Adaptive Prefetching Java Objects

Brad Beckmann

Xidong Wang

5/13/02

Abstract

This paper describes a novel adaptive prefetching technique that prefetches memory at the object granularity instead of the usual cache block granularity. Specifically our approach focuses on prefetching Java objects through a software manager called a Prefetch Module. The adaptive nature of our approach is that the Prefetch Module dynamically decides whether to profile, prefetch, or sleep by monitoring it's prefetch prediction accuracy. For instance if the Prefetch Module detects high prediction accuracy, it will continue to prefetch, but if for low prediction accuracy, it will either decide to dynamically re-profile the application or to go to sleep. We integrated the Prefetch Module with the java.util.Hash* classes and evaluated it on five macro-benchmarks including the commercial benchmark TPC-W. Our results are very promising and show a prefetch prediction accuracy upwards of 75% and higher for some Prefetch Module configurations.

1. Introduction

The difference between the vastly increasing rate of microprocessor performance and the relatively minute increasing rate of memory system performance is allowing the processor-memory performance gap to grow rapidly. Therefore the memory performance bottleneck is increasing in magnitude. Prefetching could help relieve the bottleneck to the memory system by overlapping long latency access operations with processor computation. Rather than waiting for a miss to occur, which may stall the processor, prefetching tries to predict those misses and issues fetches to memory in advance. Therefore a prefetch can proceed in parallel with processor execution. Successful prefetching depends on two factors: accurately predicting the data objects to be referenced in the near future and issuing prefetch operations in a timely fashion. If the prefetches are issued sufficiently ahead of the demand for the data, the latency of the memory operation can be overlapped with useful computation.

Automatic prefetching techniques have been developed for scientific codes that access dense arrays in tightly nested loops. Static compiler analysis is employed to predict the program's data reference stream and to insert prefetch instructions at appropriate points within the program. However, the reference pattern of general-purpose programs, which use dynamic, pointer-based data structures, is much more complex than scientific codes. The “pointer-chasing problem”, i.e. the fact that only after an object is fetched from memory can the addresses of the objects to be accessed next be computed, is a fundamental problem of linked data structures (LDS) and recursive data structures (RDS) such as linked lists, trees, and hashtables. Therefore prefetches for the objects in these structures cannot often be computed in a timely manner. Generally speaking, static compiler analysis cannot solve the prefetch problem of a general-purpose program. The addition of runtime profiling information could help predict access patterns of a general-purpose program and then perform prefetches according to the predicted access patterns. Current profiling-aided prefetching approaches [6] focus on applications written in the C/C++ language. Similar ideas could be applied to Java applications.

Most Java programs invoke memory operations by referencing Java objects, while the granularity of memory accesses in hardware is at the word or cache-line level. Each object reference operation in Java actually involves several low-level memory fetch operations, which could be treated as one prefetch unit. Prefetching at the Java object level can minimize the amount of information needed to analyze, shrink the additional space overhead for prefetching, and improve runtime performance.

Our approach implements prefetching at the Java object level through the use of specially modified java.util modules. Specifically we attach a Prefetch Module to each instance of a java.util.Hashtable within a program. The Prefetch Module observes the object reference stream within it's Hashtable. Once it observes a certain number of references, it creates the Prefetch Table based on the observed patterns. The Prefetch Table is then used to issue prefetches for the objects believed to be accessed in the near future.

This paper describes our dynamic framework for adaptively profiling the Java object reference stream, and online detection and prediction of repetitive reference sequence. The rest of the paper is laid out as follows. Section 2 details the four phases of our approach and discusses some of the issues of our approach. Section 3 provides a brief overview of our methodology. Section 4 shows the evaluation of our approach. Section 5 describes other previous hardware and software prefetching schemes and how they relate to our prefetching approach. Finally section 6 concludes the paper.

2. Our Approach

Our approach is a purely software implementation focused on prefetching Java objects before they are needed. Our algorithm can be split into four distinct phases of operation. These four phases are: the Profile Application Phase (PA), the Prefetch Table Creation Phase (PTC), the Prefetch Object Phase (PO), and the Sleep Phase (ZZZ). The first three phases are very similar to the three phases used in [6], while the sleep phase is an additional phase added to improve the applicability of our approach by reducing it's runtime overhead during phases when prefetching has no benefit. The phase transition diagram is shown in figure 1.

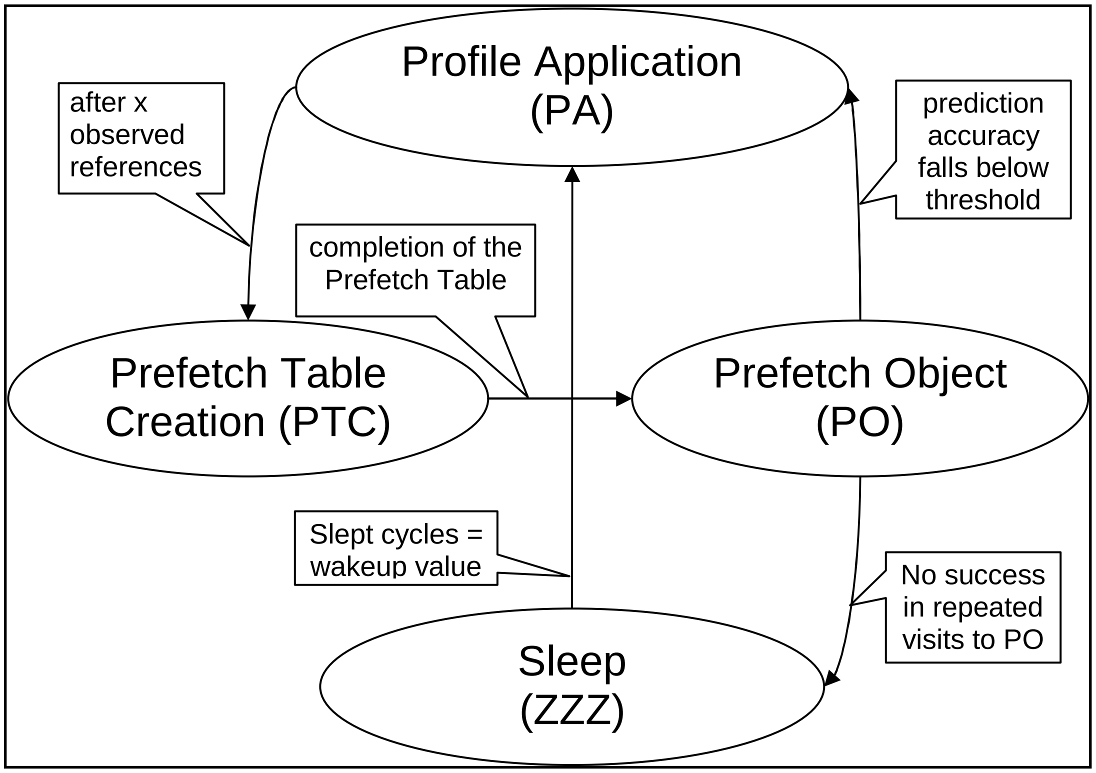

Profile Application (PA)
after x observed references
prediction accuracy falls below threshold
completion of the Prefetch Table
Prefetch Table Creation (PTC)
Prefetch Object (PO)
Slept cycles = wakeup value
No success in repeated visits to PO
Sleep (ZZZ)

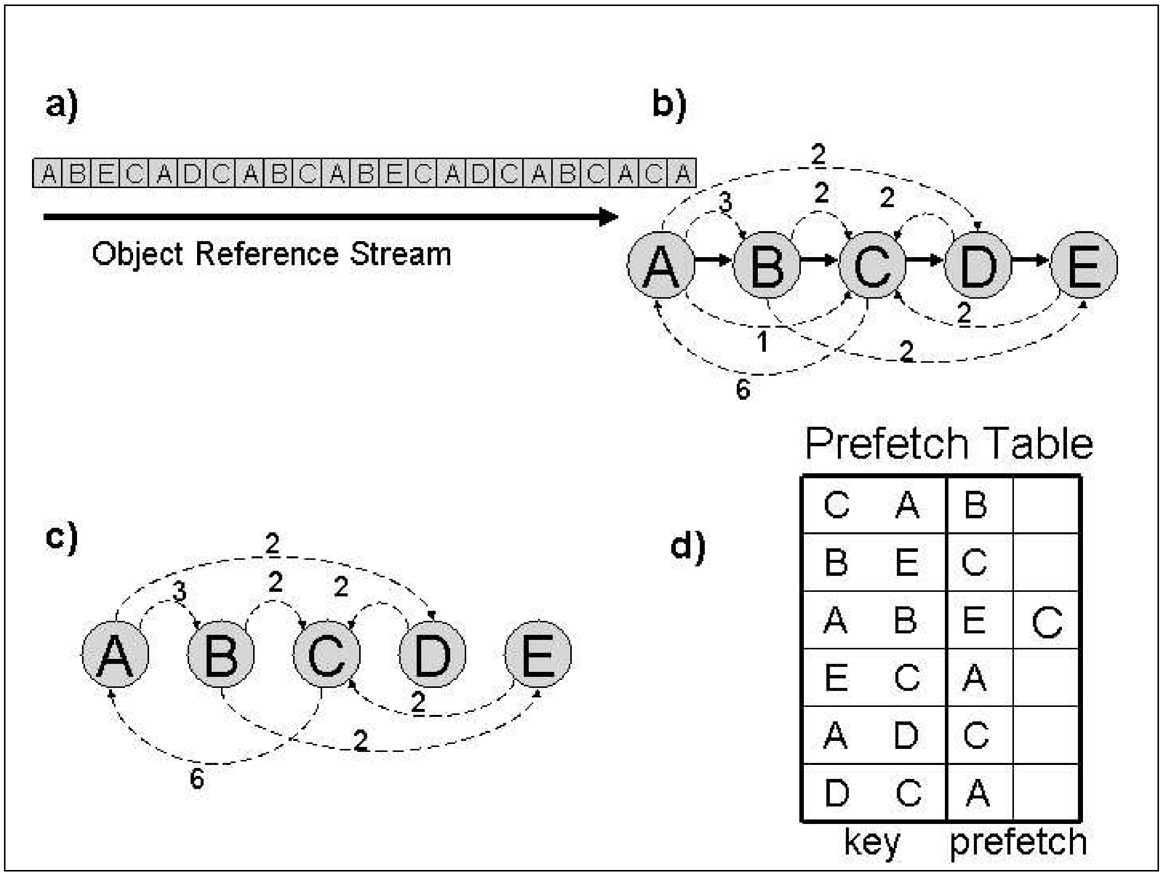
Figure 1: Prefetch Module Phase Transition Diagram

The Prefetch Module is responsible to maintain all profiling and prefetching state. Through profiling, it detects phase changes during program execution and makes corresponding actions by moving through the states of the Phase Transition Diagram.

The subsections 3.1-3.4 describe each phase in detail, while subsection 3.5 addresses the garbage collector and prefetch instruction issues of our approach.

2.1 PA phase

The profiling information collected during the PA phase is represented in a graph data structure. In this graph, a separate node represents each unique local object instance and a unique counter is associated with each directed edge. Every time a local object instance is referenced, the edge counter between the node representing it and the node representing the previously referenced object is incremented. Figure 2b shows the graphical representation of an example object reference steam given by figure 2a. One can see from figure 2b that the graph efficiently represents the general object reference behavior. For example, one may notice that a reference to object ‘D’ is followed by a reference to object ‘C’ and then a reference to object ‘A’. Therefore when the object reference series ‘DC’ is observed, the Prefetch Module can prefetch object ‘A’ with high confidence that it will be accessed next. Of course the accuracy of this prefetch algorithm is highly dependent on the temporal locality of an application’s object reference stream. Programs are known to have drastic changes in their program behavior and the reference stream in one phase of a program may be dramatically different than another phase. The ability of our approach to adjust to program behavior changes is discussed in following three subsections.



key		prefetch	
C	A	B	
B	E	C	
A	B	E	C
E	C	A	
A	D	C	
D	C	A	

Figure 2: Prefetch Table Creation

2.2 PTC phase

During the PTC phase, the information in the Profile Graph is analyzed and hot reference sequences are extracted to create the Prefetch Table. The Prefetch Table is a stripped down hashtable where the keys are a sequence of one or more object identifiers and the data returned is a set of pointers to the objects that should be prefetched. The PTC phase runs only long enough to create the Prefetch Table and like the PA, no prefetches are actually performed during this phase. The creation of the Prefetch Table is broken down to the following three steps:

1. Remove all the edges in the Profile Graph that are below a certain Extract_Threshold. The Extract_Threshold is carefully chosen so that all infrequent sequences of nodes don't expand the Prefetch Table with useless information while still guaranteing that it won't lose any hot reference sequences. For example figure 2c shows that all edges below the threshold of two are removed from the graph in figure 2b. An evaluation of different Extract_Threshold values is shown in section 4.1.
2. Search through the graph for the set (S) of all sequences with length X, where:

   X = k (key size) + d (prefetch distance) + p (prefetch group dimension)

   Therefore a sequence of length X encapsulates a long enough object reference stream so that when a sequence of k objects is observed matching the first k references of a stream from S, successful prefetches can be issued for the last p references of the stream. The actual success rate of the prefetch is dependent on the temporal locality of the application's object reference stream and that the execution time of work preformed on the d intermediate nodes is equal to or greater than the system's memory latency. One can imagine that estimating the prefetch distance in terms of the amount of work performed on an object is very difficult and changes from one application to another. However it's important to point out that it's not necessary to precisely define the prefetch distance in terms of work per object. Rather one only needs to conservatively guess a value for d that is greater than the latency to main memory while keeping X at reasonably small size.

3. Create a Prefetch Table entry for each sequence in the set S with the same first k object references. The unique identifiers of the first k objects in the sequence are used as the key and pointers to the last p objects in the sequence indicate the objects to be prefetched. Note that multiple sequences can map to the same Prefetch Table entry. Figure 2d shows an example where even if p = 1, there could be multiple objects prefetched by a single Prefetch Table access.

2.3 PO Phase

The actual prefetching of Java objects is only performed in the PO phase and therefore ideally one would like to maximize the amount of time the Prefetch Module spends in this phase and minimize the amount of time spent in the other three phases. The unique identifiers of the recent k objects are used as a key to look into the Prefetch Table and retrieve the pointers to the objects to be prefetched. Once the set of pointers has been retrieved from the Prefetch Table, the actual prefetch is performed by simply calling a basic touch() function for each object (section 2.5.2 discusses the touch function in more detail). In addition to issuing prefetches, the Prefetch Module also compares the predicted object sequence to the actual referenced object sequence to determine the recent prediction accuracy. This feedback mechanism allows the Prefetch Module to detect when the prefetch accuracy falls below a defined Accuracy_Threshold (70% by default). When the Accuracy_Threshold is met, the Prefetch Module leaves the PO phase and reenters the PA phase as shown in figure 1. An evaluation of the Prefetch Module's performance for different Accuracy_Threshold values is shown in Section 4.2.

When the Prefetch Module does not have any success after repeated visits to the PO phase, the Prefetch Module enters the fourth phase of operation, the ZZZ phase. The mechanism for detecting repeated unsuccessful visits to the PO phase is simply counting the number of successful prefetches performed before the Prefetch Module moves from the PO phase to the PA phase. If the total number of successful prefetches for the last 10 visits to the PO phase falls below a defined Success_Threshold (100 by default), the Prefetch Module slips into the ZZZ phase. An evaluation of adjusting the Success_Threshold is shown in section 4.3.

2.4 ZZZ Phase

The ZZZ or Sleep phase is where the Prefetch Module adds no noticable overhead to the application because it performs no prefetching or profiling operations. Prefetch Module enters the ZZZ phase only when it experiences very low prediction accuracy for the last X (10 by default) visits, i.e., generating little to zero successfully predicted prefetches. While in the ZZZ phase, the only state stored and maintained by the Prefetch Module is a counter of the number of object references. Once the counter reaches a certain *wakeup* value, the Prefetch Module exits the ZZZ phase, resets all state, and enters the PA phase. The *wakeup* value needs to be sufficiently large enough to skip any possible phase in which the application shows no temporal locality in its object reference stream. However the *wakeup* value needs to be sufficiently small enough so that the system won't lose prefetching opportunities in case program execution reaches a different phase. One can imagine the perfect *wakeup* value for a particular application is nearly impossible to determine. Currently the Prefetch Module rather crudely chooses the *wakeup* value to be X object references. Determining a better *wakeup* value is a subject of future work.

2.5 Other Issues

2.5.1 The Garbage Collector Issue

Java Virtual Machines, like Jikes [1], use a garbage collector to free memory after certain memory segments can no longer be used by the application. The garbage collector determines if an application can still reach a segment of memory by detecting if the application still has any pointers to that particular segment. Once the garbage collector detects that a memory segment is no longer reachable by the application's pointers, that memory is added to the free stack so that it can be used again. The prefetching approach complicates this garbage collector mechanism, because the Profile Graph and Prefetch Table save references to actual objects. Therefore even after certain objects become unreachable by the application (i.e. by setting the pointers to those objects to null), some objects may still hold memory because the Profile Graph or Prefetch Table still contain pointers to those objects. One may think that this could be a significant problem, but the following three reasons show why it is not.

1. Because each large Java object has a separate Prefetch Module associated with it, when the application resets the value of that java.util object, all the memory associated with that previous instantiation of the object including the Prefetch Module will be released to the garbage collector.
2. When the Prefetch Module exits the PA and PO phases, it releases the previous instantiations of the Profile Graph and Prefetch Table respectively. Therefore all the object pointers stored by the previous instantiations are removed as well.
3. Inevitably the Java application using the prefetching java.util objects will have a higher memory demand than the Java application running with the naïve java.util objects. However the amount of extra memory demanded by our approach is limited by the length of the PO phase. Because the PO phase only runs for a fixed number of cycles, the memory demanded by the Profile Graph and Prefetch Table data structures is limited.

2.5.2 The touch() Function Issue

As mentioned in section 2.3, a major limitation to the Prefetch Module implementation is that the actual prefetch operation is performed using a touch() function. The touch() function is simply a small function added to each small object definition and when called, fetches part of an object's memory segment. The disadvantages to this approach are that the touch() function is not guaranteed to fetch the entire object and that the memory fetches are performed using normal assembly load instructions instead of special assembly prefetching instructions. The difference between using load instructions instead of special prefetching instructions supported by the hardware is the load instructions that miss to main memory will stall the retirement of all following instruction in the reorder buffer. Current reorder buffers are not built large enough to tolerate a main memory miss without stalling the pipeline. Therefore prefetches issued with a touch() function will not overlap a main memory access with ongoing computation. A better solution would be to utilize the special prefetch instructions supported by most modern architectures that don’t stall the reorder buffer. This is a subject of future work.

3. Methodology

The Jikes RVM virtual machine [1] was modified to implement our prefetching approach. All of the code needed to support the algorithm of our approach is isolated to the java.util/ directory of the Jikes Java library. Most of that code was used to create the additional Java objects that profiled the object reference stream. There were only limited changes actually made to pre-existing Java util objects. Currently we’ve integrated the Prefetch Module with the java.util.Hashtable, the java.util.HashMap, and the java.util.HashSet classes. We believe the Prefetch Module could be easily integrated into other java.util classes such as the Vector, LinkLists, and Tree classes. Four SpecJVM benchmarks (jess, db, javac, and jack) and the TPC-W benchmark were used to evaluate the Prefetch Module integration with the java.util.Hashtable object.

4. Performance Evaluation

There are over ten independent configuration variables for the Prefetch Module and therefore a detailed evaluation of all possible Prefetch Module configurations would be impractical. Instead we defined a default Prefetch Module configuration and then conducted a performance analysis of the Prefetch Module by independently adjusting the three of the most important configuration variables. The three independent configuration variables chosen are: the

Extract_Threshold, the Accuracy_Threshold, and the Success_Threshold (Table 1 gives the default values of those three variables).

Threshold Name	Default	Usage
Accuracy_Threshold	0.7	When recent prediction accuracy is below this threshold, system will switch from PO phase to ZZZ or PA phase.
Success_Threshold	100	When amount of recent successful prefetch is below this threshold, system will switch from PO phase to ZZZ phases.
Extract_Threshold	1	The edges in Profile Graph will be removed if their weights are below this threshold.

Table 1.

4.1 Evaluation of the Extract_Threshold

The Extract_Threshold defines what edges will be filtered out of the Profile Graph before the Prefetch Table is created. Setting the Extract_Threshold to 1 will not filter out any edges because all edges in the graph must have been visited at least once during profiling or the edge would not have been created. A good value of the Extract_Threshold is highly dependent on the number of accesses profiled. Figure 3 shows the prediction accuracy of Prefetch Module for a range of Extract_Threshold values when the number of accesses profiled is set to the default of 50.

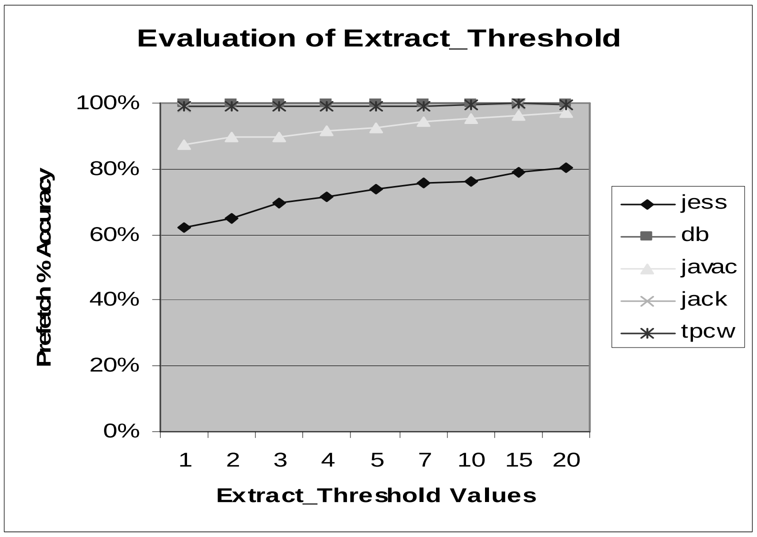


Figure 3. Evaluation of Extract_Threshold

One observes from Figure 3 that the accuracy of the prefetch predictions increases for the javac and jess benchmarks as the Extract_Threshold increases. This is due to the fact that at higher Extract_Threshold values, the Prefetch Table has higher confidence information. However there is a limitation to the benefit of increasing the Extract_Threshold value. If the value is too large, the Prefetch Module will spend a lot of time in PO phase, but will only have a few prefetch situations available to it. This eventual decrease in number of prefetches performed is shown in Figure 4 where the number of prefetches decreases after a Extract_Threshold value of 5.

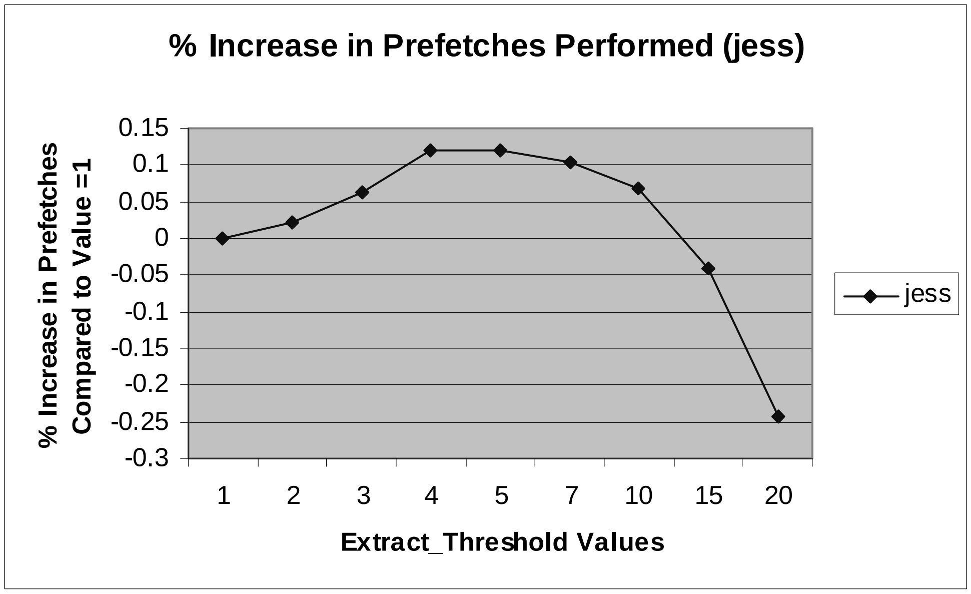


Figure 4. Percent Increase in Prefetches Relative to Extract_Threshold = 1

4.2 <u>Evaluation of the Accuracy_Threshold</u>

The Accuracy_Threshold defines how low the recent prediction accuracy of the Prefetch Module must fall before we leave the PO phase and reenter the PA phase. Prefetch Module by default waits until three prefetches have been verified as correct or incorrect before comparing against the Accuracy_Threshold. After the initial startup, the comparison is made after each prefetch verification. Setting the Accuracy_Threshold to 0% forces the Prefetch Module to stay in the PO phase after the initial profile, while setting the Accuracy_Threshold to 100% forces the Prefetch Module to leave the PO phase after only one misprediction. Figure 5 shows the total prediction accuracy for changes in the Accuracy_Threshold.

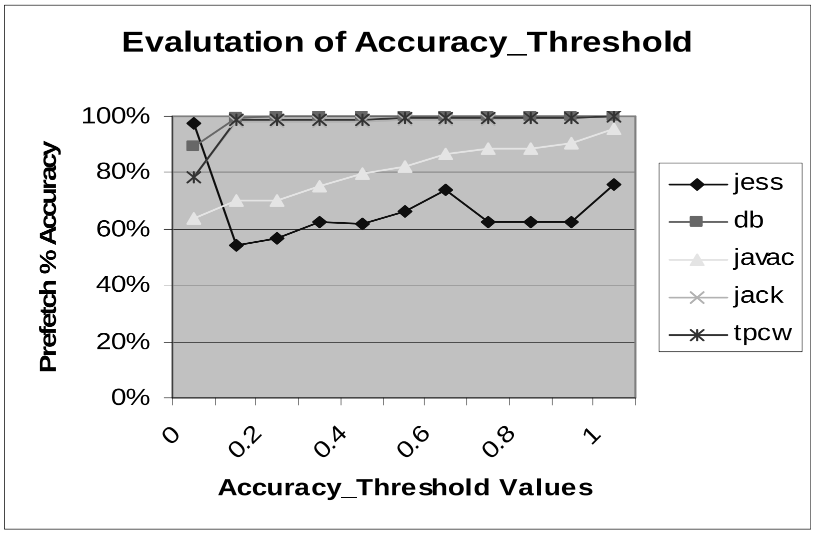


Figure5. Evaluation of Accuracy_Threshold

One observes from Figure 5 that the prediction accuracy increases as the Accuracy_Threshold increases. However the benchmark jess does fit this general trend at the higher values of Accuracy_Threshold. This is partially due to the fact that at these high values, the Prefetch module spends very little time in the PO phase (Figure 6a) and therefore the prediction accuracy numbers are based on only a few prefetches. Figure 6 shows the precent of Hashtable accesses performed while the Prefetch Module was in the PA, PO, and ZZZ phases. Note that the PTC phase is only an intermediate phase between the PA to PO transistion and therefore a Hashtable access is never performed while the Prefetch Module is in the PTC phase.

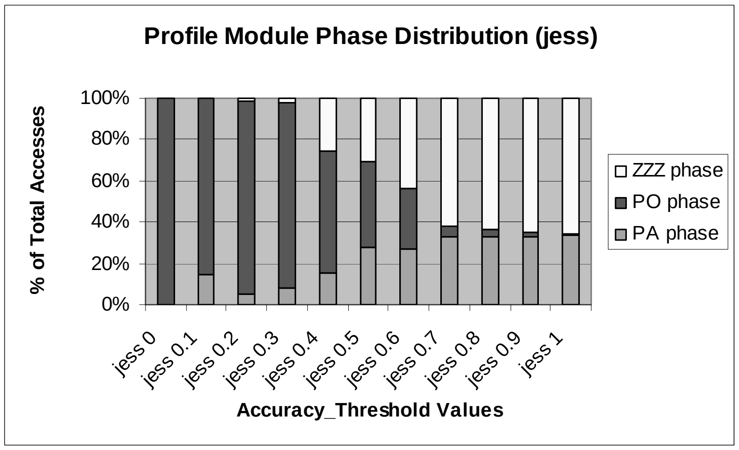


Figure 6a. Profile Module Phase Distribution for jess benchmark

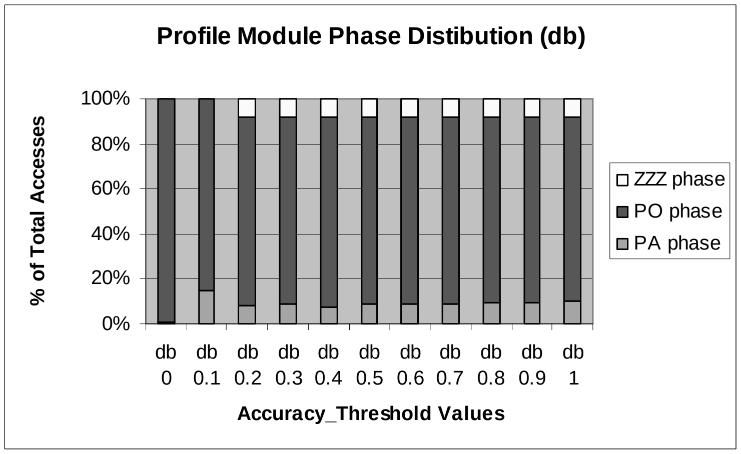


Figure 6b. Profile Module Phase Distribution for db benchmark

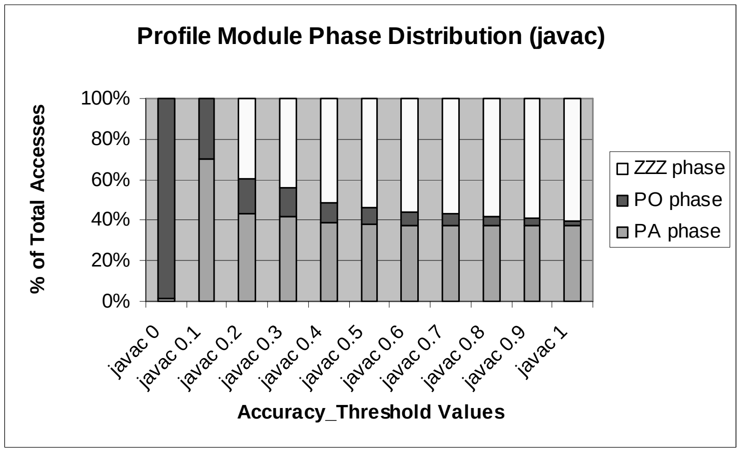


Figure 6c. Profile Module Phase Distribution for javac benchmark

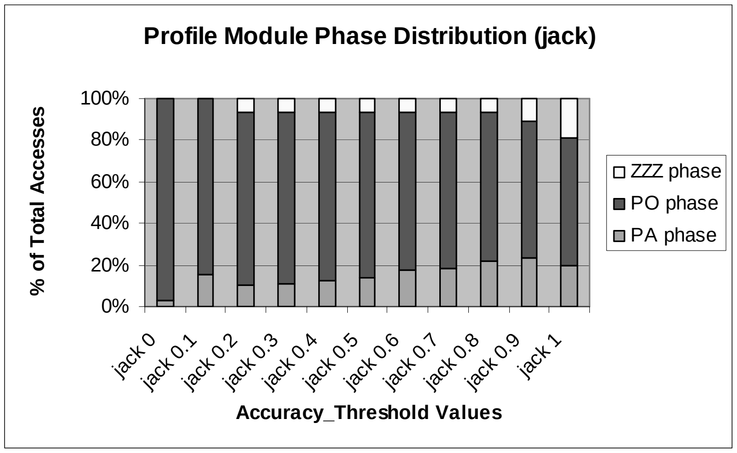


Figure 6d. Profile Module Phase Distribution for jack benchmark

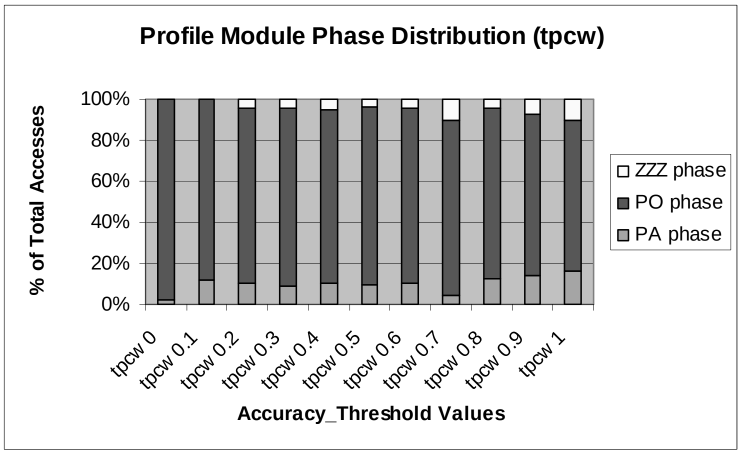


Figure 6e. Profile Module Phase Distribution for tpc-w benchmark

4.3 Evaluation of the Success_Threshold

The Success_Threshold defines the number of successful prefetches over the last 10 visits to the PO phase need to prevent the Prefetch Module from transitioning to the ZZZ phase. Increasing the Success_Threshold, increases the likelihood that when the Prefetch Module will go to sleep when it encounters a hard to predict phase of the program. Figure 7 shows this trend. Figure 8 shows that because the Prefetch Module sleeps during the hard to predict phases, it prediction accuracy slightly increases.

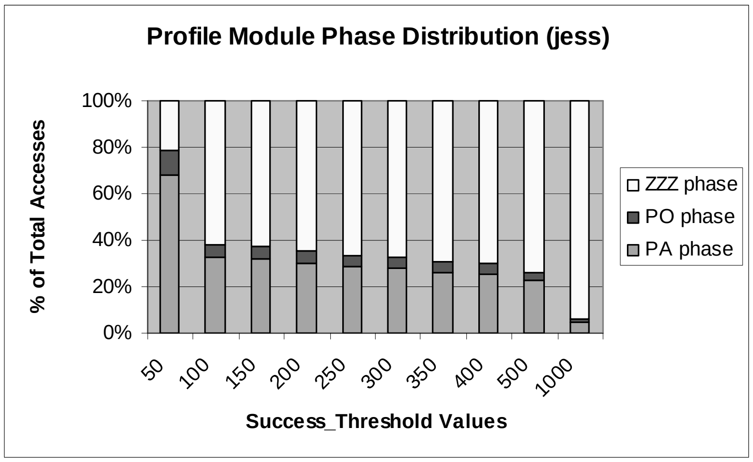


Figure 7a. Profile Module Phase Distribution for jess benchmark

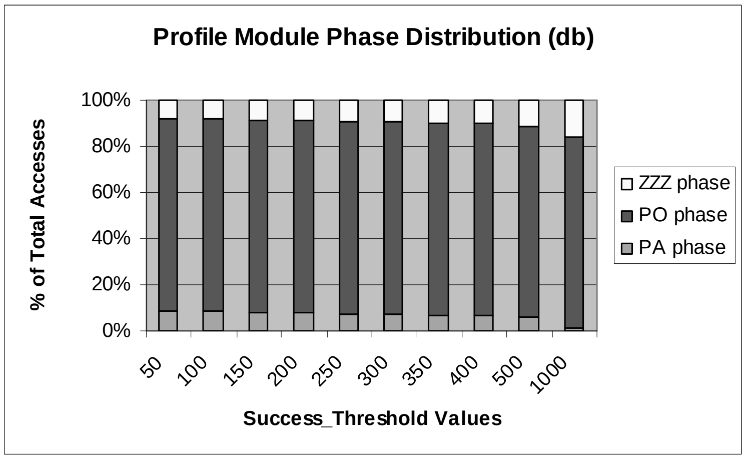


Figure 7b. Profile Module Phase Distribution for db benchmark

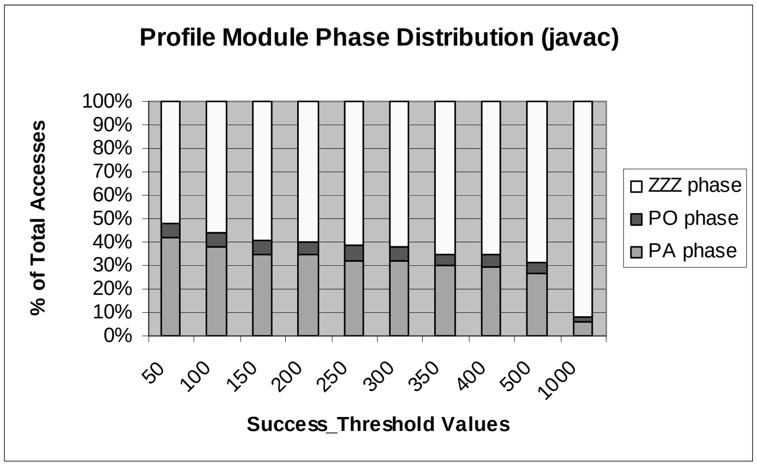


Figure 7c. Profile Module Phase Distribution for javac benchmark

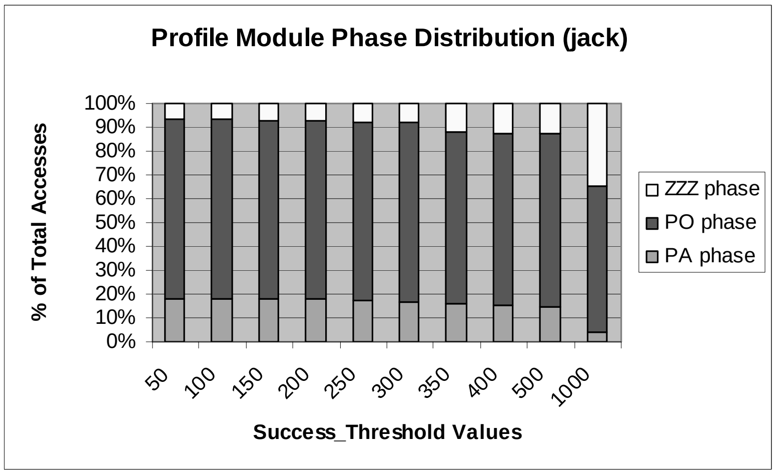


Figure 7d. Profile Module Phase Distribution for jack benchmark

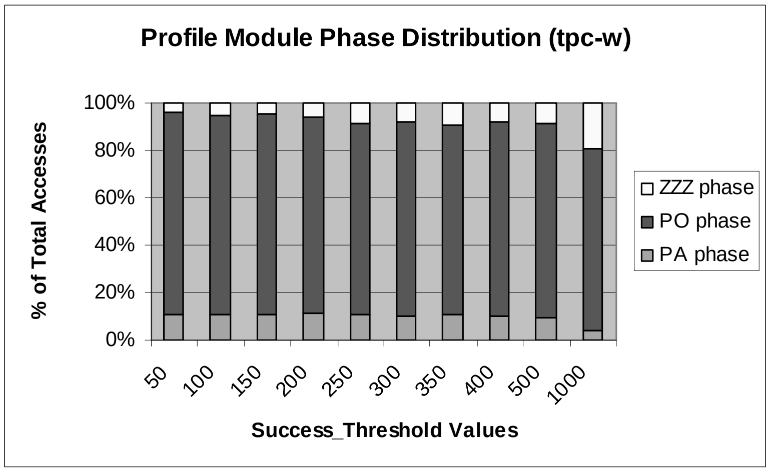


Figure 7e. Profile Module Phase Distribution for tpc-w benchmark

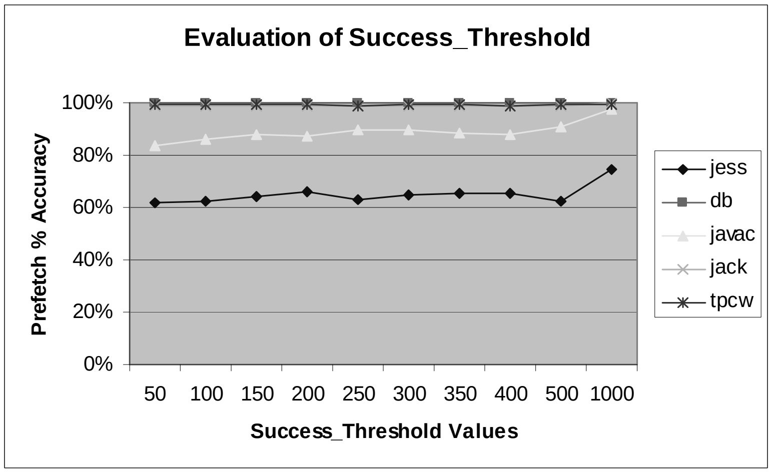


Figure 8. Evaluation of Success_Threshold

4.4 Evaluation of Runtime Overhead

As expected the runtime overhead of our approach is quite high. Table 2 compares the execution time of the five benchmarks with the unmodified Hashtable class verse the Hashtable class integrated with the default Prefetch Module. The overhead varies from 63% to .25% depending on the benchmark's utilization of the Hashtable class. Better results can be obtained by replacing the simple touch() function with a prefetch function that utilizes the specialized prefetch instructions supported by the hardware.

Benchmark	**Original exec time**	**exec time with prefetch**	**overhead**
Spec-jess	32.25s	52.6s	63%
Spec-db	36.80s	37.41s	1.6%
Spec-javac	18.87s	26.53s	40.5%
Spec-jack	7.62s	9.97s	30.8%
TPC-W	165.73s	166.15s	0.25%

Table 2.

5. Related work

Prefetching is a well-known technique used to hide latency that cause poor memory access performance. Software prefetching uses a programmer or an automatic compiler tool to instrument non-blocking load instructions into a program, while hardware prefetching performs prefetching operations by extending the architecture and issuing load instructions in advance.

Early prefetching techniques mainly focused on improving the performance of scientific codes with nested loops which access dense arrays. For such regular codes, both software and hardware techniques exist[16]. Software techniques use static compiler analysis to determine the data referenced in future loop iterations and employ program transformation optimizations such as loop unrolling or software pipelining. Hardware approaches eliminate instruction overhead and compiler analysis by supporting the prefetch capability in hardware. The most common hardware prefetching approach is stride prefetching, where cache lines following the current access are prefetched. Generally these techniques are limited to sequential access patterns in programs.

Jump pointers [13][10][4][15] are a software approach for prefetching linked data structures that overcome the regular program limitation of stride prefetching. Jump pointers can be classified as greedy prefetching or history-based prefetching. Basic greedy prefetching prefetches all objects pointed to by the current node immediately after the current node is visited and before any work is performed on the node. Greedy prefetching works when the nodes pointed to by the current node are accessed in the near future and the latency of those prefetches are overlapped with the work on the current node. Derivative chain prefetching, a more advanced type of greedy prefetching, add pointers between non-successive nodes to launch prefetches. Both ideas are based on the assumption that once an object is accessed, the objects pointed to by that object will be accessed in the near future. However, this assumption depends on regular access patterns on regular linked data structure, and the nature of static analysis makes it hard to adapt to dynamic access patterns appearing during execution. For example, the prefetching linearization idea [8] cannot prefetch trees with a high branching factor. Software data flow analysis [10][4] could help discover objects to prefetch, but that depends on regular control structures. In contrast, our approach is not based on those assumptions and therefore has no such limitation.

History pointer prefetching stores artificial jump pointers in each object. These jump pointers point to objects believed to be needed in the future based on the order of past traversals. Similar to our approach, they also depend on object reference sequence provided by previous execution. However they don’t distinguish between cold sequence and hot sequence, while our approach only aims at exploiting recurrence of hot sequences during dynamic execution. Also it only prefetches objects when a sequence of previous observed object reference stream occurs and not after a single object access, improving the prediction precision.

Data-linearization prefetching [8] maps heap-allocated objects that are likely to be accessed close together in time into contiguous memory locations. However, because dynamic remapping can incur high runtime overheads and may violate program semantics, this scheme obviously works best if the structure of a LDS changes very slowly. Cache conscious data placement [5] and memory layout reorganization optimization [11] also share the same weakness. In contrast, our algorithm works in an adaptive way, performs profiling work when the misprediction rate is high, and performs actual prefetch operations only when the misprediction rate is low and stable. Therefore our approach can adapt to faster updates to the LDS.

Various hardware techniques related to greedy prefetching have been proposed for prefetching linked data structures. In dependence-based prefetching [12], producer-consumer pairs of load instructions are identified and a prefetch engine speculatively traverses and prefetches them. The dependence idea is similar to our approach since both consider the correlations of neighboring load operations. However it uses data dependence between instructions as the information primitive, while our approach treats the whole Java object as a prefetch unit, which will save the space of representation. Also our approach is software-only and doesn't need special hardware support.

Dependence-graph precomputation [2] executes a backward slice of instructions from the instruction fetch window to compute a prefetch address. It aims to compute the prefetch address in advance but, some prefetch addresses can be value-predicted without computation. Luk [9] presents a similar idea using software to control pre-execution. However it requires simultaneous multithreading processors to generate its dependence graph dynamically, while our approach requires no such hardware support.

The hardware technique that best corresponds to history-pointers is correlation-based prefetching [14][7]. As originally proposed, this system learns a diagram of a key and prefetch addresses: when the key is observed, the prefetch is issued. Joesph and Grunwald generalized this idea by using Markov predictor. Nodes in the Markov model are addresses, and the transition probabilities are derived from observed diagram frequencies. Prefetches for a fixed number of transitions from that address issue when a data address associated to a node in the Markov model misses in the cache. The prediction is based on code address and the idea is based on assumption that same data misses will be observed on same address. However, static nature of this prediction doesn't adapt to dynamic execution. While our prediction is based on hot sequence recurrence and observation of hot sequence prefix, and can achieve better prediction accuracy.

Chilimbi et al [6] provides a solution to software prefetching for general-purpose programs. It works in three phases. First, profiling instructions are instrumented during runtime to collect a data reference profile. Then a state machine is invoked to extract hot data streams, and the system dynamically instruments code at appropriate points to detect those hot data streams and perform prefetches. After that, the system enters the hibernation phase where no profiling or analysis is invoked and the program continues to execute with added prefetch code. Their approach is thoroughly tuned to guarantee that prefetching benefit can offset the profiling and analysis overhead. Both their approach and our approach uses runtime profiling to collect a reference stream and use some algorithm to extract hot streams from profile data. They aim at C/C++ applications while our approach works for the Java applications. Also we use a graph algorithm to extract hot object sequences instead of a state machine. Another difference is that our approach creates statistics about recent prediction accuracy, and invokes the sleep phase or profile phase when no hot sequence is observed during program execution.

6. Conclusions

This paper described an adaptive prefetching technique for Java applications that prefetches memory at the Java object granularity. The central manager of this prefetch technique is the Prefetch Module that dynamically decides whether to profile, prefetch, or sleep by monitoring its own recent prefetch prediction accuracy. The Prefetch Module integrated with the java.util.Hash* classes show promising results for the five macro-benchmarks evaluated. The most promising is its high prefetch prediction accuracy, upwards of 75% and higher for some Prefetch Module configurations. Also techniques were described to keep it's overhead to a minimum while maximizing its effectiveness and applicability.

References

[1] B. Alpern, et al. *The Jalapeno Virtual Machine*. IBM System Journal, Vol 39, No 1, February 2000.

[2] Murali Annavaram, Jignesh M. Patel, Edward S. Davidson. *Data Prefetching by Dependence Graph Precomputation.* In Proceedings of the 28th International Symposium on Computer Architecture (ISCA2001). Jul 2001

[3] B. Cahoon and K. S. McKinley. *Tolerating Latency by Prefetching Java Objects*. In Workshop on Hardware Support for Objects and Microarchitectures for Java, Austin, TX, October 1999.

[4] B. Cahoon and K. S. McKinley. *Data Flow Analysis for Software Prefetching Linked Data Structures in Java*. In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, Barcelona, Spain, September 8-12, 2001.

[5] T.M. Chilimbi, M.D. Hill, and J.R. Larus. *Cache-Conscious Structure Layout.* ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 1-12, Atlanta, Georgia, May 1999.

[6] Trishul M. Chilimbi, and Martin Hirzel. *"Dynamic Hot Data Stream Prefetching for General-Purpose Programs".* to appear in Programming Languages Design and Implementation '02 (PLDI) , June 2002.

[7] Doug Joseph, Dirk Grunwald: *Prefetching Using Markov Predictors*. IEEE Transactions on Computers 48(2): 121-133 (1999)

[8] Karlsson M., Dahlgren F., and Stenström P. *A Prefetching Technique for Irregular Accesses to Linked Data Structures*. In Proceedings of the 6th international conference on high performance computer architecture, Pages 206 - 217, 2000.

[9] Chi-Keung Luk. *Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors*. In Proceedings of the 28th Annual International Symposium on Computer Architecture, Goteborg, Sweden, June 2001. ACM.

[10] C.-K. Luk and T. C. Mowry. *Automatic compiler-inserted prefetching for pointer-based applications*. IEEE Transactions on Computers (Special Issue on Cache Memory), 48(2):134--141, February 1999.

[11] C-K. Luk and T.C. Mowry. *Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation*. The 26th IEEE/ACM International Symposium on Computer Architecture (ISCA), pages 88-99, May 1999.

[12] A. Roth, A. Moshovos, and G. S. Sohi. *Dependence Based Prefetching for Linked Data Structures*. In Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems, pages 115--126, October 1998.

[13] Todd C. Mowry, Chi-Keung Luk. *Compiler-Based Prefetching for Recursive Data Structures*. In Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pages 222--233, Oct. 1996.

[14] Todd C. Mowry, Chi-Keung Luk. *Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling*. In Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 314-320, Dec. 1997.

[15] A. Roth and G. S. Sohi. *Effective Jump-Pointer Prefetching for Linked Data Structures*. In Proceedings of the 26th Annual International Symposium on Computer Architecture, pages 111--121, May 1999.

[16] S. P. Vanderwiel and D. J. Lilja. *Data Prefetching Mechanisms*. ACM Computing Surveys, 32(2):174--199, June 2000.