Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications

Lawrence Spracklen, Yuan Chou & Santosh G. Abraham
Scalable Systems Group
Sun Microsystems Inc., Sunnyvale, CA
{lawrence.spracklen,yuan.chou,santosh.abraham}@sun.com

Abstract

In this paper, we study the instruction cache miss behavior of four modern commercial applications (a database workload, TPC-W, SPECjAppServer2002 and SPECweb99). These applications exhibit high instruction cache miss rates for both the L1 and L2 caches, and a sizable performance improvement can be achieved by eliminating these misses.

We show that it is important, not only to address sequential misses, but also misses due to branches and function calls. As a result, we propose an efficient discontinuity prefetching scheme that can be effectively combined with traditional sequential prefetching to address all forms of instruction cache misses.

Additionally, with the emergence of chip multiprocessors (CMPs), instruction prefetching schemes must take into account their effect on the shared L2 cache. Specifically, aggressive instruction cache prefetching can result in an increase in the number of L2 cache data misses. As a solution, we propose a scheme that does not install prefetches into the L2 cache unless they are proven to be useful.

Overall, we demonstrate that the combination of our proposed schemes is successful in reducing the instruction miss rate to only 10%-16% of the original miss rate and results in a 1.08X-1.37X performance improvement for the applications studied.

1 Introduction

With processor speeds continuing to outpace the memory subsystem, cache missing memory operations increasingly dictate application performance. While one typically thinks in terms of data misses and data prefetching, instruction cache misses can also represent a significant performance loss for modern processors. Indeed, instruction misses are usually more expensive than data misses as instruction misses stall the processor pipeline, while a significant proportion of data misses can often be overlapped with other instructions.

For the frequently analyzed SPEC CPU2000 benchmarks, instruction cache misses have minimal impact on performance, since the instruction working set of the majority of these benchmarks is sufficiently small to fit comfortably in the level one instruction caches supported by today's high-end processors. However, for many modern commercial workloads, exemplified by databases, application servers and web servers, the instruction working set is sufficiently large to cause significant stalls due to both Level 1 (L1) and Level 2 (L2) cache misses.

Instruction prefetching schemes can be utilized to reduce the performance impact of these misses. Prefetching schemes speculatively initiate a memory access for a cache line, bringing the line into the caches before the processor requests the cache line, minimizing, or even eliminating, the stall experienced by the processor. Instruction prefetching is not a new topic and a variety of different hardware (HW) and software instruction prefetch schemes have been proposed. However, most modern processors only support very rudimentary HW instruction prefetchers that target only sequential misses.

Furthermore, as the relative distance to memory increases, prefetches must be issued increasingly far in advance. Today, this lookahead distance is in the order of hundreds of processor clock cycles. This necessitates that prefetchers become increasingly speculative and introduces new challenges: 1) how to identify the direction the application is headed sufficiently early to allow timely prefetches to be issued and ii) although the prefetcher is speculating on program behavior hundreds of cycles in the future, it must be able to do so accurately, in order to minimize the prefetch bandwidth requirements and cache pollution. These challenges are even more relevant for Chip Multi-Processors (CMP), where there is the potential for inter-core pollution via the shared L2 cache, and the per-core bandwidth and cache resources are much more limited.

This paper makes four main contributions. Firstly, we

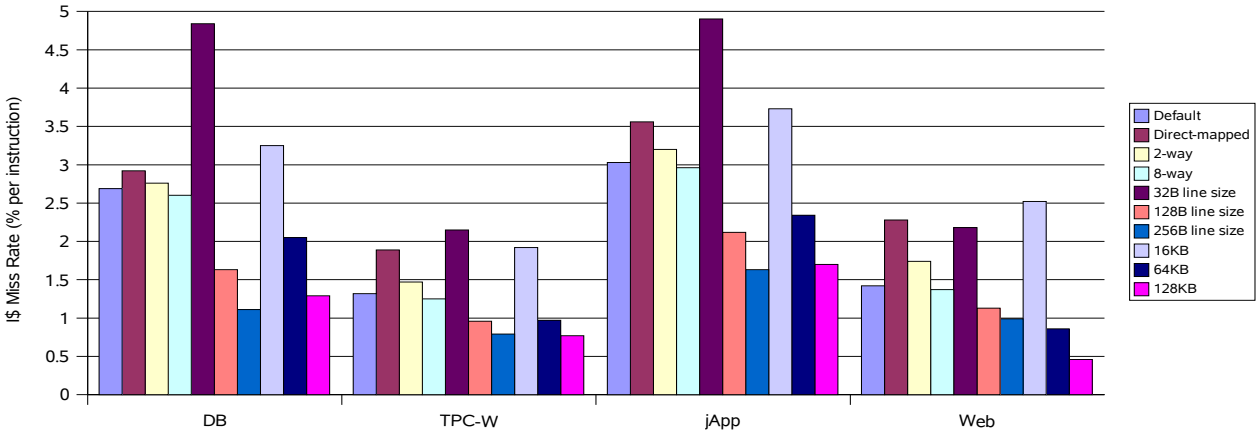

Figure 1. Instruction cache miss rates (% per retired instruction) as cache associativity, line size and capacity are varied (default is 32KB, 4-way, 64B line size)

analyze the instruction miss behavior of four modern commercial workloads, a database workload, TPC-W, SPECjAppServer2002 and SPECweb. We illustrate that it is critical for prefetchers to not only target the sequential misses, but also the misses due to branches and function calls. We further illustrate that large performance improvements can be achieved by eliminating these misses.

Secondly, we analyze the performance of a variety of previously proposed HW instruction prefetchers for these commercial workloads and propose an effective discontinuity prefetcher that is designed to allow the large lookahead distances required by today's processors. We illustrate that the combination of this discontinuity prefetcher and a traditional sequential prefetcher successfully eliminates all types of instruction misses.

Thirdly, we focus our evaluation on the emerging CMP design point, where multiple cores (with private L1 caches) share a modest L2 cache. We illustrate that, while the L2 instruction miss rate increases relative to a single core processor, HW prefetchers can eliminate almost all of these misses.

Finally, we illustrate the danger of cache pollution associated with aggressive instruction prefetching, which is especially problematic in the CMP environment. We propose a solution that eliminates this pollution and we illustrate the resultant improvement in performance.

2 Related work

Although instruction prefetching can be performed via either hardware or software schemes, we focus on HW prefetching schemes in this paper. HW instruction prefetching schemes can be divided into two basic categories; those which target sequential misses and those which target non-sequential misses.

2.1 Sequential instruction prefetchers

The simplest form of instruction prefetching is next-line prefetching [1]. In this scheme, when a cache line is fetched, a prefetch for the next sequential line is also initiated. A number of variations of this basic scheme exist, with different heuristics governing the conditions under which a prefetch is issued. Common schemes include: always issuing a prefetch for the next line (next-line always), issuing a prefetch only if the current line resulted in a miss (next-line on miss) and issuing a prefetch if the current line is a miss or is a previously prefetched line (next-line tagged) [2].

Next-N-line prefetch schemes extend this basic concept by prefetching the next N sequential lines following the one currently being fetched by the processor [3]. The benefits of prefetching the next N-lines include, increasing the timeliness of the prefetches and the ability to cover short non-sequential transfers (where the target falls within the N-line "prefetch-ahead" distance).

Alternatively, rather than increasing the "prefetch-ahead distance" and prefetching the next N sequential lines following the active line, the "prefetch lookahead distance" can be increased to N lines [4]. In this scheme, a single line is prefetched, but it is the N^{th} sequential line following the active line that is prefetched. This scheme has the ability to improve the timeliness of the prefetches without requiring a prefetch infrastructure that can handle N prefetch requests per demand fetched line. However, when control transfer instructions occur with regularity, this method can lead to significant gaps in the prefetched stream and a correspond-

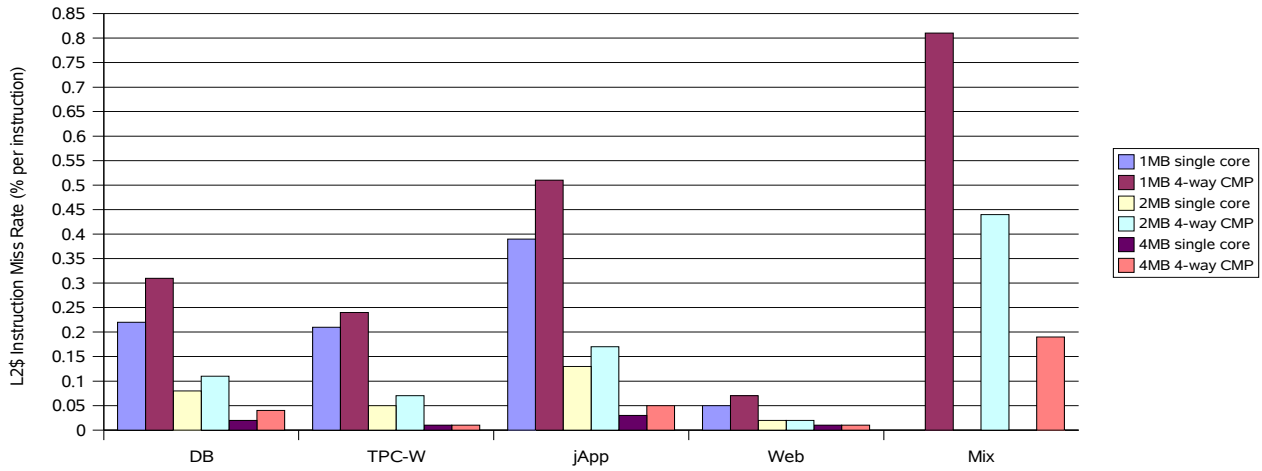

Figure 2. L2 cache instruction miss rates (% per retired instruction) for single core and 4-way CMP as cache capacity is varied (default is 2MB, 4-way, 64B line size)

ingly low coverage of misses.

2.2 Non-sequential instruction prefetchers

While next-N-line prefetching can cover small discontinuities in the fetch stream, it is not effective at eliminating the misses resulting from transitions to distant lines. This can be problematic in many applications, especially when the application is composed of small functions or there are frequent changes in control flow. There are essentially two main styles of prefetcher specifically targeted at non-sequential misses: history-based schemes and execution-based schemes.

For history-based schemes, such as the target prefetcher proposed in [1, 5], a table is used to retain information about the sequence of cache lines previously fetched by the processor. As execution continues, the table is searched using the address of each demand fetched line. If the address hits in the table, the table returns the addresses of the next cache lines that were fetched the previous times the active cache line was fetched. Prefetches can then be selectively issued for these lines. In this scheme, the behavior of the application is predicted to be repetitious and prior behavior is used to guide the prefetching for the future requirements of the application. The prediction table may contain data for all transitions, or just the subset that relate to transitions between non-sequential cache lines. A variety of other history-based schemes, each retaining past history in a different manner and triggering the issue of prefetches on different events, have been proposed [6, 7, 8].

In contrast to attempting to predict the upcoming non-sequential transitions based on an application's past behavior, execution-based prefetching techniques scout ahead of the main thread of execution and prefetch the cache lines that constitute the predicted path of execution. For instance, in [9, 10, 11], the authors propose using a branch predictor that runs ahead of the execution unit and prefetches potentially useful instructions ahead of their use.

While many non-sequential prefetchers have been discussed in literature, the vast majority are evaluated for applications with small working sets, such as SPEC CPU2000. The size and complexity of many of these prefetchers increase rapidly with the size of the working set, making them unsuited to area-constrained CMP processors running commercial workloads. For instance, in fetch directed prefetching [9], a basic block predictor predicts a sequence of basic blocks to be prefetched. However, commercial workloads have very large instruction working sets and small basic blocks. Therefore, a huge basic block predictor is required, rendering the scheme impractical. In contrast, our proposed prefetcher only needs to store large discontinuities in the instruction fetch stream, since it relies on the next-4-line sequential prefetcher to prefetch small discontinuities, such as taken-branches.

Additionally, some prior schemes only target a subset of the non-sequential misses, missing significant performance opportunities [8].

2.3 Alternative schemes

While the majority of instruction prefetching schemes fall into the two categories previously discussed, there are a couple of alternative schemes.

In [12], the authors propose "wrong-path" prefetching after observing that, for many conditional branches, both outcomes of the branch occur in a sufficiently short space

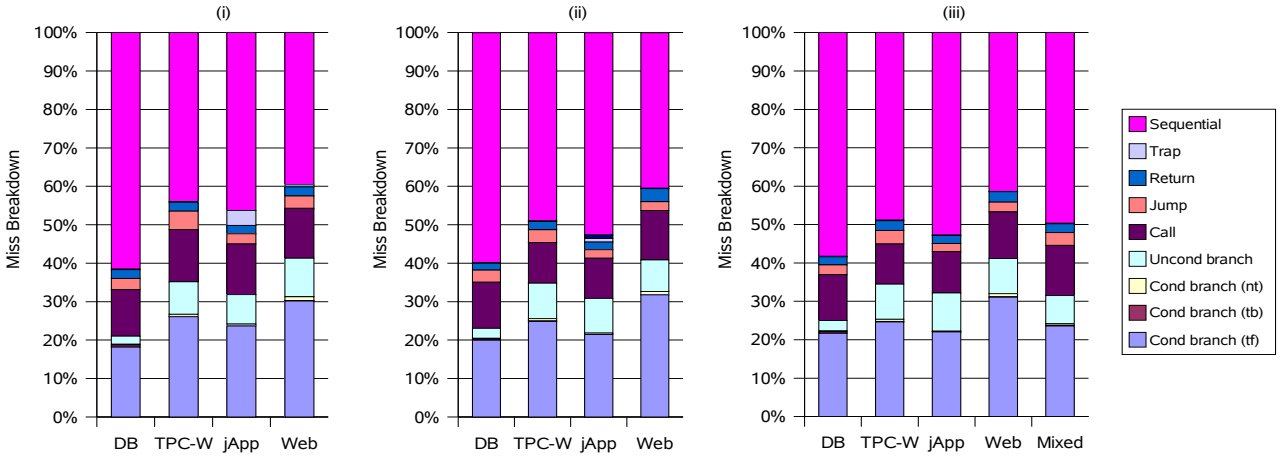

Figure 3. Breakdown of instruction misses by category; (i) Instruction cache (single core), (ii) L2 cache (single core), and (iii) L2 cache (4-way CMP)

of time that fetching for both outcomes of a branch, the correct-path and the wrong-path, effectively prefetches the wrong-path for subsequent use.

In [13], the authors state that, while the HW prefetchers are effective at eliminating sequential misses, HW prefetchers find it problematic to issue timely prefetches for non-sequential misses, and that utilizing the compiler to insert software prefetches for non-sequential misses is much more effective.

The use of helper threads to perform timely instruction prefetching is proposed by [14].

2.4 HW prefetcher refinements

There has been significant research into improving the efficiency and reducing the cost of HW instruction prefetchers.

In [15], the authors attempt to address the timeliness problems of HW prefetchers by triggering the prefetches for non-sequential misses following the issue of an instruction a known distance ahead of the miss. Additionally, the authors reduce the size of the prediction table required to accurately predict the behavior of the misses by retaining information about the number of sequential misses following a non-sequential miss in each table entry.

One detrimental aspect of utilizing large prefetch-ahead distances (i.e. large N) is the over-run that occurs when the code reaches the end of a sequential segment – resulting in the issue of potentially useless prefetches and resultant pollution. In [16], the authors propose a solution where software explicitly marks the end of a sequential sequence, allowing aggressive prefetching to be curtailed as the end of a sequential section is approached, eliminating the problem of prefetch over-run and improving prefetch accuracy. An alternative technique proposed in [13] is to retain information with each line in the L2 cache indicating whether the line was utilized when previously prefetched into the L1 instruction cache. When lines which have not previously proved useful are re-prefetched, the prefetches for these lines are just dropped.

In [15], the authors note the problems associated with accurately ascertaining whether a line to be prefetched is already in the cache. This determination is traditionally achieved by the process of inspecting the cache tags. This "cache probing" can be expensive, frequently requiring duplication of the cache tags, in order to allow the prefetcher to perform probes in parallel with demand fetches. The authors propose associating a confidence indicator with each entry in the prediction table. The confidence counter is incremented when a line is evicted from the cache and decremented when the prefetch is found to be ineffective. Prefetches are only issued if the confidence in the prediction exceeds a specified threshold. The authors claim that this filtering scheme is sufficiently accurate to remove the requirement for the duplication of the cache tags, allowing a significant reduction in the hardware cost associated with supporting an aggressive instruction prefetcher.

3 Instruction miss behavior and prefetching potential

In this section, we analyze the instruction miss behavior of four modern commercial applications - a database workload (DB), TPC-W, SPECjAppServer2002 (jApp) and SPECweb99 (web). TPC-W is a transactional web bench-

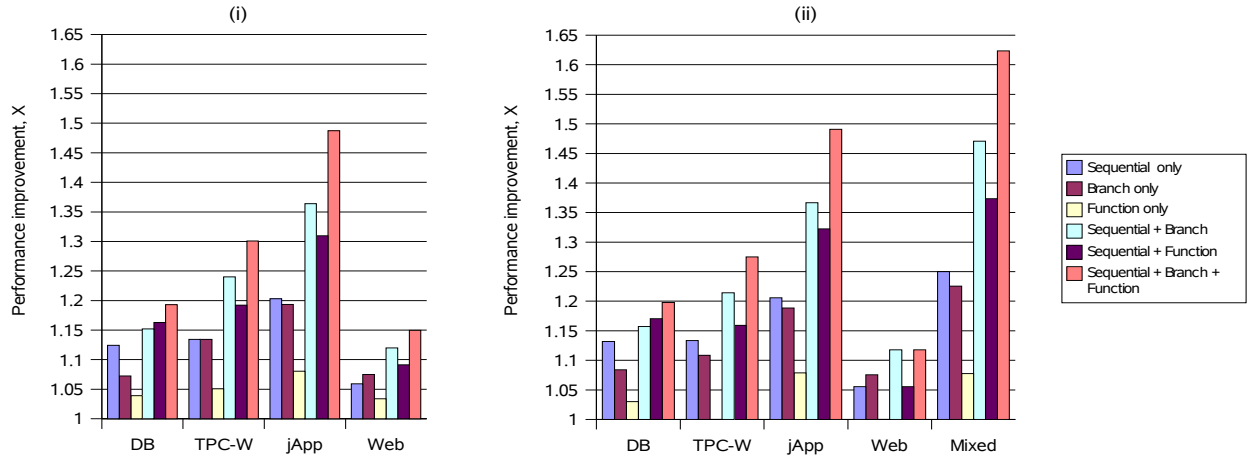

Figure 4. Performance improvement (relative to no prefetch) achievable by eliminating instruction misses; (i) single core and (ii) 4-way CMP

mark that simulates the activities of a business oriented transactional web server. SPECjAppServer 2002 is a client/server benchmark for measuring the performance of Java Enterprise Application Servers using a subset of J2EE 1.3 APIs in a complete end-to-end web application. SPECweb99 is another benchmark for measuring web server performance, but unlike TPC-W, it does not use transactions. The database workload represents an on-line transaction processing workload (due to disclosure agreements, we are unable to elaborate on the details of our database workload). For the 4-way CMP, we also evaluate a mixed/composite workload, where each core runs one of the four applications. This represents a multiprogrammed workload running on a CMP. Details about our tracing and experimental methodology are described in Section 5.

Unless otherwise stated, the default cache configuration assumed for the rest of this paper is: a 32KB 4-way 64B line size instruction cache (per core), a 32KB 4-way 64B line size data cache (per core) and a unified 2MB 4-way 64B line size L2 cache (shared by the four cores on the chip). While a larger instruction cache and a larger L2 cache will obviously reduce the number of instruction misses, we believe that it is unlikely that CMPs will have very large on-chip caches, since this involves trading off the number of cores that can be placed on the chip. Also, while the focus of this paper is on CMPs, for comparison purposes, whenever it is appropriate, we also show data for a single core processor, where the L2 cache is private to the core.

3.1 Miss rates

Figure 1 shows the instruction cache miss rate of the four applications as the associativity, line size, and capacity of the instruction cache is varied. Since each core of the CMP has its own private instruction cache, this data applies to both the single core processor and the 4-way CMP. For the default instruction cache configuration, the miss rate is between 1.32-3.16% per instruction, with SPECjAppServer2002 having the highest miss rate (as noted by others [17]). Assuming a 20-cycle L2 cache latency (and that this latency cannot be hidden), this translates into 0.25-0.65 CPI. Increasing the instruction cache line size is highly effective in reducing its miss rate (as noted by others [18]). However, it has a negative impact on both the L2 cache data miss rate, as well as the off-chip bandwidth, since a larger fraction of the longer line is usually not used. Sequential instruction prefetching achieves the same effect as larger line sizes but, since it is more selective, it has a smaller impact on the L2 cache data miss rate and the off-chip bandwidth. Therefore, sequential prefetching may be preferable to increasing the line size. As expected, increasing the instruction cache capacity improves its miss rate. However, as noted earlier, it is unlikely that CMPs will have large on-chip caches (additionally, increases in the instruction cache size rapidly result in an increased access latency). Associativity also improves miss rate, although to a lesser extent, and little benefit is obtained beyond 4-way set associativity.

We note that for all the applications except TPC-W, the instruction cache miss rate is higher than the data cache load miss rate (not shown due to space constraints). In an out-of-order processor, instruction misses are usually more expensive than data cache load misses since they stall the processor pipeline, while the latter can be usually be overlapped with other instructions.

Figure 2 shows the L2 cache instruction miss rate for the single core and the 4-way CMP as the size of the L2 cache is

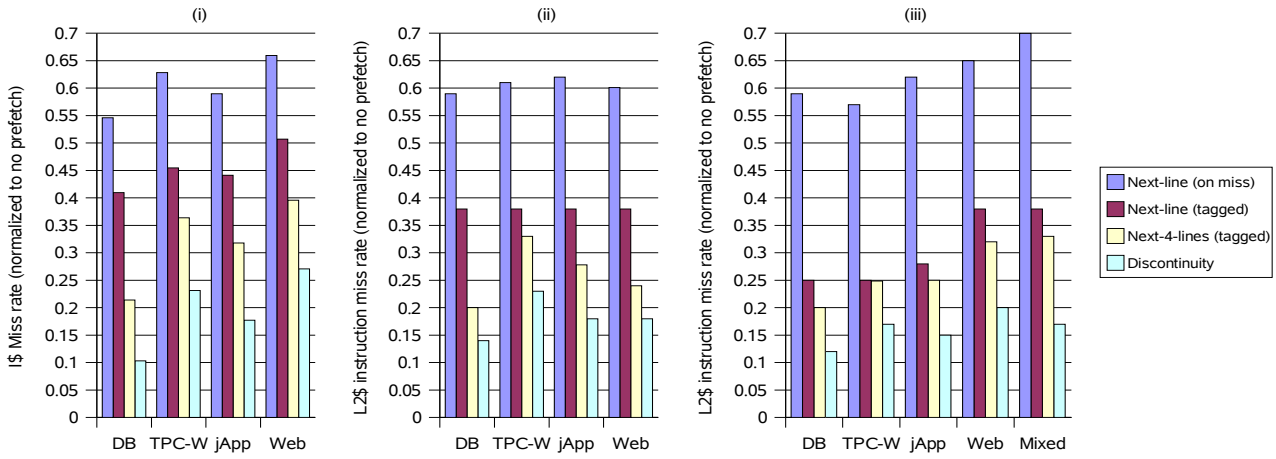

Figure 5. Instruction miss rates for different HW prefetching schemes (relative to no prefetch); (i) Instruction cache, (ii) L2 cache (single core) and (iii) L2 cache (4-way CMP)

varied. The results show that the database workload, TPC-W, SPECjAppServer and the mixed workload have high L2 cache instruction miss rates. For the default (2MB) L2 cache configuration, the miss rates on the 4-way CMP are between 0.07-0.44%. Assuming a 400-cycle memory latency (and that this latency cannot be hidden), this translates to 0.28-1.76 CPI. For a 1MB L2 cache, the miss rates on the 4-way CMP are between 0.24-0.81%, which potentially translates to 0.96-3.24 CPI. Capacity has a large impact on miss rate (and hence CPI), although the improvement from 2MB to 4MB is less than that from 1MB to 2MB. We also note that the multiprogrammed mixed workload has by far the highest L2 cache instruction miss rate and that the miss rate is higher than the sum of the miss rates of the four applications in isolation. This illustrates the importance of instruction prefetching in CMPs running multiprogrammed workloads.

Comparing the L2 cache instruction miss rates of the single core processor with that of the 4-way CMP, the increase in miss rates is substantial, especially for the database workload and SPECjAppServer.

As observed for the instruction cache miss rates, we note that for the 2MB L2 cache, the L2 cache instruction miss rates of TPC-W, SPECjAppServer and the mixed workload are comparable to the L2 cache load miss rates (not shown due to space constraints). For the 1MB L2 cache, the instruction miss rate exceeds the load miss rate. Clearly, for these applications, improving the L2 cache instruction miss rates is critical to improving their overall performance.

In summary, these results show that it is important to eliminate both instruction cache misses as well as L2 cache instruction misses.

3.2 Miss categorization

In order to design effective instruction prefetching schemes that reduce the number of instruction misses, it is necessary to understand why these misses occur. In particular, it is important to understand whether these misses are sequential or the result of a change in control flow. Control flow changes can be due to branches, function calls or traps. Branches can be divided into conditional and unconditional, with the conditional branches being subdivided into taken(t) and not taken(nt). Furthermore, the conditional taken branches can be further subdivided into taken forward(tf) and taken backward(tb). In the SPARC ISA, all branches are PC relative and therefore their targets can be trivially computed. In this ISA, function calls are implemented using call, jump and return instructions. The call instruction is direct while the jump and return instructions are indirect (i.e. their targets are computed from registers).

Figure 3(i) shows the breakdown of instruction cache misses by category. Sequential misses account for only 40-60% of the misses. Branches account for 20-40% of the misses and function calls account for 15-20% of the misses. In contrast, traps account for a negligible fraction of the misses. Among the misses due to branches, conditional taken forward(tf) branches are the most prevalent, followed by unconditional branches. Among the misses due to function calls, those due to the call instruction are most prevalent. Note that the target of the call instruction is embedded within the instruction and does not need to be predicted.

Figure 3(ii) shows the breakdown of L2 cache instruction misses in a single core processor by category, while Figure 3(iii) shows the same breakdown for the 4-way CMP. Similar to the behavior of the instruction cache misses, se-

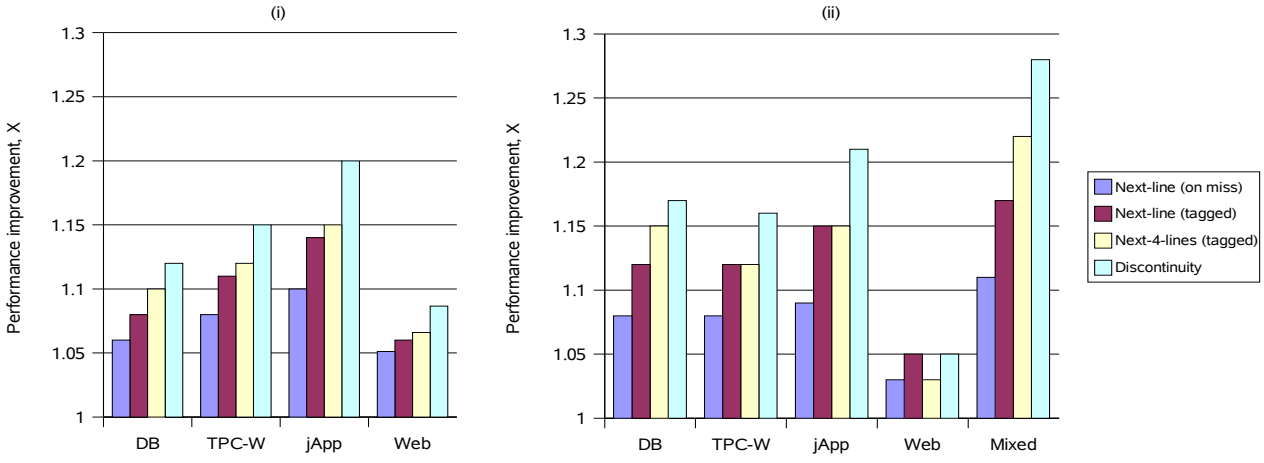

Figure 6. Performance gains achieved by different HW prefetching schemes; (i) single core and (ii) 4-way CMP

quential L2 cache instruction misses account for only 40-60% of the misses and branches and function calls account for almost all of the rest of the misses.

These results show that it is insufficient to eliminate only sequential instruction misses. It is also important to eliminate misses due to branches, as well as function calls. Also, it is interesting to note that there are more misses due to branches than misses due to function calls, curtailing the performance benefits of previous prefetching schemes [8].

3.3 Instruction prefetching potential

Figure 4(i) shows the performance improvement achievable by eliminating different types of instruction cache misses for a single core processor, while Figure 4(ii) shows the same results for the 4-way CMP. The details of the processor model simulated are described in Section 5. The results confirm the observations in Section 3.2, that it is important to eliminate more than just the sequential misses. While eliminating sequential misses alone improves performance more than eliminating branch misses alone or eliminating function call misses alone, vast performance improvements can be obtained by eliminating all three types of misses, especially for TPC-W, SPECjAppServer and the mixed workload.

These results show that an effective instruction prefetching scheme must address sequential and branch, as well as function call misses. They also show that there is great potential for improving the performance of these applications by effective instruction prefetching.

4 A discontinuity prefetcher

From the previous section it is apparent that, in order for an instruction prefetcher to be effective, it must target more than just the sequential instruction misses. While sequential misses are significant, Figure 3 illustrates that up to 60% of the instruction misses in key commercial applications are non-sequential. Furthermore, these non-sequential misses are not all attributable to the occurrence of a specific event, but rather are the result of a wide variety of control transfer instructions (CTI), including branches and function calls. Consequently, a HW instruction prefetcher is required that, in addition to effectively tackling sequential misses, can also eliminate a wide variety of CTI induced instruction misses. We utilize a “discontinuity prefetcher” to target these misses.

CTI instructions cause a ‘discontinuity’ in the instruction fetch stream, and a transition to a non-sequential address. From the perspective of an instruction prefetcher, discontinuities that cause transitions to non-sequential addresses within the same cache line can be ignored.

We propose utilizing a predictor that keeps track of previously observed discontinuities, in a manner reminiscent of the target prefetcher [1]. As execution progresses, this predictor is probed and prefetches are issued for upcoming discontinuities. Since a significant component of the stalls are a result of accesses that miss in the L2 cache, it is necessary to launch the discontinuity prefetches as early as possible, in order to cover the several hundred cycles of memory latency. To facilitate this, rather than simply probing the discontinuity prediction table with the address of current cache line, as done in previous schemes [1], the discontinuity prefetcher is probed using cache line addresses

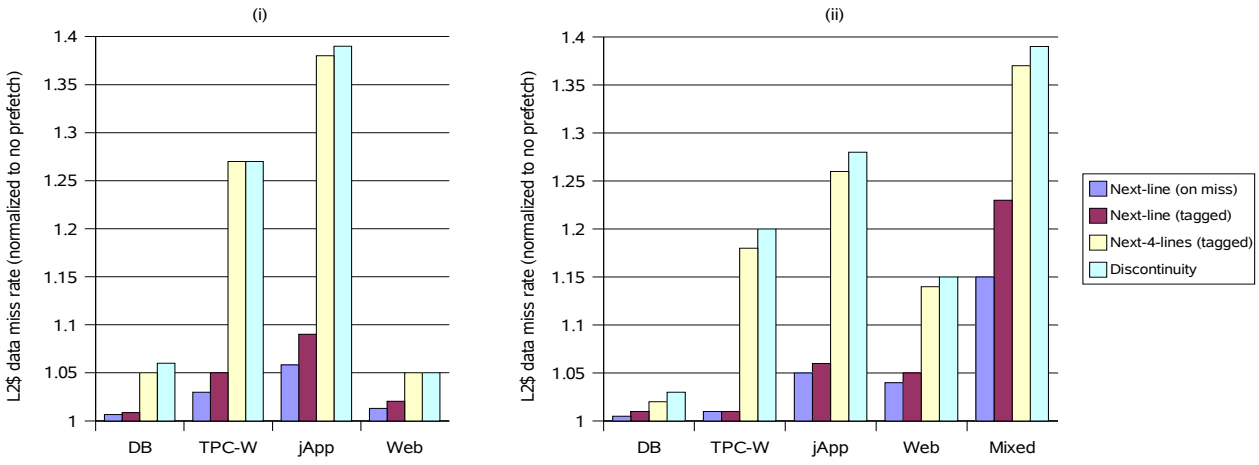

Figure 7. L2 cache data miss rate; (i) single-core and (ii) 4-way CMP

up to a defined prefetch-ahead distance (i.e. if the current PC refers to cache line L, the discontinuity predictor may be probed with L, L+1, L+2 etc.).

Analysis for a variety of commercial workloads found that, for the majority of discontinuities, for any one start address (cache-line granularity), there is just one associated target (not shown due to space constraints). Consequently, the discontinuity prediction table is implemented as a direct mapped structure with one target per entry (significantly reducing its size compared to other previously proposed predictors, which retain space for multiple targets per entry [8]). The table is managed as follows:

1. **Allocation:** when a discontinuity occurs in the fetch stream and this transition results in an instruction cache miss, the discontinuity becomes a candidate for insertion into the prediction table. If the transition is not already present in the table, an attempt is made to enter the discontinuity information into the prediction table (see replacement). The table is indexed using a portion of the address of the triggering cache line. Each table entry contains the address of the target cache line and a 2-bit saturating eviction counter, which is used to aid table management. On allocation, this counter is set to its upper saturated value.

2. **Prediction:** the discontinuity predictor is probed by the sequential prefetcher moving ahead of the demand fetch stream. If a probe locates a valid entry in the table, a prefetch is issued for the potential target of the discontinuity. Additionally, to maximize prefetch timeliness, prefetches are also issued for the remainder of the prefetch-ahead distance following the target line – it will be too late to completely cover L2 cache misses if the prefetcher waits until the discontinuity prediction is verified before issuing additional prefetches for the predicted discontinuity. Once issued, prefetches reside in the prefetch queue until resources are available to launch the prefetches. If the prefetch is subsequently found to be useful, the eviction counter is also incremented.

3. **Replacement:** when a discontinuity that doesn't have an associated entry in the table causes an instruction cache miss, this entry becomes a candidate for insertion, potentially evicting an entry already in the table. To avoid thrashing of the prediction table and the eviction of useful entries by stray events, an entry can only be evicted when its associated eviction count reaches zero. Every time an unrepresented discontinuity occurs, the counter associated with the existing entry is decremented. If this counter reaches zero then the existing entry can be evicted and a new discontinuity inserted.

We paired the discontinuity prefetcher with the next-N-line sequential scheme to provide low-cost and effective coverage for the sequential misses, removing the requirement to retain information about sequential misses in the prediction table (significantly reducing the table size). After experimentation, we found that a prefetch-ahead distance of 4-lines represented a good balance between prefetch timeliness and prefetch accuracy (with our modeled off-chip bandwidth constraints).

4.1 Filtering

Aggressive prefetching schemes such as next-N-line and discontinuity prefetching can generate a significant number of prefetches. As we want to minimize the HW cost of the prefetcher, duplicating the instruction cache tags was

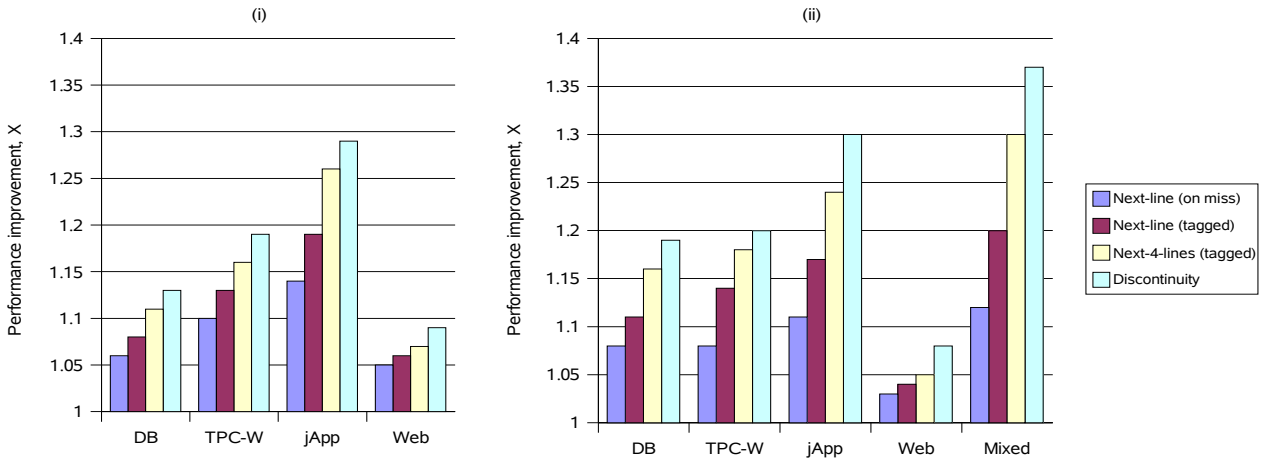

Figure 8. Performance gains achieved by different HW prefetching schemes (with L2 cache bypass prefetches); (i) single core and (ii) 4-way CMP

not considered to be cost-effective. Consequently, the instruction prefetches must contend for access to the instruction cache tags along with demand fetches. Due to the low priority of the prefetches with respect to demand requests, prefetches only obtain access to the tags when a core has no demand fetch to issue (due to the fetch width of modern microprocessors, even when not stalled, there is sufficient tag bandwidth available for modest, filtered, prefetch probing).

After a prefetch is generated by the prefetcher it is sent to the core's prefetch queue until such time as there is a free time slot to inspect the instruction cache tags (the queue is managed on a last-in, first-out basis to de-emphasize the older prefetches). As the prefetch queue is of finite size, it can readily overflow if the prefetcher is generating prefetches faster than they can be issued (the oldest prefetches are dropped first). Additionally, if the prefetch queue becomes clogged with useless prefetches, this can delay the issue of useful prefetches, impacting their timeliness. It is therefore of overwhelming importance to filter the prefetches generated by the prefetcher, rather than just allowing the inspection of the tags to perform this function.

This filtering is achieved by keeping track of the most recent demand fetches and checking each prefetch prediction against this list. If it is determined that the prefetch corresponds to a line that has been recently demand fetched, the prefetch is dropped. Similarly, while the prefetches reside in the prefetch queue waiting to be issued, every subsequent demand fetch is compared with the unissued entries and any matching entries are marked as invalid.

Additionally, before a prefetch is inserted into the prefetch queue, the queue is checked for other entries that correspond to the same cache line i.e. the prefetch queue can never contain duplicate prefetches. Furthermore, unused entries in the prefetch queue are used to retain information about previously issued and invalidated prefetches. Consequently, the prefetch can match with an entry that is i) waiting to be issued, ii) already issued or iii) invalidated. If a prefetch for this line has been invalidated or already issued, the new prefetch is dropped. If the prefetch is still awaiting issue, it is hoisted to the head of the queue.

Using this filtering mechanism, the vast majority of the unnecessary prefetches are dropped prior to accessing the cache tags (after filtering, for up to 90% of prefetch tag accesses, the desired line is not found in cache and a prefetch is issued), minimizing pressure on the tags. The performance implications of the filtering mechanism were observed to be extremely minor.

5 Methodology

In Section 3, we described the four commercial applications used in this study. The traces used in our simulations were generated from highly optimized binaries on which aggressive link-time code layout has been performed (these binaries are used by our company for reporting benchmark results). The traces were collected when the workloads were warmed and running in steady state and were meticulously validated against hardware counter statistics.

For our simulations, we used the first 50M instructions in the trace to warm the caches, as well as the processor pipeline structures, and the next 100M instructions to collect statistics. All four applications are transaction-oriented and do not exhibit phase changes. We found that 50M instructions are sufficient for warming the L2 cache and 100M instructions are sufficient for collecting a representative transaction mix that enables accurate statistics collec-

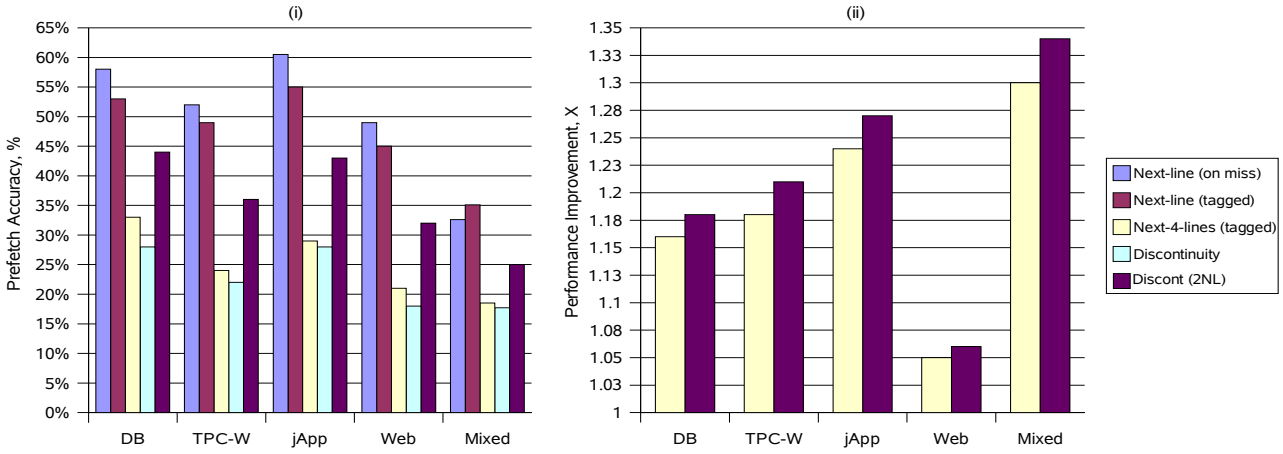

Figure 9. (i) Prefetch accuracy (4-way CMP) and (ii) Performance improvement for a next-2-line discontinuity prefetcher (4-way CMP)

tion.

The processor simulator used is a cycle-accurate timing simulator that models either a single-core out-of-order processor or a 4-way CMP with four out-of-order processor cores. Each out-of-order (OoO) core is 8-wide fetch, 3-wide issue, has a 64 entry issue window, a 64 entry reorder buffer, and a 16-stage pipeline. The L1 instruction and data caches are each 32KB (4-way set associative, 64-byte line size, 4 cycle latency) and the unified L2 cache is 2MB (4-way set-associative, 64-byte line size, 25 cycle latency). In the 4-way CMP, the L2 is shared by the four cores. The modeled memory latency is 400 cycles. The cores are assumed to run at 3GHz and the off-chip bandwidth for the single-core processor is 10GB/s, while the off-chip bandwidth for the 4-way CMP is 20GB/s (for comparison, the off-chip bandwidth for IBM POWER5 systems is approximately 25GB/s, while it is approximately 4GB/s for HP Itanium systems [19]). The primary instruction and data TLBs are 128 entry 2-way set-associative, while the secondary unified TLB has 2K entries. The conditional branch predictor is a 64K entry gshare predictor, the branch target buffer (BTB) has 1K entries and is direct-mapped and tagless, while the return address stack (RAS) has 16 entries. While the simulator is trace-driven, it is augmented with an instruction set emulator. Using the register and memory values from the traces, the emulator allows the simulator to accurately model control and data speculation, thereby achieving the accuracy of execution-driven simulation.

The chosen performance metric is instructions per cycle (IPC) and we present the percentage performance gains relative to the baseline system that does not support a HW prefetcher. Additionally, no software instruction prefetching is performed in the applications.

In addition to investigating the performance of the discontinuity prefetcher, we also look at a variety of other HW prefetchers which predominantly target sequential misses, including next-line on miss, next-line tagged and next-4-line tagged. We note that the next-4-lines prefetcher has the potential to eliminate branch misses as well as sequential misses, since most taken forward branches (shown in Section 3.2 to be the majority of branch misses) have targets that are within four cache lines of the current cache line.

The discontinuity predictors are per-core and each predictor has 8k-entries, although it is later shown that significant miss coverage can still be achieved with as few as 256-entries. Each core has a 32-entry prefetch queue and each core also keeps track of its last 32 demand fetches for the filtering previously described.

6 Results

Figures 5(i), 5(ii) and 5(iii) illustrate the decrease in instruction miss rate (for both the L1 and L2 caches) achieved by each of the HW prefetching schemes under analysis. It is apparent that the more aggressive schemes are especially effective at eliminating instruction misses and the combination of the discontinuity predictor and the next-4-line sequential prefetcher can eliminate the vast majority of the misses. In addition, the more aggressive prefetching schemes are even more effective on the 4-way CMP than on the single core processor, which helps mitigate the higher original miss rate of the 4-way CMP.

Given this significant reduction in instruction misses, we would expect significant increases in the performance of the commercial workloads under consideration. Figures 6(i) & 6(ii) illustrate the performance improvements achieved by

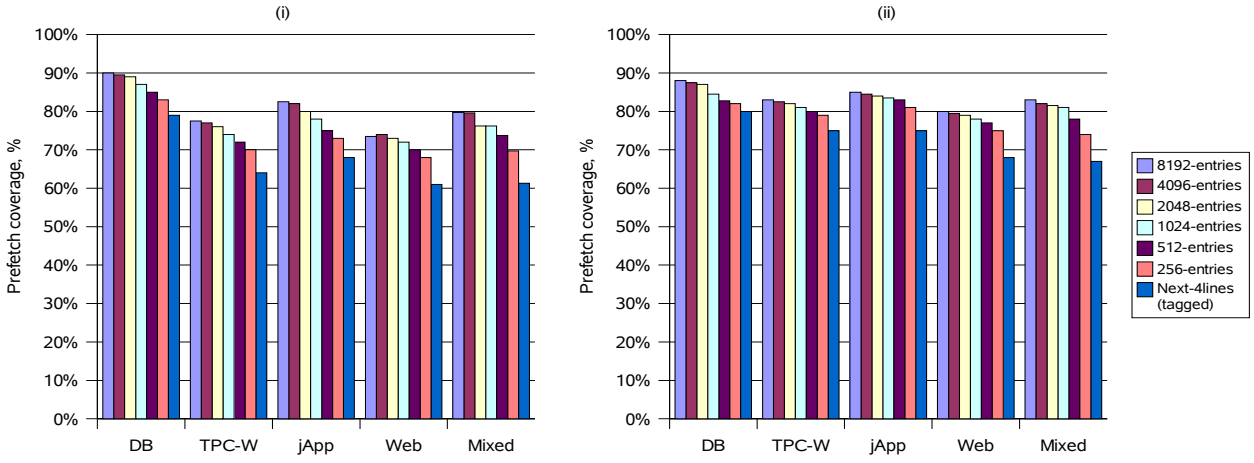

Figure 10. Prefetch coverage achieved with various sizes of the next-4-line discontinuity predictor; (i) L1 cache (ii) L2 cache (4-way CMP)

each of the prefetching schemes, and, given the effectiveness of the prefetchers, the performance improvements are significantly less than suggested by the initial limits study (see Figure 4).

A major cause of this discrepancy can be found by analyzing the L2 cache data miss rate for the applications. From Figures 7(i) & 7(ii), it is apparent that the L2 data miss rates increase significantly when aggressive instruction prefetchers are utilized. This is a result of the additional cache lines brought speculatively into the L2 cache by the instruction prefetching schemes. As the L2 cache is a unified cache (i.e. contains both instruction and data lines), the increased instruction presence leads to increased data evictions and a corresponding increase in the data component of the L2 cache miss rate. Consequently, while the instruction prefetching schemes are extremely effective at eliminating instruction cache misses, the increase in data L2 cache misses effectively counterbalances the resulting performance improvements and leads to the situation where there is overall little performance improvement achieved by the instruction prefetchers.

Therefore, a method is required to reduce this L2 cache pollution.

7 Reducing L2 cache pollution

From the previous section, it is apparent that a significant number of the instruction misses can be eliminated by utilizing aggressive prefetching techniques. Unfortunately, the pollution of the L2 cache mitigates the benefits of the instruction prefetching to a significant extent. To address this problem, we modified the instruction prefetches to initially bypass the L2 cache, the line only being installed in the L2 cache iff the prefetched line proves to be useful. During the time the prefetched line is resident in the instruction cache, its use is tracked using the same infrastructure utilized to perform prefetch tagging. If, on eviction, the prefetched line has been used, the line is installed in the L2 cache. As a result, no useless prefetches are installed in the L2 cache, eliminating the previous pollution problem.

The effectiveness of this technique is evident in Figures 8(i) & 8(ii). Compared to Figure 6(ii), the performance improvement of the discontinuity prefetcher (relative to no prefetching) in the 4-way CMP increases from 1.05X-1.28X to 1.08X-1.37X. Also, comparing Figure 8(i) with Figure 8(ii), the performance improvement of the aggressive prefetchers is generally greater in the 4-way CMP than in the single core processor.

Figure 9(i) illustrates the prefetch accuracy for the different HW prefetchers analyzed. Predictably, the prefetch accuracy is noticeably lower for the more aggressive prefetchers. Even though inaccurate prefetches can no longer pollute the L2 cache, they still consume off-chip bandwidth, potentially delaying other useful prefetches. Other than its imperfect (albeit very good) coverage, this is another reason why the discontinuity prefetcher does not fully achieve the potential performance gains shown in Section 3.3. However, every realistic prefetcher must inevitably trade-off coverage and timeliness against accuracy.

Figure 9(i) also shows that significant improvements in the accuracy of the discontinuity prefetcher can be achieved by reducing the prefetch-ahead distance (and unfortunately, timeliness) to next-2-lines (discont (2NL)). Additionally, Figure 9(ii) shows that the next-2-line discontinuity prefetcher achieves superior performance to the next-4-line (sequential) prefetcher. It does so while attaining al-

most 50% higher accuracy. As a result, in environments where off-chip bandwidth is constrained, the next-2-line discontinuity prefetcher may be a good choice.

Finally, Figure 10 illustrates the miss coverage that is achieved for different sizes of the discontinuity prefetcher's prediction table. While larger prediction tables achieve better coverage of both L1 and L2 cache instruction misses, smaller prediction tables still provide significant improvements in coverage with respect to that achieved by the next-4-line (sequential) prefetcher. Indeed, for all of the workloads investigated, the table size could be reduced by a factor of 4 with minimal impact on miss coverage.

8 Conclusion

In this paper, we showed that modern commercial applications, such as databases, application servers and web servers, have high instruction miss rates at the L1 as well as L2 levels, especially when run on CMPs, and that effective instruction prefetching is imperative to mitigate the performance loss due to these misses. We showed that it is necessary to address all types of instruction misses, and not only sequential misses, as is done in current commercial processors. Toward this end, we presented an effective discontinuity prefetcher that addresses all types of instruction misses and successfully reduces the miss rate to only 11%-16% of the original.

Additionally, we illustrate that, with a simple prefetch filtering scheme, the tag bandwidth required by the prefetches can be reduced sufficiently to be supported without duplication of the cache tags.

We also showed that aggressive instruction prefetching in CMPs can cause pollution in the shared L2 cache and increase the L2 data miss rate, thereby offsetting the performance gain resulting from the reduction in the number of instruction misses. To solve this problem, we proposed and evaluated a scheme that does not install instruction prefetches into the L2 cache until they are proven to be useful. We showed that this scheme successfully enhances the performance of aggressive instruction prefetchers.

Overall, we showed that the combination of the discontinuity prefetcher, prefetch filtering and the selective L2 installation scheme improves overall performance of the four modern commercial applications studied by 8%-37%.

References

[1] J. E. Smith and W.-C. Hsu, "Prefetching in Supercomputer Instruction Caches," in *Proc. Intl. Conf. on Supercomputing*, 1992, pp. 588–597.

[2] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, pp. 473–530, sept. 1982.

[3] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Vol. 11, pp. 7–21, 1978.

[4] T. Han, S. Kim, G. Park, O. Kwon and S. Yang, "An Improved Lookahead Instruction Prefetching," in *High Performance Computing on the Information Superhighway*, 1997, pp. 712–715.

[5] W. Hsu and J. Smith, "A Performance Study of Instruction Cache Prefetching Methods," *IEEE Transactions on Computers*, Vol. 47, pp. 497–508, May 1998.

[6] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," *IEEE Transactions on Computers*, Vol. 48, No. 2, pp. 121–133, 1999.

[7] G. Tyson, M. Charney, V. Srinivasan, E. Davidson and T. Puzak, "Branch History Guided Instruction Prefetching," in *Proc. Intl. Symp. on High-Performance Computer Architecture*, 2001, pp. 291–300.

[8] E. Davidson, M. Annavaram, and J. Patel, "Call Graph Prefetching for Database Applications," *ACM Transactions on Computer Systems*, pp. 412–444, November 2003.

[9] B. Calder, G. Reinman and T. Austin, "Fetch Directed Instruction Prefetching," in *Proc. Intl. Symp. on Microarchitecture*, 1999, pp. 16–27.

[10] B. Calder, G. Reinman and T. Austin, "Optimizations Enabled by a Decoupled Front-end Architecture," *IEEE Transactions on Computers*, April 2001.

[11] C. Lee, K. Chen and T. Mudge, "Instruction Prefetching using Branch Prediction Information," in *Proc. Intl. Conf. on Computer Design: VLSI in Computers and Processors*, 1997, pp. 593–601.

[12] J. Pierce and T. Mudge, "Wrong-Path Instruction Prefetching," in *Proc. Intl. Symp. on Microarchitecture*, 1996, pp. 165–175.

[13] C. Luk and T. Mowry, "Architectural and Compiler Support for Effective Instruction Prefetching: A Cooperative Approach," *ACM Transactions on Computer Systems*, pp. 71–109, February 2001.

[14] P. Chow, P. Hammarlund, T. Aamodt, P. Marcuello and H. Wang, "Hardware Support for Prescient Instruction Prefetch," in *Proc. Intl. Symp. on High Performance Computer Architecture*, 2004, pp. 84–95.

[15] S. Haga, Y. Zhang and R. Barua, "Execution History Guided Instruction Prefetching," in *Proc. Intl. Conf. on Supercomputing*, 2002, pp. 199–208.

[16] C. Xia and J. Torrellas, "Instruction Prefetching of Systems Codes with Layout Optimized for Reduced Cache Misses," in *Proc. Intl. Symp. on Computer Architecture*, 1996.

[17] E. Hagersten, M. Karlsson, K. Moore and D. A. Wood, "Memory System Behavior of Java-based Middleware," in *High Performance Computing on the Information Superhighway*, 1997, pp. 712–715.

[18] L. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," in *Proc. Intl. Symp. on Computer Architecture* , 1998, pp. 3–14.

[19] "STREAM: Sustainable Memory Bandwidth in High Performance Computers," in http://www.cs.virginia.edu/stream/