proceedings of Hawaii International Conference on System Sciences*, page 780, 1998.

[31] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *the 5th International Conference of Architectural Support for Programming Languages and Operating Systems*, Pages 62, October 1992.

[32] O. Mutlu, Hyesoon Kim, and Y. N. Patt. Address-Value Delta (AVD) Prediction: A Hardware

Technique for Efficiently Parallelizing Dependent Cache Misses. In *IEEE Transactions on Computers*, Pages 1491-1508, 2006.

[33] K. J. Nesbit, J. E. Smith. Data Cache Prefetching Using Global History Buffer. In *Proceedings of International Symposium on High-Performance Computer Architecture*, Pages 14-18, 2004.

[34] Kyle J. Nesbit, Ashutosh S. Dhodapkar, and James E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of International Conference on ParallelArchitecture and Compilation techniques*, Pages 135, 2004.

[35] S. Palacharla, R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of International Symposium on Computer Architecture*, pages 24-33, 1994.

[36] Salil Pant. Slipstream-mode prefetching in CMP. Master's thesis, North Carolina State University, 2004.

[37] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of International Symposium on Microarchitecture*, pages 43-54, 2004.

[38] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. Compiler orchestrated prefetching via speculation and prediction. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* , pages 189-198, 2004.

[39] Kaushik Rajan, Govindarajan Ramaswamy. Emulating optimal replacement with a shepherd cache. In *Proceedings of International symposium on Microarchitecture,* pages 445-454, 2007.

[40] Luis Ramos, P. Ibanez, V. Vinals, and J. M. Llaberia. Modeling load address behavior through

recurrences. In *Proceedings of International Symposium on Performance Analysis of Systems and Software*, pages 101-108, 2000.

[41] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of International Symposium on Computer Architecture*, pages 214-224, 2000.

[42] G.. Reinman and N. P. Jouppi. CACTI 2.0 Beta. In www.research.digital.com/wrl/people/jouppi/CACTI.html, 1999.

[43] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[44] A. Roth and G.S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of International Symposium on Computer Architecture*, pages 111-121, 1999.

[45] S. Sair, T. Sherwood, and B. Calder. Quantifying load stream behavior. In *Proceedings of International Symposium on High-Performance Computer Architecture*, pages 197-208, 2002.

[46] S. Sharma, J. Beu, and T. M. Conte. Spectral prefetcher: An effective mechanism for L2 cache prefetching. In *ACM Transactions on Architecture and Code Optimization*, pages 423-450, 2005.

[47] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of International Symposium on Microarchitecture*, pages 42-53, 2000.

[48] T. Sherwood, E. Perelman, G. Harmerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[49] James E. Smith. Decoupled access/execute computer architectures. in *ACM Transaction on Computer Systems*, pages 289-308, 1984.

[50] G. S. Sohi, A. Roth. Speculative multithreaded processors. In *Computers*, Pages 66-73, April 2001.

[51] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 252-263, 2006.

[52] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, David A. Wood. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. In TR #1420, Computer Sciences Department, University of Wisconsin, Madison, 2000.

[53] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of International Sysmposium on High Performance Computer Architecture*, pages 63-74, 2007.

[54] Dimitrios Stiliadis, Anujan Varma. Selective Victim Caching. In *IEEE Transactions on Comupters*, Pages 603-610, 1997.

[55] K. Wang, and M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proceedings of International symposium on Microarchitecture*, Pages 281-290, December 1997.

[56] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *Proceedings of International Symposium on Computer Architecture*, pages 388-398, 2003.

[57] Youfeng Wu. Efficient discovery of regular stride patterns in irregular program and its use in

compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* , pages 210-221, 2002

[58] G. Yao, S. Chheda, I. Koren, M. C. Krishna, and C. A. Moritz. Energy characterization of hardware-based data prefetching. In *Proceedings of International Conference on Computer Design*, pages 518-523, 2004.

[59] C. Yang and A. R. Lebeck. Push vs. Pull: Data movement for linked data structure. In *Proceedings of International Conference on Supercomputing*, pages 176-186, 2000.

[60] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *Proceedings of International Symposium on High Performance Computer Architecture*, pages 86-95, 2007.