Prefetching Java Objects

Brad Beckmann

Xidong Wang

Abstract

(To be completed when the results section is complete)

1. Introduction

The difference between the greatly increasing rate of microprocessor performance and the relative minute increasing rate of memory system performance is allowing the processor-memory performance gap to rapidly grow. Therefore the memory performance bottleneck is increasing in magnitude. Prefetching could help relieve bottleneck pressure to 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 predicts those misses and issues fetches to memory in advance. Therefore the prefetch can proceed in parallel with processor execution. Successful prefetching depends on two factors, accurate predicting the data objects to be referenced in near future and issuing prefetch operations in a timely fashion, sufficiently ahead of the demand for the data, so that 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 insert prefetch instructions at appropriate points within program. However, the reference pattern of general purpose programs, which use dynamic, pointer-based data structures, is much more complex than scientific codes. Particularly the linked data structure

(LDS), or called recursive data structure (RDS) such as linked lists, trees, and hashtables expose the pointer-chasing problem. The pointer-chasing problem is only after an object is fetched from memory can the addresses of those objects pointed by it be computed. Therefore many times the prefetches for these objects cannot be computed in a timely manner. Generally speaking static compiler analysis technique itself cannot solve the prefetch problem of a general-purpose program. The augment of runtime profiling information could help predict access patterns of general-purpose program and perform prefetch according to predicted access pattern. Current profiling-aided prefetching approach[15] focus on C/C++ language. Similar ideas could be applied to object-oriented Java language.

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 a couple of memory fetch operations in low level, which could be treated as one prefetch unit. Therefore, prefetching at the Java object level can minimize the information needed to process, shrink the additional space overhead for prefetch, and improve runtime performance.

This paper describes a dynamic framework for adaptively profiling of Java object reference stream, online detection and prediction of repetitive reference sequence. The rest of the paper is laid out as follows. Section 2 describes other previous hardware and software prefetching schemes and how they relate to our prefetching approach. Section 3 details the 4 phases of our approach and discusses some of the issues of our approach. Section 4 provides a brief overview of our methodology. Section5 shows the results of our approach. Finally section 6 concludes the paper.

2. Related work

Prefetching is a well-know technique to hide latency that causes from poor memory access performance. Software prefetching is where a programmer or automatic compiler tool

instruments non-blocking load instructions into program, while hardware prefetching performs prefetching operations by extending hardware 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[1]. Software techniques could use static compiler analysis to determine the data reference in future loop iterations and employ program transformation optimization such as loop unrolling or software pipelining. Hardware approaches eliminate instruction overhead and compiler analysis by supporting the prefetch capability in hardware. Generally these techniques are limited to regular code and access patterns in programs.

Jump pointers [5][8][9][14] are a software approach for prefetching linked data structures that overcomes the regular program limitation of stride prefetching. Jump pointers can be classified as greedy based prefetching or history based prefetching. Basic greedy prefetching tries to overlap the latency of the prefetch for the next node in LDS, with all the work between two consecutive LDS access. Derivative chain prefetching, a more advanced type of greedy prefetching, add pointers between non-successive nodes to launch prefetches. Both ideas are based on assumption that once an object is accessed, the objects pointed by that object will be accessed in near future. A limitation of these techniques is that their static analysis is restricted to regular linked data structures accessed by local regular control structures. For example, the prefetching linearizion idea [4] cannot prefetch trees with a high branching factor. Software data flow analysis [8][9] 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 information provided by previous execution. However, our approach only prefetches objects when a sequence of previous observed object reference occurs and not after a single object access.

The data-linearizion prefetching [4] 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 LDS changes very slowly. Cache conscious data placement [6] and memory layout reorganization optimization[7] also share the same weakness. In contrast, our algorithm works in an adaptive way, performs profiling work when misprediction rate is high, and perform actual prefetch operations only when 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[2], 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 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.

In dependence-graph precomputation[3], a backward slice of instructions in the instructions fetch window is used to choose a few instructions to execute speculatively to compute a prefetch address. It aims to compute out the prefetch address in advance. However, some prefetch address can be value-predicted without computation. Luk present a similar idea while he try to use software to control pre-execution. It needs hardware support such as simultaneous multithreading processors to generate its dependence graph dynamically. While our idea is software only.

The hardware technique that best corresponds to history-pointers is correlation-based prefetching[10][11]. As originally proposed, it 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. Upon a data cache miss to an address that has a node in the Markov model, prefetches for a fixed number of transitions from that address are issued.

Chilimbi[15] provide a solution to software prefetching for general-purpose program. It works in three phases. First, profiling instructions are instrumented during runtime to collect data reference profile. Then a state machine is invoked to extract hot data streams, and system dynamically instrument code at appropriate points to detect those hot data streams and perform prefetch. After that, system enter the hibernation phase where no profiling or analysis is invoked and program continue to execute with added prefetch code. Their approach are thoroughly tuned to guarantee that profiling, analysis and profiling overhead can be offset by prefetching benefit. Both their approach and our approach choose runtime profiling to collect reference stream and use some algorithm to extract hot stream from profile data. They aims at C/C++ language while our approach work for Java language, and we choose Java object as a prefetch unit.

3. Our Approach

Again, our approach is a purely software implementation focused on prefetching Java objects before they're 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 [15], while the sleep phase is an additional phase added to our approach. The phase transition diagram is shown in figure 1.

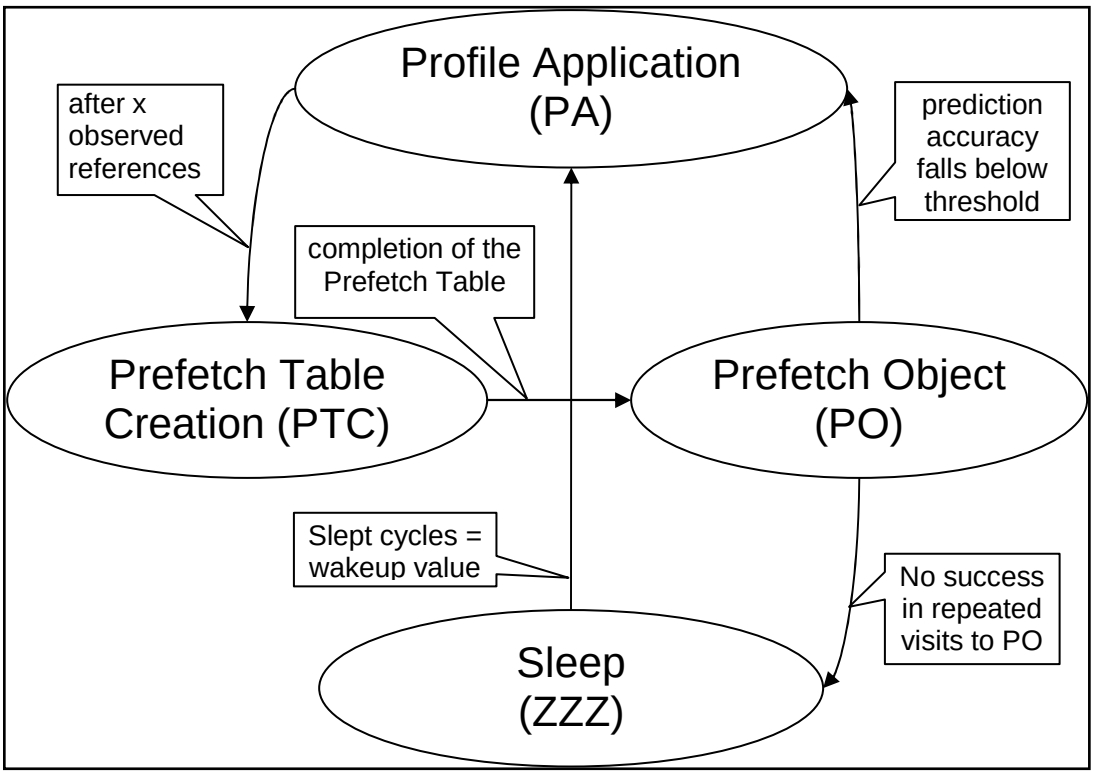

Figure 1: Phase Transition Diagram of the Prefetch Module

Prefetching prediction is enabled by on-line collecting and analyzing profiling data. A Prefetch Module is responsible to maintain all profiling and prefetching states. It will detect phase switch during program execution and make corresponding action by travelling in Phase Transition Diagram.

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

3.1 PA phase

The profiling information collected during the PA phase is represented in a graph data structure. In this graph each unique local object instance is represented by a separate node and each uni-directional edge connecting two nodes in the graph has a unique counter associated with it. 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 the example object reference steam given by figure 2a.

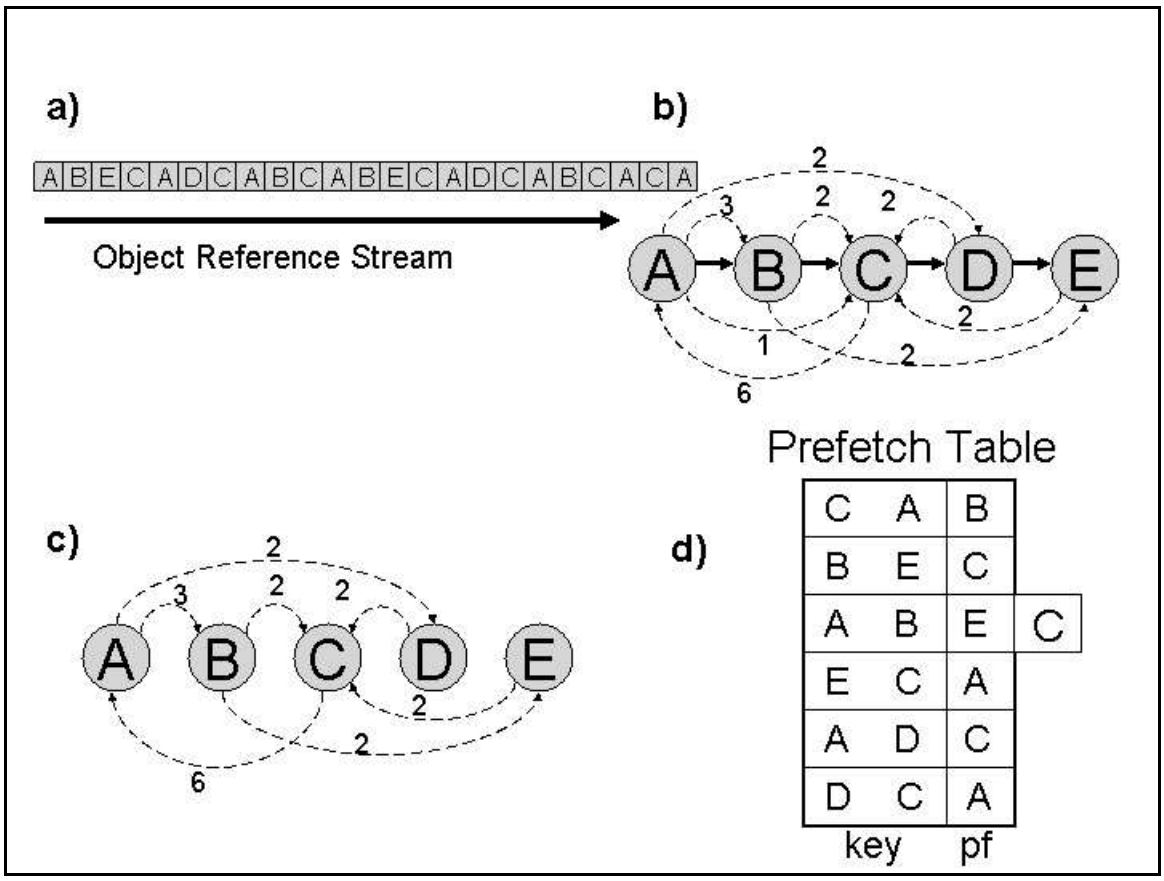

Figure 2: Prefetch Table Creation

3.2 PTC phase

During the PTC phase, the information in the Profile Graph is analyzed and hot reference sequence is extracted to create the Prefetch Table. The Prefetch Table is actually just a stripped down Hashtable where the keys are a sequence of one ore more object identifiers and the data returned is a set of pointers to the objects that should be prefetched. The creation of the Prefetch Table is broken down to the following 3 steps:

1. Remove all the edges in the Profile Graph that are below a certain threshold. The threshold is carefully chosen so that all infrequent sequences of nodes don't expand the

Prefetch Table with useless information while still guarantee that it won't lose any hot reference sequence. For example figure 2c shows that all edges below the threshold of 2 are removed from the graph in figure 2b.

2. Search through the graph for the set (S) of all sequences with length X.

   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. The execution time of work preformed on the d intermediate nodes is equal to the system's memory latency.

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.

3.3 PO Phase

The actual prefetching of Java objects is performed in the PO phase. The unique identifiers of the recent k objects are used as key to lookup 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. 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 certain threshold (say 80% by default). When the threshold is met, the

Prefetch Module leaves the PO phase and reenters the PA phase as shown in figure 1. However if 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.

3.4 ZZZ Phase

The ZZZ phase or Sleep phase is the phase where the Prefetch Module adds no noticable overhead to the application because it performs no prefetching or profiling operations. Prefetch Module is only triggered to enter ZZZ phase when it experiences very low prediction accuracy for the last X (say 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 the 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 it won't lose prefetching opportunity 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 we rather crudely chose the "wakeup" value to be X object references. Determining a better "wakeup" value is a subject of future work.

3.5 Garbage Collector Issues

Java Virtual Machines, like Jikes, 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. Our prefetching approach complicates this 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 we give the following three reasons 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.

4. Methodology

We modified the Jikes RVM virtual machine [cite jikes] 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. Only very 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.

We've tested the modified java/util/Hash* classes with various microbenchmarks. We plan to evaluate the Hash* classes with those SpecJbb benchmarks that use the classes as well as TPC-W.

5. Performance Evaluation

All the code described in this paper has been implemented and verified to work through the use of a microbenchmark. We are currently in the process of collecting data from real benchmarks. We plan to display data of the prediction accuracy, the amount of references spent in each phase, and the execution runtimes. We would like to hear your suggestions of what other results we should include.

References

[1] Data Prefetching Mechanisms

[2] Dependence Based Prefetching for Linked Data Structure

[3] Data Prefetching by Dependence Graph Precomputation

[4] A Prefetching Technique for Irregular Accesses to Linked Data Structures

[5] Compiler-Based Prefetching for Recursive Data Structures

[6] Cache-Conscious Structure Layout

[7] Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation

[8] Automatic Compiler-Inserted Prefetching for Pointer-Based Applications

[9] Data Flow Analysis for Software Prefetching Linked Data Structures in Java

[10] Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling.

[11] Prefetching using Markov predictors

[12] Tolerating Latency by Prefetching Java Objects

[13] Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors.

[14] Effective Jump-Pointer Prefetching for Linked Data Structures.

[15] Dynamic Software Prefetching for General-Purpose Programs