Call-chain Software Instruction Prefetching in J2EE Server Applications

Priya Nagpurkar[†] Harold W. Cain[‡] Mauricio Serrano[‡] Jong-Deok Choi[*] Chandra Krintz[†]
[†]University of California, Santa Barbara [‡]IBM T.J. Watson Research Center
[*]Samsung Electronics, Korea

Abstract

We present a detailed characterization of instruction cache performance for IBM's J2EE-enabled web server, WebSphere Application Server (WAS). When running two J2EE benchmarks on WebSphere, we find that instruction cache misses cause a 12% performance penalty on current-generation Power5-based multiprocessor systems. To mitigate this performance loss, we describe a new call-chain based algorithm for inserting software prefetch instructions, and evaluate its potential for improved instruction cache performance. The performance of this algorithm depends on the selection of several independent parameters which control the distance and number of prefetches inserted for a particular method. We select these parameters through characterization of the WebSphere applications, and ultimately find that our call-chain based insertion algorithm achieves significant reduction in instruction cache miss rate for Java methods.

1. Introduction

Despite an abundance of research over the years, instruction cache (icache) miss stalls remain a source of performance degradation for many commercial applications [2][7][10][11][15][17][23]. Due to the relatively larger performance cost of data cache misses in most applications, research and development has focused primarily on the data cache miss problem instead. As evidence, only a few architectures (IA-64, PA-RISC, and SPARC v9) include instruction cache prefetch instructions, and many architectures (e.g. IA-32, x86-64, and PowerPC) include no support for instruction prefetching. In contrast, all major architectures include support for software-directed data prefetching.

In theory, the icache miss problem is an easier problem to solve, because selecting blocks for instruction prefetching is solely a function of predicting control flow, while a data prefetching mechanism must also solve the considerably more difficult problem of predicting the data address that will be touched. In addition, control flow, through both branch prediction and phase behavior [20] [18], has been shown to be highly predictable. Consequently, given hardware support for software-directed instruction cache prefetching, we believe that it should be possible to significantly reduce instruction cache miss stalls for all applications (large and small).

In this paper, we present a detailed characterization of instruction cache performance across three Java server workloads running natively on an IBM Power5 multiprocessor, showing that instruction misses are indeed still a problem for large-scale server applications. We follow this characterization with a detailed study of the industry-standard SPECjAppServer2004 J2EE benchmark [12] running in a full-system simulator. As a means of reducing the performance penalty of instruction cache misses, we describe a simple algorithm for inserting software instruction prefetches at points in the call-chain that lead to specific delinquent methods, that we call *call chain-based instruction prefetching*.

Prior work discussed a call-graph based software prefetch mechanism for instruction misses which uses caller-callee relationships, inserting prefetches for a callee at the entrance of a caller [4]. In object-oriented programs which are characterized by small method sizes (such as WebSphere), this may result in late prefetches due to the short distance between the caller being invoked and its subsequent calls. We investigate moving prefetches up the call chain, inserting them at a greater distance from the callee. This leads to a trade off between prefetch coverage, accuracy, and timeliness.

Using this simple scheme for software prefetching, we show that coverage improves as we move up the call chain, resulting in better performance improvements because of more timely prefetches. However, hoisting prefetches too far can result in a decrease in useful prefetches and increase cache pollution. Our results indicate that our call-chain prefetching enables a 31% reduction in icache misses for Java code. This reduction translates into a significant reduction of the stalls caused by instruction cache misses, resulting in a 5% improvement in overall application server performance.

In summary, we make the following contributions:

- We characterize the instruction cache behavior of J2EE applications using a real machine and a system simulator, showing that the instruction cache miss problem remains significant.
- We describe an algorithm for prefetching using a call-chain, using two parameters in selecting prefetch points: confidence and miss distance.
- Using execution-driven simulation, we demonstrate the effectiveness of the algorithm in terms of accuracy, coverage, prefetch interference, and performance.

2. Characterization of Instruction Cache Behavior

This section shows that instruction cache misses are a significant source of stalls in our J2EE applications. We also show that it is difficult to attribute a significant number of stalls to a few methods, given the object-oriented nature of our target applications.

2.1. Methodology

Our study uses a two-pronged experimental setup. We first experiment with WebSphere performance running natively on a 2-socket, 4-way Power5 multiprocessor running AIX 5.3, configured with 16GB of DRAM. We use SPECjAppServer2004 and Trade6 [8], a J2EE online brokerage benchmark internally developed by IBM. These heavily-multithreaded applications run in a three-tier configuration with a DB2 8.2 back-end tier and a client/driver front-end tier. We report results for the middle-tier only, which is running the WebSphere application server, running on top of IBM's Java virtual machine [14]. Performance counters are sampled after a 15 minute warmup period, with three 30-second sampling intervals for each set of counters. We solely focus on the L1 instruction cache in our study, which is 64KB, 2-way set-associative with 128 byte cache lines. These results are presented in Section 4.

For more detailed analysis (solely on SPECjAppServer2004), we use *Mambo* [9], a full system simulator developed at the IBM Austin Research Lab. Mambo simulates the underlying hardware in enough detail that it can run the entire software stack used in our native system (AIX 5.3, J9 v2.3, WebSphere 6.1). Figure 5 illustrates this experimental methodology. Using Mambo, we gather a method entry/exit trace augmented with information about cache misses, and analyze it to select relevant methods for which prefetching will be useful, and points at which we will insert prefetches (which we refer to as *prefetch points*) for these relevant methods. The trace is gathered over the execution of 1 billion instructions (post-SPECjAppServer2004 warmup), which is also the length of our simulations.

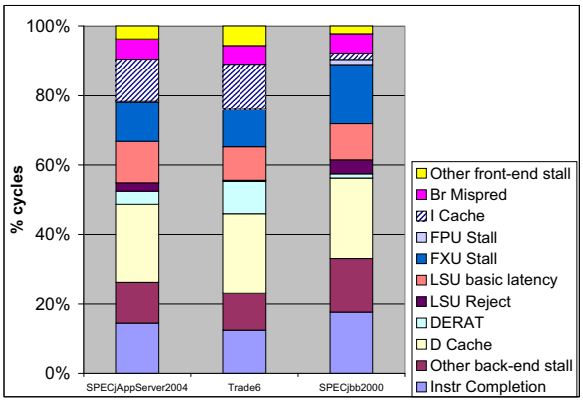

Figure 1: Commit Stall Cycle Categorization (icache miss stalls in striped bar).

2.2. Stall Cycles

The Power5 performance counter facility offers a set of counters that are incremented at each stall of the processor's commit stage, where each counter corresponds to the cause of the stall. Figure 1 shows a breakdown of stall cycles created using these counters, for SPECjAppServer2004 [12], Trade6 [8], and SPECjbb2000 [24]. Power5's 2-way SMT feature does a relatively good job of keeping the pipeline busy, however instruction cache misses still account for a significant fraction of stall cycles (12%) for both of the WebSphere J2EE applications. It has been shown that the Power5 counter mechanism actually underestimates the performance penalty of icache misses [13]. Consequently, we consider this estimate a conservative lower bound. SPECjbb2000 exhibits only a small (2%) instruction miss penalty; we have observed similar results for SPECjbb2005.

Figure 2 shows the number of L1 icache misses per 100 committed instructions, broken down by the location from which they are serviced. The vast majority of icache misses (92% and 93% for SPECjAppServer2004 and Trade6, respectively) are satisfied from the 1.8MB L2, and nearly all of the remaining misses are satisfied from the 36MB L3. An insignificant fraction of misses (less than 1%) is satisfied from memory or from remote caches from the other socket.

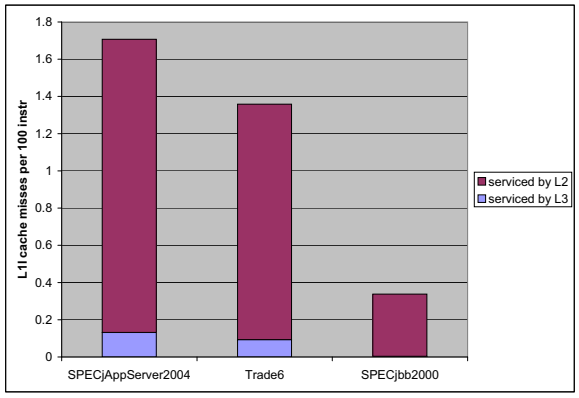

Figure 2: Icache misses per 100 committed instructions

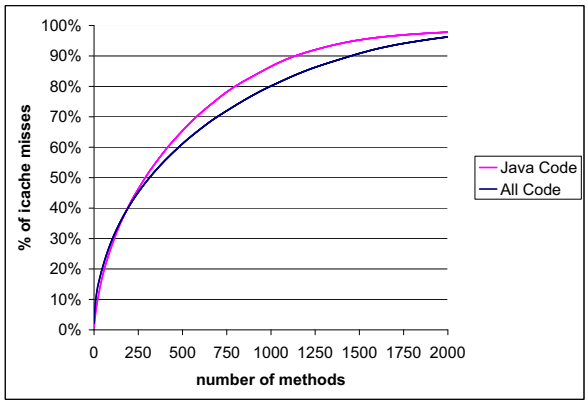

Figure 3: Per-method Contribution to Total icache Misses (cumulative distribution)

2.3. Simulation results

Using our Mambo-based simulation methodology [9], we have also collected a profile of these instruction cache misses for WebSphere running SPECjAppServer2004, mapping each cache miss back to the individual method that caused it. In terms of high-level software components, JITTED Java code accounts for the majority of icache misses (71%), followed by the AIX kernel (12%), and the J9 runtime system (7%). The remaining 10% of misses are distributed over a large set of system libraries.

When taking a closer look at the individual methods in this profile, we find that the profile is extremely flat; no single method accounts for more than 2%, and the largest contributor from the Java portion of code is merely 0.525%. Figure 3 shows the contribution of total instruction cache misses by the number of methods causing those misses. This chart includes data for all misses, and data for misses that are caused by Java code. In both cases, 50% of all misses can be attributed to the 300 worst offending methods. In order to cover 75% of all misses, more than 700 methods must be considered. Obviously, in order for an instruction prefetching mechanism to be beneficial, it must target a large number of methods.

We analyzed the top ten Java methods with the highest number of misses in more detail. Table 1 summarizes this information. The columns list percentage icache misses, average per-invocation miss count, number of instruction cache blocks that are ever actually touched, and the number of direct callers for each method. The last row shows the average values, averaged across all Java methods. Note that the fourth column refers to the number of static instruction blocks in a method that are actually used, and is an indication of the instruction cache footprint for a method..

We observe that the number of callers for a method varies and is not always small. Consequently, code positioning schemes [19] [14] might not be effective in addressing the cache miss problem, because each method often has many callers. In addition, the IBM J9 JIT [14] already optimizes code layout by reordering basic blocks such that the commonly executed code appears close in memory, thus we expect the benefits of further code reordering to be small.

Table 1: Prefetch Target Characteristics

Top Methods	ICache Misses (% of Total)	Avg. Per-invocation Misses	Num. Used Cache Blocks (Static)	Num. Direct Callers
1	0.525	4.404	20	2
2	0.468	2.833	14	94
3	0.446	54.897	64	9
4	0.436	2.720	16	3
5	0.377	61.214	64	1
6	0.350	0.844	3	7
7	0.280	16.431	16	1
8	0.277	7.100	9	1
9	0.273	1.194	21	1
10	0.259	13.044	14	2
Average (4562 total)	0.013	3.094	4.203	2.168

Our results reflect many of the facts of large-scale applications written in object-oriented styles:

- there are a large number of small methods
- each method contributes a small amount to the total execution time
- there is a substantial reuse of functionality, for example in the form of foundation libraries, resulting in multiple callers for a method

We also observe that in most cases, excluding method numbers 3 and 5, the average per-invocation misses for a method is low. This bi-modal data suggests a bi-modal optimization: for methods with a small number of blocks and small number of misses per invocation, prefetches should

be inserted for the entire method; for methods with a large number of blocks or a large number of misses per invocation, only some of the blocks should be prefetched upon entry to the method, since there will be ample time to overlap the prefetch latency within the method. We briefly discuss such a bi-modal scheme in section 4. For now, we adopt the all-or-nothing approach: at each prefetch point, we perform prefetching for all of the blocks that have been identified as useful (i.e. all of the method's blocks, excluding those blocks that are never used during our one billion instruction profile).

3. Call-chain Prefetching

Call-chain based prefetching is an all-software profile-driven prefetching mechanism which individually targets methods that cause significant icache misses. It can be utilized in both dynamic compilation environments and statically compiled applications. Beginning with an instruction cache miss profile for a particular application, delinquent methods are first chosen as prefetch targets, based on applying a threshold value. For each target method, one or more predecessor methods in the call chain are chosen as *prefetch points*, where we define the call chain as the set of methods on the execution stack when the target method is invoked. The method prologue of each selected prefetch point is augmented with instruction cache prefetch instructions for one or more target methods. These prefetch points are selected using the algorithm described here.

A prefetch must meet several criteria: it must be timely enough to overlap the latency of the prefetch request with other work before the prefetched block's use, but not so far in advance that the prefetch request is evicted from the cache prior to its use. The confidence of the request must also be high to reduce the risk of cache pollution with data that will never be used. In order to maximize confidence and timeliness, our profiling mechanism includes a means of approximating each.

Confidence

When a method is invoked, the control flow in its body governs which callsites are executed, and as a result, which methods are subsequently invoked. Consequently, different call-paths can be followed for different invocations of a method. Confidence of a prefetch point refers to the probability of reaching the target method on a call path beginning with the invocation of the prefetch point method. From the partial call graph shown in Figure 4, given a call-chain G containing a prefetch point at node A and prefetch target at node B, confidence is the probability of reaching B by following a call path starting from A. The confidence of prefetch point A in this case is 80%. When choosing

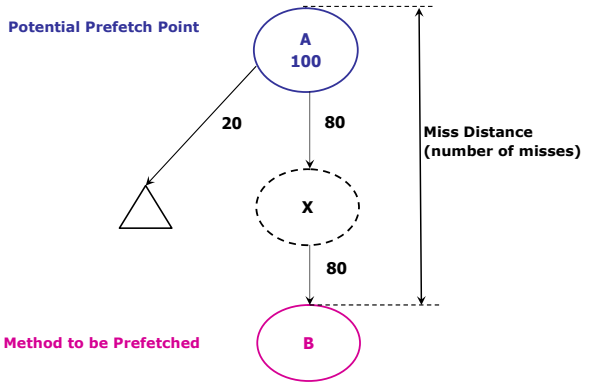

Figure 4: Example call chain. Edges are labeled with the number of calls from parent to child.

prefetch points for a method, points with high confidence values will minimize the amount of cache pollution due to bad prefetches, because it likely that a target will be called if a prefetch point is reached.

Profile collection of the confidence metric is straightforward given a call graph profile [22] [5] or calling context profile [6] [26]. To estimate confidence, we need to track method invocation counts, and the set of methods that are on the stack when a prefetch target is called. A complete call graph or calling context tree, however, is not necessary, making it possible to efficiently estimate confidence.

Miss Distance

Miss distance is a parameter we use to ensure prefetch timeliness. We define miss distance as the number of instruction cache misses on the path between the entry for the potential prefetch point and the entry for the target method. This metric is an indication of the amount of changing code and thus replacement demand on that path. When choosing prefetch points, we can selectively filter the set of candidates by measuring the average miss distance from the potential prefetch point to the prefetch target. If we require an average miss distance of at least one between the prefetch point and the target, we will usually be able to ensure that the prefetch will not be late (the miss latency can be used to fulfill the prefetch). A suitable upper bound is also chosen to avoid prefetches that are likely to be displaced prior to reaching the prefetch target.

Profile collection of the miss distance metric is slightly more difficult than the confidence metric, however there are at least two means by which it can be obtained. Using performance counter hardware that includes an instruction cache miss counter (which most processors provide), one can instrument the application to read the counter value at the entry of a prefetch point, and read again at the entry of

a prefetch target. Based on the difference of these values, miss distance is calculated. Alternatively, one can collect this profile for all points simultaneously using cache simulation, as described in Section 4.

We use these metrics to select prefetch points from predecessors in the call-chain for the method to be prefetched. Note that the prefetch points selected after applying the criteria described above form a subset of all possible prefetch points for the target method. Depending on the control flow and thresholds chosen, the selected prefetch points might not trigger prefetching for every instance of the prefetched method. For example, given a target method that is rarely called from any other methods, it is possible that no prefetch points will be chosen for the target.

4. Results

We next empirically evaluate the efficacy of our all-software approach to prefetching. We describe the methodology we used to collect the results in this section, and then present our performance data and analysis.

4.1. Experimental Methodology

For the performance data collected in this section, we utilize one of Mambo's timing simulators that models the processor and system microarchitecture of the IBM Power5. This timing model has been validated to be cycle-accurate with respect to Power5 hardware within a 2% margin of error averaged across a suite of benchmarks [25]. The publicly available attributes of the machine are detailed in prior work [21]. Most pertinent to this research is its 8-way set associative 64KB instruction cache, with 128 byte cache blocks. The instruction cache is limited to two outstanding misses, and uses a pseudo-LRU tree-based replacement algorithm. Instruction prefetch is modeled by inserting the prefetch address into a 512-entry prefetch FIFO when the trigger instruction commits. The FIFO arbitrates with the processor front-end for one of the icache's two cache ports.

Given the recent trend towards on-chip EDRAM caches, we increase the L2 cache latency from the Power5 model to 35 cycles, reflecting both EDRAM's increased latency as well as the increasing delays due to CMP arbitration logic in front of future shared L2 caches. All other simulation parameters are identical to the Power5. Due to workload setup issues, we only simulate a single-threaded uniprocessor for this evaluation. We expect the addition of multiple threads per core to only exacerbate the instruction cache problem, therefore these results should represent a conservative estimation of potential performance improvement. For each prefetch configuration, we simulate 500 million instructions

Due to our interest in building a dynamic instruction prefetching mechanism into a Java Virtual Machine, we

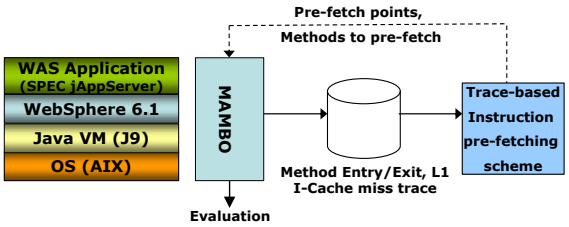

Figure 5: Trace-based Analysis Methodology

only consider prefetching for Java methods that have been compiled. Java methods also constitute the majority of icache misses, at 71%. We have implemented a trace-based algorithm to select prefetch points and studied its effect in a timing simulator. Our objective is to observe the effect that an offline profile-based scheme could have on the real machine, so that future systems may consider it as an online optimization.

For the trace-based analysis, we gather traces with method entry and exit events, annotated with icache miss counts, using Mambo. We then process these traces to find both methods to prefetch and prefetch points for these methods, using the parameters described in Section 3. For the results presented, we target the top 750 Java methods with the most L1 icache misses. Since WebSphere has multiple threads, we gather per-thread traces and aggregate the results of our analysis across threads. Our implementation of the prefetching mechanism in Mambo uses a prefetch table generated by the analysis to issue prefetches for the specified addresses at the specified prefetch points.

4.2. Evaluation

In our evaluation, we focus on the timeliness of the prefetching mechanism, and analyze the effect of several different miss distance ranges. Unfortunately, due to the large number of combinations of parameters, we are unable to collect and present sensitivity data for many combinations. We consider a miss distance lower bound of two, and experiment with six different upper bounds. The confidence threshold is fixed at 90%, which means only prefetch points with at least 90% confidence are chosen. Note that with a fixed lower bound, miss distance ranges with increasing upper bounds will include all prefetch points from ranges with a smaller upper bound. Since the lower bound is fixed, we only use the upper bound to denote the miss distance range. We also include an additional *entry* case, for which entering the method to be prefetched triggers prefetching. This case is similar to the scheme described in [4]. We report improvements with respect to the baseline, no-prefetching case. We consider three aspects: prefetch accuracy, coverage, and impact on execution time. We next describe the

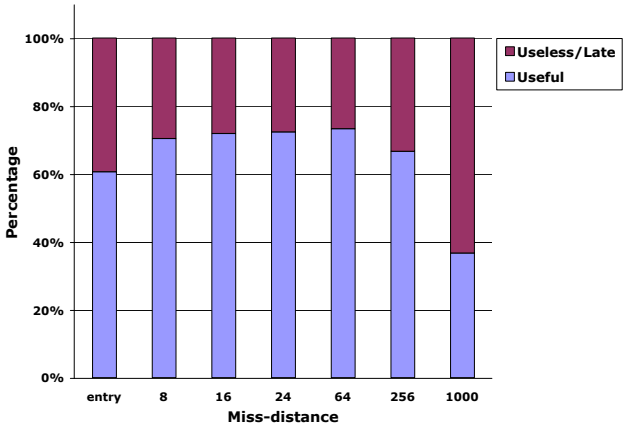

Figure 6: Prefetch Accuracy. This graph shows the percentage of Useful and Useless prefetches for different miss distance ranges.

metrics used and then present results for each of these aspects.

Prefetch Accuracy

While coverage and impact on execution time quantify the overall efficacy of the prefetching mechanism in terms of either reduction in the number of misses, or improvement in execution time, accuracy analyzes individual prefetch requests. More specifically, prefetches issued are categorized as follows:

- *Prefetch Hits* are prefetch requests that are already in the L1 icache.
- *Prefetch Misses* are prefetch requests that are not in the L1 icache and must be fulfilled.
- *Useful Prefetches* are prefetch requests that bring a block into the cache which is subsequently touched by an instruction fetch prior to its eviction.
- *Useless Prefetches* are prefetch requests that bring a block into the cache which is either unreferenced before eviction, or is requested by an instruction fetch while the prefetch miss is outstanding (untimely).

Accuracy is computed as the percentage of *Prefetch Misses* that are *Useful*. Using our current prefetching mechanism, we found approximately 20% of the total prefetches issued to be *Prefetch Misses*. In future work, we plan to reduce the number of unnecessary prefetch requests issued by eliminating redundant prefetch requests.

Figure 6 shows the percentage of *Useful* and *Useless* prefetches for different miss distance ranges. *Useless* prefetches include prefetches that are late. As expected, prefetch accuracy increases as the prefetches are hoisted farther from the method to be prefetched, owing to fewer late prefetches. However, once the miss distance reaches an upper bound of 256, accuracy begins to decline as a result of cache pollution from prefetches that are too early.

Table 2: Coverage achieved for different miss distance ranges. Coverage is the percentage reduction in misses compared to the baseline, no-prefetching case. We show coverage for methods targeted by the prefetching mechanism (top 750), coverage for Java code, and coverage for all code.

	Coverage (% Reduction in Misses)						
	Entry	**8**	**16**	**24**	**64**	**256**	**1000**
Prefetched	30.0	28.9	36.1	40.0	48.7	53.3	48.2
Java	18.5	18.3	22.9	25.5	31.4	33.6	29.5
All	12.1	12.4	15.7	17.2	21.5	21.8	17.2

Coverage

Coverage is defined as the percentage reduction in misses as a result of prefetching, compared with the baseline, no-prefetching case. We track three categories of coverage: *Prefetched*, *Java*, and *All*. *Prefetched* coverage is the coverage for methods targeted by the prefetching mechanism, *Java* coverage is the coverage with respect to all Java methods, and *All* coverage refers to coverage with respect to all code, including non-Java methods.

As a result of carrying out prefetching, there is a possibility of increasing the number of misses for some methods, especially if the number of useless prefetches is high. These methods include non-Java methods, Java methods not targeted by the prefetching mechanism, and in some cases Java methods that are targeted by the prefetching mechanism. *Interference* measures the negative effects due to prefetching, and is measured as the percentage increase in misses excluding methods that are helped by prefetching, relative to the baseline case without prefetching.

Table 2 lists coverage for different values of miss distance, whereas the graph in Figure 7 shows trends for overall coverage and interference. The three separate interference curves represent interference for non-Java code, interference for Java code, and total interference, which is the sum of the first two. Overall coverage is the achieved coverage with respect to all code, and includes interference effects (that is it incorporates both positive and negative change in the number of misses due to prefetching) Achieved coverage depends on several factors, including potential coverage and interference. As mentioned in Section 3, the set of prefetch points chosen using a particular confidence threshold and miss distance range might only target a subset of the prefetched method's invocations, and therefore a subset of the misses incurred by that method.

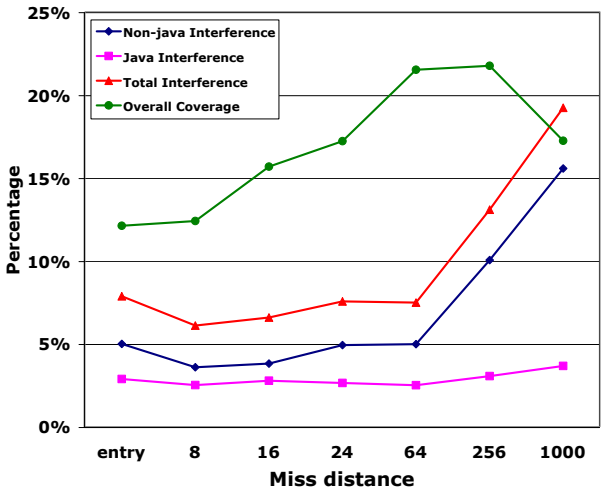

Figure 7: Effect of Miss distance on Coverage and Interference. Coverage shown is overall coverage for all code.

Potential coverage refers to the maximum number of misses in the prefetched method that could be reduced using the chosen prefetch points for it. Given a certain potential coverage, achieved coverage is further affected by interference, which in turn depends on prefetch accuracy.

From Table 2, we can see that coverage improves with increasing miss distance upper bound until the upper bound reaches the value of 1000. From Figure 7 we can see that interference is very high for this case. 64 is the best observed upper bound for miss distance, with a relatively high overall coverage of 21.5% and before the steep rise in interference. For the entry case, it is interesting to note that coverage is only 30%, in spite of the fact that potential coverage is high (since every instance of the targeted method is prefetched). This can be attributed to the large number of late or useless prefetches that this case generates.

Impact on Execution Time

Finally, we measure the net effect of our prefetching mechanism on execution time by computing percentage improvement in instructions per cycle (IPC). Table 3 lists these values for the miss distance ranges considered. With a miss distance range of [2,64], which is the best case from our results above, we achieve a 4.6% improvement in IPC. This is 2% better than the entry case. The IPC for the baseline, no-prefetching case was 0.518.

Our results verify that timeliness is an important consideration for prefetching schemes targeting the L1 icache, given the trend of increasing L2 latencies for modern processors. We show that using a simple call-chain based mechanism to hoist prefetch requests at a suitable distance

Table 3: Improvement in IPC as a result of prefetching. Improvement is calculated with respect to the base, no-prefetching case. The simulation length is 500 million instructions

% Improvement in IPC						
Entry	**8**	**16**	**24**	**64**	**256**	**1000**
2.6	2.7	3.4	3.6	4.6	5.1	3.1

from the target method, we can increase the accuracy and efficacy of prefetching. For the benchmark we analyzed, we found the miss distance upper bound of 64 to be best. We plan to explore a larger set of configurations in future versions of this work.

4.3. Discussion: Potential Improvements

In this section, we discuss two ways of increasing coverage that we have identified, with exploration of these observations planned in future work.

Callsite Interference and Incremental Prefetching

In addition to the factors affecting coverage that we discussed before (choice of prefetch points and prefetch accuracy), there is one more factor that can limit coverage. As mentioned in Section 2 we currently issue prefetch requests for all used cache blocks for a method, when a prefetch point for that method is encountered. However, if there are callsites within the prefetched method that divert execution to other methods, the prefetched blocks might be replaced before they can be used. We call this *Callsite Interference*, and propose an incremental prefetching scheme to address it. Under this scheme, we first find callsites that cause significant interference (more than a certain number of misses before returning to the caller), and partition the method into blocks before and after the callsite. We then select additional prefetch points that issue prefetches for blocks after the callsite using the same analysis parameters described before. These prefetch points must be chosen so that they issue timely prefetches for blocks that will be used after returning from the interfering callsite. For the top 750 methods targeted by our prefetching mechanism, we found that 159 of them had one or more interfering callsites using an interference threshold of 64.

Balancing Confidence and Potential Coverage

Confidence is one of the two parameters we use in selecting prefetch points. For the results presented above, we use a high confidence threshold of 90%, which means, we do not select prefetch points with confidence values lower than

90%. The aim is to minimize useless prefetches. However, by not choosing some prefetch points, we lose the corresponding potential coverage, thereby limiting the coverage that can be achieved. To minimize useless prefetches due to prefetch points with low confidence, but at the same time not ignore prefetch points that have high coverage, we considered using a hybrid parameter composed of confidence and potential coverage, instead of considering confidence alone. This hybrid parameter would allow prefetch points with relatively lower confidence values, but high coverage, to be selected, potentially improving the coverage achieved.

5. Related Work

A significant body of work has been proposed and implemented for instruction prefetching. In this section, we review the research most related to the approach that we present herein. The primary difference that sets our work apart is that it a software-only technique for reducing the overhead of instruction cache misses in server applications.

Annavaram et al. [3] [4] describe a profile-based software scheme for call graph prefetching for database applications, using both hardware and software approaches. Their software prefetching scheme [4] uses a labeled call graph to insert a prefetch instruction for the first callee; a prefetch for the second callee function is inserted immediately after the call to the first callee function, and so on. To reduce cache pollution, only the first n cache lines of a function are prefetched; the rest of the callee function is prefetched after entering the callee function. Their scheme may not be timely enough for some prefetches due to the distance between the prefetch point and the first miss, although it may achieve some partial overlapping of misses.

The authors of this work also conclude that code positioning alone is not effective enough to reduce instruction cache misses because of several factors, e.g., the number and size of small functions, the same functions invoked by multiple call sites. Regarding the latter, although code duplication or aggressive function inlining may be useful to eliminate some of the callers of a function so that code placement is more effective, the resulting code increase may result in an increase in the number of instruction cache misses. Based on the data presented in Table 1, we believe this will also be the case for WebSphere.

Spracklen et al. [23] characterize instruction cache prefetching in modern commercial applications, showing that these applications incur significant instruction cache misses. Their paper proposes a hardware scheme using both sequential and non-sequential hardware prefetchers. Sequential prefetchers using next-N-line prefetching can cover small discontinuities in the fetch stream, however it is not effective at eliminating the misses resulting from transitions to distant lines. In their non-sequential prefetcher, a basic block predictor predicts a sequence of basic blocks to be prefetched, achieving a better coverage at the cost of substantial hardware investment. Since our current scheme considers software prefetches only for the cache blocks of a method that are ever used, it is more precise than sequential prefetching; however, it may prefetch a block that may be later discarded. In future work, we plan to investigate a more fine-grain approach using control branches triggered from method calls and returns.

Luk et al. [16] present a cooperative, hardware-software approach to prefetching. The compiler aggressively inserts prefetch instructions to prefetch the targets of control transfers far enough in advance, often in multiple ways. To reduce cache pollution by the software prefetches, the hardware has a filtering mechanism to allow it to get far ahead without polluting the cache.

Other approaches to instruction prefetching include using helper prefetching threads whose only purposes is to run ahead to provide prefetching for the main thread. [1]. We do not explore this option since server applications are typically highly multithreaded and as such, the use of helper threads instead of worker threads may be a liability rather than an advantage. Another problem with this approach is the overhead of triggering helper threads and that helper threads need to run ahead enough of the worker threads to be able to hide latency.

6. Conclusions and Future Work

We presented a characterization of instruction cache performance for IBM's WebSphere Application Server and proposed a call-chain based instruction prefetching mechanism to improve cache performance of large scale server applications. We evaluated the potential of our mechanism and found a 31% reduction in icache misses for Java code (and 22% overall) by targeting only a subset of executed methods.

As part of future work, we plan to evaluate the improvements discussed briefly in Section 4. We also plan to investigate several additional enhancements to our current scheme including: considering more events, in addition to method entry, in choosing prefetch points, refining our algorithm to optimize prefetch decisions in the presence of multiple methods to prefetch, minimizing redundant or useless prefetches, and strengthening our confidence metric using context-sensitive analysis. Finally, we plan to convert our trace-based approach into an efficient, online prefetching mechanism for virtual execution environments, like Java virtual machines.

Acknowledgments

We thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by IBM Research and NSF grants CCF-0444412 and CNS-0546737.

References

[1] T. M. Aamodt, P. Marcuello, P. Chow, A. Gonzalez, P. Hammarlund, H. Wang, and J. Shen. A framework for modeling and optimization of prescient instruction prefetch. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2003.

[2] A. Ailamaki, D.J. Dewitt, M.D. Hill, and D.A. Wood. DBMSs on a modern processor: where does time go? In *The VLDB Journal*, pages 266–277, 1999.

[3] M. Annavaram, J. Patel, and E. Davidson. Call graph prefetching for database applications. In *International Symposium on High Performance Computer Architecture (HPCA)*, February 2000.

[4] M. Annavaram, J. Patel, and E. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems*, 21(4):412–444, November 2003.

[5] M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *International Symposium on Code Generation and Optimization*, March 2005.

[6] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. IBM Research Report, July 2000.

[7] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proc. of the 26th Intl. Symp. on Computer Architecture*, May 1999.

[8] IBM Trade Performance Benchmark. Trade6. `https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6`.

[9] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Hensbergen, and L. Zhang. Mambo: A full system simulator for the powerpc architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2004.

[10] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *Proc. of the Seventh Intl. Symp. on High-Performance Computer Architecture*, pages 229–240, Monterrey, Mexico, January 2001.

[11] Q. Cao, P. Trancoso, J. Larriba, J. Torrellas, B. Knighten, and Y. Won. Detailed characterization of a quad pentium pro server running TPC-D. In *Proc. of the IEEE International Conference on Computer Design*, 1999.

[12] Standard Performance Evaluation Corporation. Specjappserver2004 benchmark. http://www.spec.org/jAppServer2004/, 2004.

[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[14] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VEE)*, 2004.

[15] K. Keeton, D. Paterson, Y.Q. He, R. C. Raphael, and W. Baker. Performance characterization of a quad pentium pro smp using oltp workloads. In *Proc. of the 26th Intl. Symp. on Computer Architecture*, pages 25–26, May 1998.

[16] C-K. Luk and T. C. Mowry. Architectural and compiler support for effective instruction prefetching: A cooperative approach. *ACM Transactions on Computer Systems*, 19(1):71–109, February 2001.

[17] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proc. of the Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, 1994.

[18] P. Nagpurkar, M. Hind, C. Krintz, P. Sweeney, and V.T. Rajan. Online Phase Detection Algorithms. In *International Symposium on Code Generation and Optimization (CGO)*, 2006.

[19] K. Pettis and R. Hansen. Profile guided code positioning. In *International Conference on Programming Language Design and Implementation (PLDI)*, June 1990.

[20] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[21] B. Sinharoy, R. N. Kalla, J. M. Tendler, R.J. Eickemeyer, and J.B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49((4-5)):505–522.

[22] J. M. Spivey. Fast, accurate call graph profiling. *Software–Practice and Experience*, 34(3):249–264, 2004.

[23] L. Spracklen, Y. Chou, and S. Abraham. Efective instruction prefetching in chip multiprocessors for modern commercial applications. In *Proc. of the Eleven Intl. Symp. on High-Performance Computer Architecture*, pages 225–236, San Francisco, CA, January 2005.

[24] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000, 2000.

[25] Lixin Zhang. Personal communication, 2007.

[26] X. Zhuang, M.J. Serrano, H.W. Cain, and J.D. Choi. Accurate, efficient, and adaptive calling context profiling. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2006.