Exploiting Producer Patterns and L2 Cache for Timely Dependence-Based Prefetching

Chungsoo Lim, Gregory T. Byrd
North Carolina State University
{clim,gbyrd}@ncsu.edu

***Abstract*— This paper proposes an architecture that efficiently prefetches for loads whose effective addresses are directly dependent on previously-loaded values. This dependence-based prefetching scheme covers most frequently missed loads in programs that contain linked data structures (LDS).**

For timely prefetches, memory access patterns of producing loads are dynamically learned. These patterns (such as strides) are used to prefetch well ahead of the consumer load. 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.

We also examine how to capture pointers in 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. We show that storing the prefetch table in a partition of the L2 cache outperforms using the L2 cache conventionally.

I. Introduction

During the past decades, the gap between processor speed and memory speed has 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 to increase density rather than reduce access latency.

Current processors need more instruction level parallelism to overcome the gap, but finding enough parallelism is not easy in many applications. Consequently, processors are likely to stay idle for many cycles waiting for memory to provide requested data. To alleviate this problem, *prefetching* can be used to retrieve data from memory before it is needed, allowing memory requests to be overlapped with computation.

This paper extends a dependence-based prefetching mechanism [13], which targets loads whose addresses are computed from previously-loaded values. To improve timeliness over previous dependence-based schemes, correlations in an individual load's address stream, or correlations between the load's address stream and its corresponding loaded value stream, are exploited to issue prefetches far in advance. To design an efficient prefetcher, most representative patterns (correlations) should be chosen to work within the general framework i.e. dependence-based prefetching mechanism. For this, several patterns are selected and their effectiveness is studied. Especially for linked data structures (LDS), an efficient way of storing pointers dynamically with hardware is presented. The details of the prefetching scheme are discussed in Section II.

The proposed prefetcher is located close to the processor core and targets L1 cache misses. This is because perfect L1 cache has much more potential than perfect L2 cache. One reason is that perfect L1 cache naturally hides data translation look-aside buffer (DTLB) misses. In addition, current trends in microarchitecture favor perfect L1 cache over perfect L2 cache. Since the size of the L2 cache in commercial microprocessor is getting bigger, the access latency (for L2 hits) is getting longer. Also, the ability for the processor to tolerate L2 misses is degrading due to simpler core design.

This paper also evaluates using the L2 cache for storing large prefetch tables (Section III). This is motivated by the design philosophy of general purpose processor. A general purpose processor design has to perform well with any applications, without favoring only specific applications. If a prefetch mechanism requires a huge history table and it boosts performance for only a small number of applications, having such a huge table is a waste of precious resources. Instead of having dedicated history table, it can be moved into the L2 cache. The simulation results in Section IV show that storing prefetch table in a reasonably-sized L2 cache yields better performance than using the L2 cache conventionally.

II. TIMELY DEPENDENCE-BASED PREFETCHING

A. *Motivations*

Data dependence involving long memory latency is one of the biggest performance limiting issues in processor design. The problem increases as program working sets increase. Many prefetch schemes have been proposed to attack this problem. In order for data prefetching to be effective, it must possess the following three properties. First, it should eliminate cache misses for loads in the critical path -- in other words, it should have good *coverage* of the critical loads. Second, it must *accurately predict* future memory accesses. Last, but not least, prefetched data should be *timely* -- i.e., available to the processor before they are actually used.

Some prefetchers are able to only predict future addresses if they were seen in the past [3] [4] [7] [8] [14] [15] [16] [18], but some are capable of predicting future addresses even if they were not seen before [1] [5] [6] [10] [13]. One such mechanism is Dependence-based prefetching [13], which identifies producer-consumer relations among loads and uses them to prefetch data. When a producer's value is loaded, it is then used to issue prefetches for consumers. This doesn't require knowledge of past access patterns of consumer loads. Once the dependence relation is learned, prefetches can be made. Since every load has a preceding instruction that produces the effective address, prefetching mechanism based on this dependence between address-producing and address-consuming instructions has great coverage. Dependence-based prefetching also has good prefetch accuracy because no prediction is involved in prefetching. However, this scheme works well only when there is enough work between producer and consumer.

While Roth et al. [13] targeted only linked data structures (LDS), Murali et al. [1] proposed a more general dependence-based prefetching scheme. It generates dependence graph for loads and stores responsible for most data cache misses. All the instructions in the graph are marked and precomputed using dedicated separate hardware. Since the number of instructions in the graph is usually fewer than the corresponding original program fragments, the prefetch engine can get ahead of the main pipeline. A shortcoming of this approach is that it may not work well if there are cache misses along the dependence graph. This is because the prefetch engine has to go through the cache miss, but the main pipeline does not, so it is difficult for the prefetch engine to stay ahead.

The major issue of such approaches is prefetch timeliness. There is no guarantee that prefetches are issued early enough for data to be available before it is needed. Timeliness in these schemes depends on the available work between producer and consumer. To resolve this problem, Roth et al. [14] utilize *jump pointers* on top of dependence-based prefetching. Jump pointers point to non-adjacent nodes in an LDS, so that the prefetch engine can run ahead of the processor, resulting in timely prefetches.

There are three main differences between the jump pointer approach and our proposed scheme. First, Roth et al. used just jump pointers to resolve the prefetch timing issue, but our scheme is capable of utilizing more types of patterns, resulting in a general dependence-based framework. The type of producers' patterns and how they are utilized are described in Section II.B. Second, while the jump pointer scheme focused on using software for jump pointer creation, we propose a dynamic hardware mechanism to identify jump pointers and an efficient way of populating the jump pointer table. Third, to effectively use the resources assigned to identifying producer-consumer pairs and storing producers' patterns, only small numbers of static loads that miss frequently in L1 cache are dynamically identified.

B. Utilizing Producers' Patterns to Improve Dependence-based Prefetching

In order to issue a prefetch request for a consumer well before the consumer accesses the memory hierarchy, either its producer's future effective address or its producer's future value should be predicted first. Once the producer's future address or value is predicted, the consumer's future address is computed from the dependence relation between the producer and the consumer.

Any pattern that enables the prefetcher to predict future information of producers is exploitable. Patterns that naturally satisfy this requirement are ones that exist in the producer's effective address stream. Three well-known such patterns are *stride pattern*, *recurrent pattern* [13], and *pointer pattern* [4]. If correlation doesn't exist in the producer stream, then correlation in the entire address stream or the miss address stream can be used. An advantage of combining dependence-based prefetching scheme with correlation-based prefetching [7] is that the table size for the correlation-based scheme can be reduced. First, a dependence relation can replace quite a few correlation patterns of a consumer load, because addresses of the consumer load can be predicted from the dependence relation and the producer's output values. Second, chain prefetching based on dependence relations also replaces many correlation patterns, because one correlation pattern along with dependence relations is able to prefetch every node in a whole dependence tree.

Stride pattern in producer's address stream makes predicting a future address straightforward. Simply multiplying the stride with small non-zero integer will result in a future address. This pattern is very efficient in terms of space, and it is able to predict addresses that have not been seen before. One drawback of this pattern is that it requires two prefetches for a consumer: one for getting the loaded value of the producer and the other for accessing the consumer's effective address.

Recurrent pattern involves not only address streams but also loaded value streams. It is made up of two correlations. The first correlation exists between an effective address and the value loaded from it. The second correlation exists between a loaded value of an instance and the effective address of the next instance. Since this pattern can be seen in a load that is responsible for generating the address of the next node in an LDS, it is likely that the corresponding long address sequence repeats quite a few times. Because of this strong correlation, it is possible to have jump pointers with fairly long distance. Another advantage of this pattern is that it is easy to identify. Almost every load that has the second correlation also possesses the first correlation, and the second correlation is easy to identify. This means that once the second correlation is identified, it is highly probable that the corresponding load has strong correlation in its address stream.

Pointer pattern is identical to the recurrent pattern, but without the second correlation. With this pattern, only the address of consumer in the same iteration can be predicted. If a producer with this pattern hits in L1 cache frequently, there is no point for prefetching, because looking up the producer's loaded value in the prefetch table takes more cycles than getting the value from the L1 cache. If the time taken for accessing prefetch table is about the same as the time taken for handling a cache miss, there is also no gain. A subset of this pattern has very useful property: address-value delta [10]. This pattern has a constant difference between addresses and values, so that loaded values can be computed from corresponding addresses.

Correlation in address stream from processor core the correlation most prefetchers are based on. As it is, this correlation cannot be used for predicting a producer's future address. To be more exact, it is possible to predict future addresses, but associating them with static loads is impossible. But if load information is appended to each correlation, this type of correlation can be used on top of dependence-based prefetching mechanism.

In this paper, *stride pattern* and *recurrent pattern* are implemented. *Pointer pattern*, as it is, doesn't work well with using L2 cache for storing pointer table because the time accessing a pointer from L2 cache is about the same as loading the actual corresponding data from L2 cache in case where the data is in L2 cache. *Correlation in address stream from processor core* is left for future work.

C. Implementation of Dependence-based Prefetching Mechanism

In this section, hardware implementation details of the prefetch scheme are presented. This is an exemplary implementation of dependence-based prefetching mechanism that combines stride prefetching and jump pointer prefetching. As shown in Figure 1, the proposed prefetcher is composed of four parts: frequently-missed load table (FMLT), potential producer window (PPW), dependence information table (DIT), and jump pointer creation unit (JPCU). (The jump pointer table (JPT) is not shown, because it is stored in the L2 cache, as described in Section III.)

Every committed load instruction is sent to the four tables by the processor. FMLT is in charge of pinpointing static load instructions that miss in L1 cache frequently. With this filtering mechanism, the space requirements for DIT, JPCU, and JPT are reduced.

The FMLT is updated by committed loads from the processor; it 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.

The 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. If a producer-consumer pair is identified, the pair is inserted into the DIT.

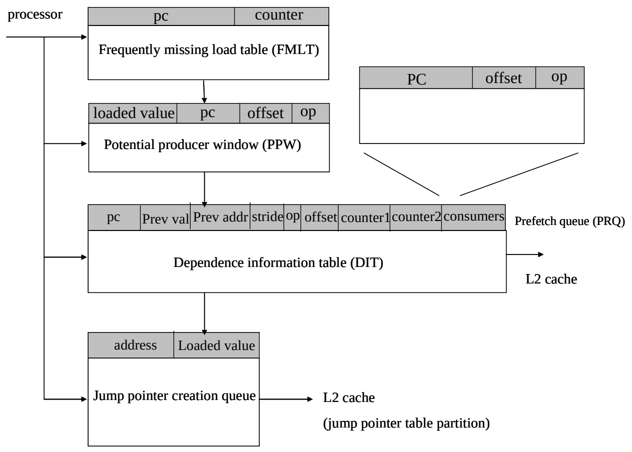

Fig. 1. Dependence-based prefetching mechanism with stride and jump pointer prefetching.

The DIT plays two roles. It detects producers' patterns, and it stores producer-consumer pairs. One producer can have multiple consumers, which is common in producer-consumer relationships. 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 DIT. A load's PC is used to index into DIT. If the indexed entry has a stable pattern, a prefetch request is issued and stored in prefetch request queue (PRQ). For a stride pattern, the future address is computed by first multiplying the stride by a small constant, and then adding the result to the current address. For a recurrent pattern, JPT is accessed with the current address to get a future address. The address returned from JPT is added to a consumer's offset to form a prefetch address. For chain prefetching, DIT is looked up again. When a prefetch request is queued in PRQ, the corresponding DIT entry's index is stored. When the prefetched data arrives, DIT 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.

Every committed load updates the DIT. If the load is a producer, its value is used to detect stride or recurrent patterns. 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. If a match is found, and the corresponding producer has been identified as a recurrent load, the JPT can be updated. The DIT sends the PC of the corresponding producer to JPCU, and JPCU sends two addresses to JPT.

In JPCU, there are 16 FIFO queues, each corresponding 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 queue assigned to the PC, gets two addresses from the queue, and sends them to JPT. The two addresses are chosen using a predefined constant jump pointer distance. For a distance of *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, and a portion of it is stored as a tag.

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.

To effectively make the most of the limited-sized JPT, JPCU does more than the basic operation described above. There are two optimizations. First, jump pointers are formed only for consumers that miss in the L1 cache. Second, chain prefetching mechanism is utilized. Some instances of recurrent loads are prefetched via chain mechanism, eliminating the need for storing jump pointers. Figure 2 shows the effectiveness of these two mechanisms, normalized to the IPC of the no-prefetch case. For this result, only recurrent pattern is used for prefetching.

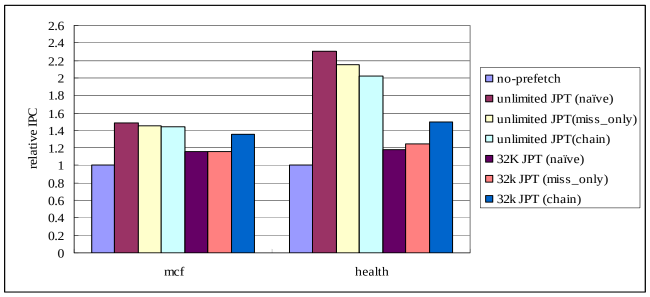

Fig. 2. The effectiveness of jump pointer creation mechanisms.

Since *mcf* and *health* are the benchmarks that require big JPT, only these two benchmarks are studied here. For unlimited JPT, naïve creation, with which jump pointers are created for all dynamic instances of producers as in the original jump-pointer prefetching [14], 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 a 32K-entry JPT, chain-prefetch-aware creation performs the best. For *mcf*, the number of jump pointers created for missed-instance-only creation and chain-prefetch-aware creation are similar, but chain-prefetch-aware creation results in a larger number of effective prefetches through chain prefetching.

III. USING L2 CACHE FOR STORING PREFETCH TABLE

A. *Motivation*

For a hardware prefetching mechanism to be implemented in commercial microprocessors, the mechanism needs to be simple yet effective. Unfortunately, there is no such prefetcher that works well with 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 [9] [14] 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 [3]. 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.

In this paper, 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. 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 table, 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.

B. *Implementing Prefetch Table in L2 Cache*

Ranganathan et al. [11] 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 3 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 [12]. They found that the overall microprocessor cycle time and cache access latency are not affected.

Table 1 summarizes the design choices that we make for implementing reconfigurable cache. As a partitioning mechanism, associativity-based mechanism is selected because it just requires a little change in existing cache structure.

Virtual addresses are used to access the prefetch table partition. To uniformly distribute accesses to ways of prefetch table, hash functions are in charge of generating index to the table from virtual addresses. Because of the limited associativity and size of prefetch table implemented in L2 cache, the role of the hash functions is critical to prefetching performance. This is why two hash functions are used instead of one. If an index generated by a hash function is to an entry that's been already occupied, the index from the other hash function is used instead. In this way, a relatively small prefetch table in L2 cache can be efficiently utilized.

Data consistency ensures that data belonging to prefetch table resides only in its partition. If a reconfiguration occurs, data in the portion of the cache that is converted from prefetch table to conventional cache should be discarded. If a way is converted from conventional cache to prefetch table, its dirty data must be written back to main memory. Such a cache-scrubbing mechanism requires examining all the locations of the cache to check for their validity and performing appropriate actions.

Repartitioning is assumed to be kept minimal and detection of reconfiguration is controlled by software.

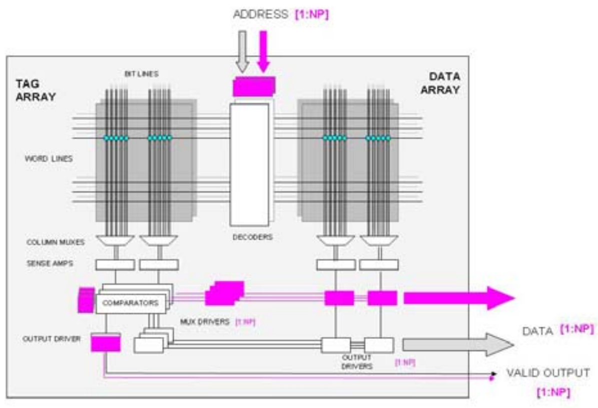

Fig. 3. Implementation of a reconfigurable cache. [11]

Table 1. Design Choices for Reconfigurable Cache.

Design issue	**Used in this paper**
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

IV. EVALUATION

In this section, we experimentally evaluate the effectiveness of the proposed dependence-based prefetch mechanism, and the effectiveness of using L2 cache for storing the prefetch table.

Table 2. 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, 4 port data cache; 32K, 2way, 32B, 16 MSHR, 1 cycle L1 instruction cache; 1M/2M, 8way, 64B, 16 MSHR,15/18 cycle, 1 port 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(1.1KB) FMLT; 128 entry(2.4KB) PPW; Fully associative 128 entry(9.1KB) DIT(4 consumers per producer); 32 entry(0.32KB) PRQ; 128 entry(4KB), 32B Prefetch buffer; 16 8-entry FIFO(2KB) in JPCU;

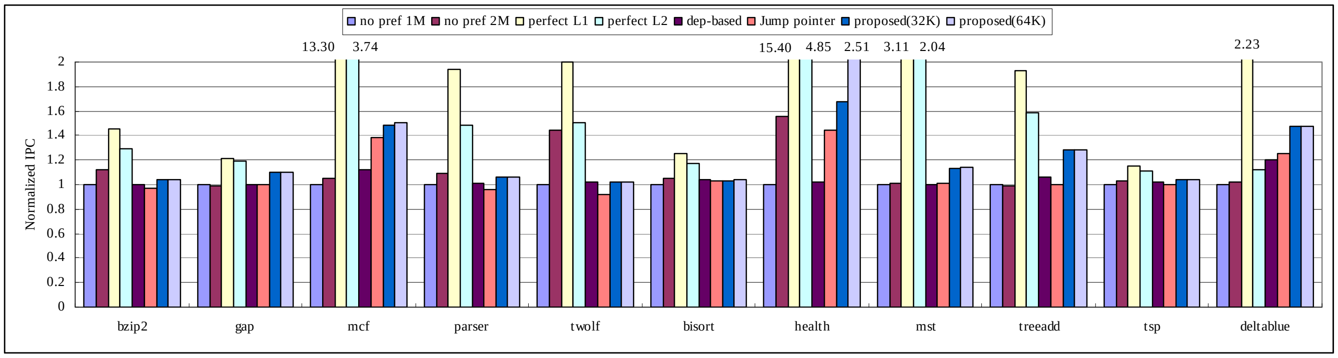

Fig. 4. Performance comparison

A. *Experimental Setup*

Our experiments were performed using part of SPEC2K integer benchmark suite, Olden benchmark suite, and *deltablue* benchmark, which is a pointer-intensive benchmark. Olden and *deltablue* benchmarks were simulated to completion. For SPEC2K benchmarks, a representative 200 million instructions were simulated, which were identified by SimPoint [17]. SPEC2K benchmarks use reference inputs. For our experiments, SimpleScalar simulator is used [2]. The memory interface has been rewritten to model cache hierarchy, bus occupancy, and limited bus bandwidth, similar to Sharma et al. [15]. The simulated processor configuration is summarized in Table 2.

To accurately model prefetch table in L2 cache, the number of ports on L2 cache is set to 1. Demand fetch and prefetch requests share L1-L2 bus, and a virtual address provided by prefetch table L2 goes through DTLB before the corresponding prefetch request is queued in PRQ.

B. *Speedups*

We now compare the performance of our proposed prefetching scheme with larger cache, perfect cache, and other prefetching schemes that are closely related to it. Figure 4 shows the IPCs of seven different configurations. The first two bars represent IPCs of different L2 cache sizes (1MB and 2MB) without prefetching. The next two bars show IPCs of perfect L1 Cache and perfect L2 Cache, respectively. The last four are IPCs of three prefetching schemes: original dependence-based prefetching [13], jump-pointer prefetching (with 64K-entry JPT) [14], and the proposed prefetching (with 32K- and 64K-entryJPT). 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.)

Except for *health* and *twolf*, going from 1M L2 cache to 2M L2 cache doesn't improve performance more than 10%. This means that the advantage of reduced L2 cache misses does not significantly overcome the disadvantage of longer hit latency.

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, indicates that using inefficiently-utilized L2 cache has great potential. 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 little benefit, there is still much room for performance improvement.

The proposed prefetching mechanism is implemented in two different configurations. The first configuration has 32k entries in JPT, consuming approximately 0.5MB, and the second configuration has 64k entries, requiring about 1MB of space. Both configurations are implemented separately from 1MB L2 cache. If we compare the corresponding two bars 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, bisort, parser, and twolf*, even the combination of 0.5MB prefetcher with 1MB L2 cache outperforms a 2MB L2 cache.

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. 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 also better than original jump-pointer prefetching, for two 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.

In *health*, recurrent pattern is dominant, whereas in *parser* and *treeadd*, stride pattern is dominant. Other benchmarks such as *gap*, *mst*, *deltablue*, and *mcf* are getting benefit from both patterns.

C. Results for implementing prefetch tables in L2 cache

This sub-section shows the results for implementing prefetch tables in L2 cache. Two L2 cache configurations are used: 1MB 8way and 2MB 8way. The hit latency of 1MB cache is set to 15 cycles, and that of 2MB cache is set to 18 cycles. In addition to traditional cached data, the L2 cache holds the jump pointer table; each 64-byte cache block holds four jump pointer entries. We vary the size of the jump pointer table in L2 cache to see how it influences the overall performance.

In Figure 5, there are two graphs. The x-axis of each graph represents the number of ways reconfigured for storing the jump pointer table, and the y-axis indicates IPC. For 1MB cache, one way is 128KB, and for 2MB cache, it is 256KB. The upper graph is for 1MB cache configuration, and the other one is for 2MB L2 cache configuration. There are six plots in each graph (three per benchmark). The plot labeled "no pref" represents the case where the partition set aside for the jump pointer table 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 jump pointer table is actually used and prefetching mechanism is put into action. The plot labeled "conventional cache" indicates the case where the whole 1MB or 2MB of cache is used conventionally without prefetching.

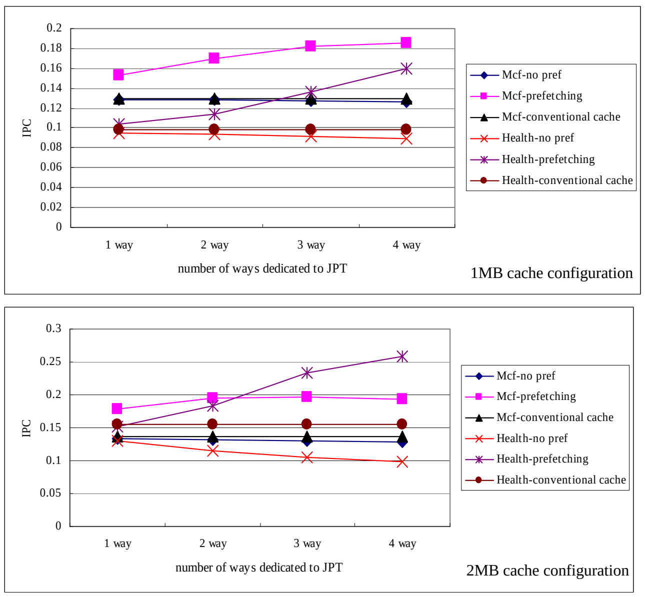

Fig. 5. Performance with varying the partition size for the jump pointer table in L2 caches: 1MB L2 Cache (upper) and 2MB L2 (bottom)

For *mcf*, IPC drops marginally for both configurations, as the effective L2 cache size decreases without prefetching. Therefore, prefetching using the jump pointer table in L2 is able to improve performance regardless of the number of ways assigned to the jump pointer table.

Though not shown here, IPC drops drastically with 8MB L2 cache as more ways are assigned to the jump pointer table. In this case, using only 1 way is advantageous. This tells us that there may be an L2 cache size for a benchmark, where borrowing some portion of the space to prefetching mechanism doesn't help. It should also be noted that the working set of *health* doesn't fit in 2MB L2 cache.

When reconfigured cache size of 2MB cache for conventional caching is 1MB (4 ways are used for the jump pointer table), the IPC without prefetching is just 0.1277. This is even lower than the IPC (0.1299) when the whole 1MB cache is used conventionally. Since the associativity of the former is just four compared to eight of the latter, IPC is a little bit reduced.

Similarly, for *health*, prefetching backed by the jump pointer table in L2 cache always has higher IPC than using the whole 2MB of cache conventionally. However, there are some differences. First, *health* is more sensitive to the jump pointer table size. When one way is used for the table, prefetching doesn't improve performance much. On the contrary, as the number of ways for the table increases, IPC increases faster than that of *mcf*. This is because *health* needs to store more jump pointers to improve performance.

Second, IPC drops steadily for 2MB L2 cache, as the effective L2 cache size decreases without prefetching. Fortunately, the IPC increase due to prefetching is much greater than this loss due to reduced effective L2 cache size. This tells us that even benchmarks that use L2 cache effectively may benefit from prefetching mechanisms that borrow some portion of L2 cache.

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 jump pointer table 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, a benchmark that utilizes L2 cache effectively can benefit a lot from this approach, just like *health* does in this experiment.

V. Conclusion

This paper has implemented and evaluated 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 paper, we have used stride and recurrence patterns, commonly found in pointer-intensive programs.

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.

Because the space requirement for the prefetching tables is still quite large, we also 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.

As future work, more complex producer-consumer relations will be studied to cover more consumer load instructions. If other relations can be found that are a little more complex than direct relation and cover more consumer load instructions, more benefit would be gained with small increase in complexity. Another future direction will be considering control path information to increase prefetch accuracy. It is possible that a producer has multiple consumers, some of which are from a different control path. Therefore, prefetching for all of them without considering control flow information may result in issuing too many unnecessary prefetches. To remedy this drawback, taking control path information into account may be necessary.

REFERENCES

[1] 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.

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

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

[4] 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.

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

[6] 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.

[7] D. Joseph, D. Grunwald. Prefetching using Markov predictors. *IEEE Transactions on Computers*, 48(2):121-133, February 1999.

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

[9] Chi-Keung Luk, Todd C. Mowry. Compiler-based prefetching for recursive data structure. *ACM SIGOPS Operating Systems Review*, 30(5):222-233, December 1996.

[10] O. Mutlu, Hyesoon Kim, and Y. N. Patt. Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses. *IEEE Transactions on Computers*, 55(12):1491-1508, December 2006.

[11] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of International Symposium on Computer Architecture*, Pages 214-224, 2000.

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

[13] 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*, pages 115-126, 1998.

[14] 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.

[15] S. Sharma, J. Beu, and T. M. Conte. Spectral prefetcher: An effective mechanism for L2 cache prefetching. *ACM Transactions on Architecture and Code Optimization*, 2(4):423-450, December 2005.

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

[17] 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*, pages 45-57, 2002.

[18] 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.