Prefetching for mobile computers using shape graphs

Kristian Kvilekval and Ambuj Singh
Department of Computer Science, University of California, Santa Barbara
{kris,ambuj}@cs.ucsb.edu

Abstract

We introduce a prefetching technique that is able to accurately prefetch complex pointer-based data structures. Compile time shape analysis is used to determine the set of possible future objects accessed by a program. We use the reachability graph generated during the shape analysis to create prefetching schedules for selected program points. The program is annotated to visit the reachability graph with the set of accessible objects at any program point, generating a superset of future object references. The technique is applicable to both static and dynamically created object structures. We demonstrate this technique in the context of mobile computing and show a speedup for a variety of Java benchmarks in the presence of disconnections.

1 Introduction

Mobile environments differ from traditional environments in many ways. Low bandwidth, limited power, and poor connectivity are some of the challenges faced by designers of mobile systems. Designing programs for the mobile environment with frequent disconnections is especially difficult. In order to overcome the lack of connectivity, several researchers have investigated prefetching of data [GKLS94, KS91, KP97, Sat96]

Our model consists of programs accessing a large object store over an unreliable communication link. This model is useful for current and future mobile platforms running applications such as CAD/CAM, OO databases, and GIS. In general, we expect the client to have limited memory and energy. Mobile platforms will not keep the entirety of the application database locally, but download the needed objects while running the application. Though certainly improving, mobile networks will not have continuous connectivity nor will said connectivity be of always the same quality. The mobile platform must take advantage of those times when connectivity is good in order to continue computing when connectivity is poor.

Our prefetching approach is based on the program code itself. While the stored object structures in a remote database may be complex, only those objects actually referred to by the program will ever be accessed. By accurately determining what objects will be accessed after a particular point in the program and prefetching those objects before a disconnection, we can alleviate or eliminate the effects of the disconnections.

In order to determine what future accesses a program will make, we use compile time shape analysis. Shape analysis produces a shape graph for a program point, representing the way the program traverses program data structures from that particular point. A shape graph is a directed graph with nodes representing runtime program values and edges representing program field references from those values. The shape graph is generated by symbolically executing the program code and adding edges for each access. Shape graphs represent the future accesses of the program and are necessarily imprecise as they contain all paths extended from the program point.

While the program is running, we prefetch objects from the remote store based on the current program point and the associated program's shape graph. In order to determine the set of likely objects, we examine all visible object references, usually the global object references, the object itself and any associated parameter references of the currently executing method. The prefetcher uses these references and the method's shape graph to determine the set of possible future references. As the cache size is limited, we limit the number of prefetched objects.

The main contribution of this paper is the introduction of a prefetching technique based on static program analysis that effectively removes both cold misses as well as other cache misses. Using this technique, we demonstrate speedups in the range of 6% to 171% in the presence of disconnections.

The remainder of the paper is organized as follows. In section 2 we discuss previous prefetching research for mobile platforms. Section 3 examines the construction and shape graphs in Java. This is followed by our technique for using reachability graphs to prefetch in the presence of disconnections in section 4. Our experimental results are presented in section 5. We conclude and present our future plans in section 6.

2 Previous work on prefetching

Past research in prefetching for systems has spanned from cachelines to file systems to object bases. Prefetching of memory is well known and has become necessary for speed improvements in the microprocessor domain [VL00, VL97]. Several authors [CWT$^+$01, ZS01] have investigated program slices for multithreaded processors. Speculative program slices extract cache-miss causing events and perform these in a separate thread slightly before original code. Our work could possibly be used to generate efficient slices for these architectures as shape graphs extract and represent these future events efficiently.

While not designed for the mobile domain, page or block level prefetching for file systems has been investigated and shown to reduce the disk latency in several important areas. Mowry et al. [MDK96] present a compiler based technique for page based prefetching of matrix codes. Their approach is applied only at loop constructs where I/O requests follow a regular pattern. Mitra et al. [MYC00] extract I/O related statements from a program in order to create a concurrent prefetching thread.

Most previous prefetching research has been concerned with reducing latency. However, prefetching for mobile platforms is mostly concerned with the ability to continue to do useful work while the user/program is in a disconnected state. This is sometimes referred to as *hoarding*. Under these conditions it becomes of primary importance to have the data available for further processing. Whole file prefetching has also been shown to produce performance improvements [GA94].

Several systems have attempted prefetching/hoarding of complete files for mobile disconnected systems. The CODA file system [KS91] provides "hoarding profiles" which augment the usual LRU replacement policy for caching files. The profile allows the user to manage the local file system cache by manually attaching priorities to certain files and directories. The cache manager combines the current priority of a cached object with its hoard priority and some function of its recent usage. Low priority objects are reclaimed when needed. The SEER [KP97] system provides an automatic predictive hoarding system based on inferring which files would be used together. In the SEER system, a system observer tracks file references (open and closes). File references patterns are monitored to create a *semantic distance* which is used to create clusters of related files. Both use past behavior in order to prioritize or create prefetching schedules for files. Our technique determines the future access patterns of object oriented systems and uses the program code itself to create the schedule.

Prefetching techniques for object oriented systems can be broken into three main categories: history, attribute, and code based. History based techniques monitor the user and/or programs access patterns and prefetch according to past behavior. Attribute based techniques allow the programmer to mark a class of needed runtime objects. Code based techniques are generally in the form of a prefetching runtime in conjunction with explicitly placed prefetch calls.

Approaches to data availability have included full replication, application specific partitioning, and object clustering based on past behavior. The full replication approach [PST$^+$97] has a high overall cost in terms of duplicated space. Thor [DLMM94, GKLS94] provides a distributed object-oriented database system where an Object Query Language could provide object navigation and allow for hoarding queries to be executed by the application. Phatak [PB99] considered *Hoard attributes* to be attributes that capture access patterns for some set of objects. In both cases, it was up to the user/programmer to determine the needed query and/or set of attributes. Another approach, *Prefetch support relations* [GK94], provides precomputed page answers in order to support prefetching from an objected oriented database. Rover [JTK97] uses application specific partitioning. The application designer must specify which objects are to be placed on the mobile node. In these systems, the user or designer must classify objects into prefetchable groups. Knafla provides an extensive review of prefetching for object oriented databases in [Kna99]. His work focused on analyzing the structure of object relationships and past work loads in order to predict page/object access patterns. Our work differs in that we are interested in analyzing programs instead of data relationships in order to predict access patterns. Krintz et al. [KCH99] show how to prefetch active portions of classfiles in order to lessen the startup delay of java programs. We do not deal with class files but focus on the more dynamic runtime object structures.

3 Overview of shape analysis

Shape analysis is a program analysis technique based on an abstract interpretation of a program to determine the possible runtime data structures. These data structures are termed "shape graphs". The shape graphs created by the analysis are a static representation of the dynamic runtime data structures manipulated by the program code.

Shape graphs have been used for many compile time optimizations including removing synchronization primitives [BH99, Ruf00], parallelization of codes [CAZ99], and type safety. Other uses include null analysis, pointer aliasing, cyclicity detection, reachability, and shaping [GH96, NMM98, WSR00].

Each shape graph is a static representation of the abstract state produced by a program fragment. The graph is made up of abstract locations as nodes. Each abstract location represents all the runtime values arrived at by the program. The graph edges represent program actions that access abstract locations: all edges are labeled with the field names of access operations. Our shape graphs represent the program actions leading from one abstract location to another.

```
class Connector{
  Part partA, PartB ; ... }
class Part {
  Connector left , right , up, down;
  Material material;
  Supplier    supplier;
  Cost        cost ;
  ...
  int volume (); }
weight = 0
while (part != null)
  weight += part.material.density
            * part.volume();
  connector = part.right;
  if (connector)
    part = connector.partB;
```

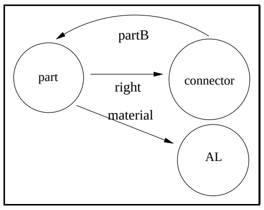


Figure 1: Fragment and shape: only items used in code (`part`,`material`) need to be prefeteched.

As an example consider the simplified program fragment, shown in Figure 1. It navigates an OO7 like database in order to weigh the elements. While the database may be large and have a very rich object structure, many programs may use only part of that structure. The above example code uses only the `material`,and `right` fields of each `part` in the database ignoring the `cost` and `supplier` among other fields. The access pattern is also revealed in the fact that the code fragment iterates through a list of `part` using the `right` field. A partial shape graph for this fragment is also shown in Figure 1. In the graph, the node (`part`) is used to represent the values of the variable `part` which access `connector` through the field `right`. The runtime value `part.material` is shown in the shape graph as `AL`. The cycle `part` $\overrightarrow{\texttt{right}}$ `connector` $\overrightarrow{\texttt{partB}}$ `part` contains the needed loop information from the original code. In order to be completed, the `volume` method would need to be analyzed and merged with the shown shape.

3.1 Building shape graphs

We construct the shape graphs of the program with a whole-program flow-insensitive, context-sensitive analysis. Our analysis works directly on Java byte codes, however this is not a necessity. We summarize the shapes manipulated by each method and propagate the shapes through the call graph.

A shape graph is a set of nodes AL and edges AE that represent the heap allocated structures of the program. Each $n \in AL$ represents the summary of runtime memory references, and is referred to as an *abstract location*. Each edge $e \in AE$ is labeled with a tuple (f : *count*) consisting of a field reference *f*, and a value *count* representing the first time the field is accessed, i.e. the earliest point in the program that we can expect to follow the field reference. This count allows the scheduling of the prefetches. We denote the abstract location corresponding to a program construct `x` as $AL(x)$ and the abstract location reached by following the field *field* from `x` by $AL(x)$.*field*.

3.1.1 Intraprocedural analysis

The basic shapes are built during the intraprocedural analysis. Initially the code is broken into basic blocks. Each basic block is then symbolically executed creating and extending the abstract locations. We examine only instructions involving references, all other instructions may be safely ignored as we are only interested in reads and writes to objects. Simple edges are created by field reference operations. Graphs are extended through field references and through *unification* of abstract locations.

We denote the unification of two abstract locations n_1 and n_2 with the operation unify(n_1, n_2). Unification is a recursive merge of edge sets based on edge names. The unification process places nodes of the graph into equivalence class sets. When two unified nodes have a common field, we choose the earlier accessed edge and continue unifying the abstract locations reached by the common edges. Unification recursively unifies all the access edges and their descendents. Table 1 summarizes the effects of unification corresponding to various program statements.

An abstract location may contain several (possibly) incompatible shapes. The analysis is flow-insensitive causing shapes generated by different program branches to be merged. In Figure 2, we demonstrate of the unification of two graphs with a single common edge; the new graph contains all the edges of the originals and is recursively unified.

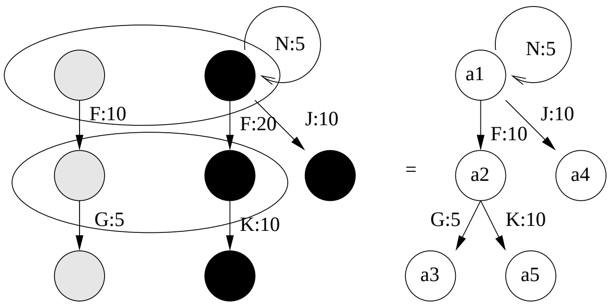


Figure 2: Unification of graphs: Common edges are recursively unified and contain the earliest expected access.

3.1.2 Interprocedural analysis

We construct a static call graph which is used to drive the analysis. The call graph is subsequently partitioned into strongly connected components (SCC). The call graph is topologically sorted and the method contexts (locals, globals, return value, and exceptions) for each method are propagated bottom-up through all possible call sites. The shape graphs are propagated from callee to caller during this phase. This allows the analysis to be context sensitive as the caller's shape information is not mixed into callee. We lose this sensitivity for methods belonging to the same SCC (mutually recursive meth-

Statement	Abstract Location	Description
`x = y`	unify$(AL(x), AL(y))$	Assignment
`x.`*field*`=y`, `y=x.`*field*	unify$(AL(x)$.*field*, $AL(y))$	Field assignment
`x = a[i], a[i] = x`	unify$(AL(x), AL(a)$.array)	Array assignment
`return x`	unify$(AL(x), AL(m)$.return)	Function return
`throw x`	unify$(AL(x), AL(m)$.exception)	Exception
`v = f(`$a_1, ..., a_n$`)`	$\forall_{t \in \mathtt{target}(f)}$unify$(AL(a_i), AL(t)$.formal$(i))$	Invocation
	$\forall_{t \in \mathtt{target}(f)}$unify$(AL(v), AL(t)$.return)	
`x = new T`	$AL(x) = \emptyset$	Allocation

Table 1: Statements causing unification of shape graphs and their effect. The fields *array*, *return*, and *exception* are special fields for array reference, method return, and method exception values respectively.

ods) as all methods will share a single shape context. In many cases the actual method receiver cannot be determined at compile time and this is a cause of uncertainty in the graph. Rapid type analysis [BS96] is applied to each call site in order to reduce the number of possible targets for each call site. For each target, the actual parameters are unified with a copy of callee method context in the caller's method context.

4 Prefetching using shape graphs

In this section, we introduce the use of shape analysis for prefetching support. Each shape graph represents the possible "points-to" relationship of runtime values of the program. A prefetching runtime system interprets the shape graphs on an actual runtime object structure generating a set of prefetchable objects.

From any point in the program, we can follow the generated shape to find a superset of the available objects. Given an actual runtime object and the program point's associated shape graph, we generate all actual objects that might be accessed before the next prefetch point. We generate shape graphs for all method entry points. Each shape graph represents how the method will manipulate structures referred to by its visible references (the object, arguments, globals) in the method body and its sub-method invocations.

Interestingly, while history based techniques would have little information of dynamically created structures, our technique uses all future program paths in order to generate prefetch schedules. This fact, however, comes at a price. Program flow (and access patterns) based on dynamic runtime values will be merged into single shape graphs. Currently we limit the number of object fetched during prefetch attempt to limit the impact of this imprecision, waiting for more precise shape graphs later in the call graph. In the future, we may use specialized shape graphs to represent different program flow choices.

4.1 Annotated programs

The programs are automatically annotated to use the compile time generated shape graphs. On entry to a method during an execution, the set of active objects (invocation target, arguments, and globals) used by the method are passed to the prefetching runtime system along with the method identifier. The runtime system is responsible for interpreting the shape graphs and generating the actual runtime references, which are scheduled for prefetching. The prefetchable objects are ordered by a breadth-first descent of both the shape graph and the real object graph sorted by the distance to the node (edges have weights denoting the expected first-time access).

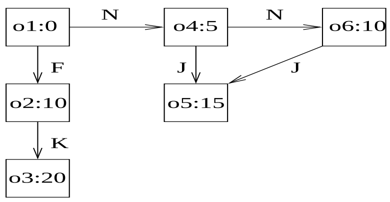


Figure 3: An object graph : Objects are linked through fields and have been labeled with expected access time using shape graph in Figure 2.

During a prefetch attempt, we walk the shape graph with a real object reference (o_n) and an initial abstract location (a_n). The computational cost of prefetching is the cost of interpreting the shape graph over the input object graph. The prefetcher ensures that the pair (o_n,a_n) is visited at most once in order to prevent infinite loops. For example, applying the shape graph in Figure 2 to the object graph in Figure 3 would produce the prefetch sequence $(o1, a_1) \rightarrow (o4, a_1) \rightarrow (o2, a_2) \rightarrow (o6, a_1) \rightarrow (o5, a_4) \rightarrow (o3, a_5)$. While the edges of the shape graph are matched with the edges of the object graph, the object prefetch order is sorted by the expected first-time access of the object.

5 Evaluation

5.1 Experimental model

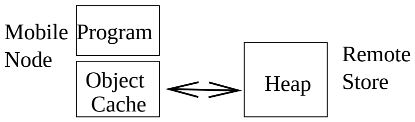


Figure 4: Program model: programs access the heap through a limited cache. The cache communicates with the heap over an unreliable link.

In order to test our implementation, we instrumented the programs shown in the first column of Table 2. The instrumented programs track all object level references and place them in a local cache. At specific program points, we allow the prefetcher to run, repopulating the local cache. In the model (Figure 4), the local cache represents the limited local storage of the mobile computing device while the main heap represents the larger remote store. All accesses are checked and tracked through the local cache. Those references that are not available from the local cache are obtained from the remote store and placed in the local cache before the program is allowed to proceed.

We simulate disconnections in our model by periodically disabling access to the main heap. The disconnection events are exponentially distributed (Poisson process) with each disconnection lasting a Gaussian distributed period of time. While disconnected, any object reference not found in the local cache must wait for reconnection.

In the experiments that follow, we varied several parameters: the frequency of prefetching invocation (*interval*) which controls the count of method invocations between prefetches, the number of prefetched objects from the object graph (*lookahead*), the mean cost of a disconnection in terms of accesses (*disconnect cost*), and the size of the local cache in terms of object handles (*cache size*). We assume that most programs reference memory at an approximately constant rate in order to measure time as a function of total number of accesses. This is reasonable as few programs spend the majority of their time computing solely in registers. Unless specifically mentioned, our standard interval was 2, the lookahead was 512 objects, the cache 2048 objects, and the mean failure cost 500 accesses. Our overhead is measured in terms of discarded unused objects. In the future, we will include communication overhead in addition to the cost of calculating the prefetching schedule.

We first examine the cold miss behavior of the applications. We expect that programs that exhibit clustered cold misses, with periods of intervening calm, will be good candidates for prefetching. We then test the accuracy of our approach by allowing the prefetcher infinite cache (all objects seen will be cached), and infinite lookahead (explore each shape graph fully). We then examine the frequency of cold misses and the frequency of prefetching. This analysis shows that though cold misses are clustered, frequent prefetching is still necessary. We then vary both the available cache space and the object lookahead of the prefetcher.

5.2 Benchmarks

For our analysis, we used several several SPECJVM 98 benchmarks and also included the `007` benchmark [CDN93]. Benchmarks `compress` and `mpegaudio` were omitted due to their small number of objects and integer nature. Our experiments needed access to the source code in order to simulate the disconnected database, we therefore also omitted `javac` and `jack`. The benchmarks were modified (1 additional line) to clear the local cache once the internal data structures had been constructed. This had the effect of constructing the database, flushing the cache, and restarting, allowing the application to access the heap in a way similar to an application restarting on an external database.

The prefetcher runs periodically while the program is running. In order to be effective, we expect the prefetcher to capture groups of objects that have not otherwise been accessed previously. In Figure 5, we plot the cold misses without prefetching vs. time (in program accesses). As seen from the graph, the clustering is both program and program phase dependent. In the first graph, we show `mtrt` which has a highly clustered set of cold misses. The next three graphs show `007` on different time scales. The graphs reveal a self repeating fractal structure; determining the expected clustering parameters warrants further study. However, there is a general clustering of cold misses and periodic prefetching should work well.

We then examined the case of using an infinite cache with infinite lookahead in order to determine the accuracy of the prefetching approach. Once an object has been seen, it will be cached for the lifetime of the program. We expect the prefetcher to reduce or eliminate the number of cold misses suffered by the program. In Table 2, we show the total number of allocated objects (Allocations), the objects in the cache after initialization (Initialized), the cold misses with an infinite cache/no lookahead (INF) and the cold misses with an infinite cache/infinite lookahead (PRE). The small number of prefetcher (PRE) misses are objects initialized by the JVM and therefore seen by neither the prefetcher (nor the reachability analysis) but accessed during the program. In order to get an idea of possible improvements in running time, we ran a simple experiment in which the prefetcher was allowed to run on every method invocation and the mean cost of disconnection was 500 accesses. As seen from Table 3, the reduction in cold misses realizes an immediate improvement in running time ranging from 5% to 63%.

Benchmark	Description	Allocations	Initialized	INF	PRE
`jess`	Expert system shell	26435	11196	1182	27
`db`	Small address database	542	528	51	1
`mtrt`	Raytracer	209630	179527	2498	3
`007`	OO database benchmark	266240	228535	107550	1

Table 2: Annotated benchmarks: Allocations: total allocations. Initialized: allocations during initialization. INF: Cold misses, no lookahead. PRE: Cold misses, infinite lookehead.

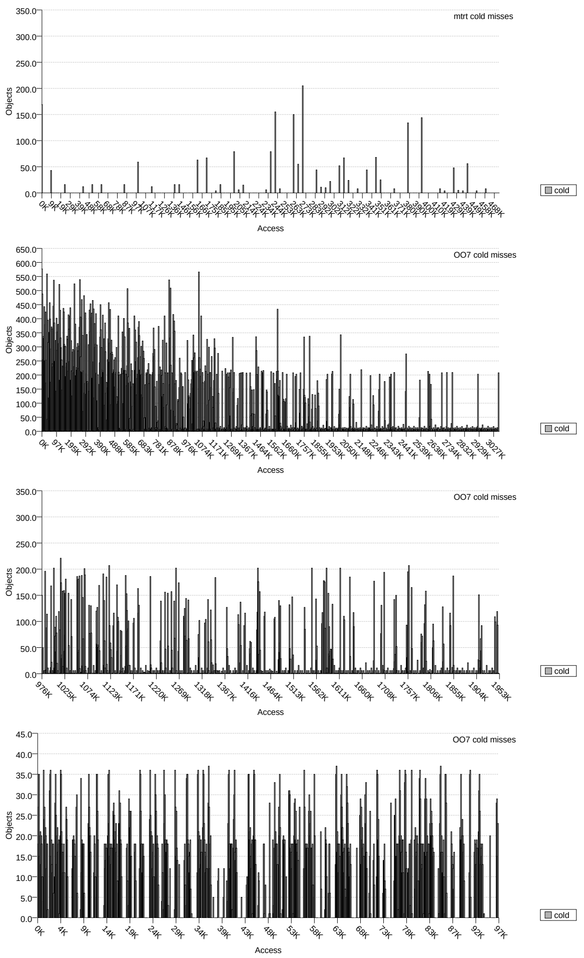


Figure 5: Coldmiss clustering : Coldmisses versus time.

The interval between prefetches affects the total number of objects available for prefetching. By varying the interval on which the prefetcher ran, we can lower the overhead of the prefetcher. However, the shape graphs will now not be used as effectively to prefetch. Table 4 shows the overall effect of increasing the interval between prefetch attempts. In the first column (∞), we show the cold misses with no prefetching. We find that at an invocation interval of 64, the prefetching accuracy has dropped significantly. In the future, we hope to limit prefetcher invocations to high yield points (such as points high in the call graph).

running time	`jess`	`db`	`mtrt`	`007`
INF	328K	11002	1803K	3133K
PRE	310K	3982	1222K	2547K

Table 3: Comparison of running time savings for no prefetching (INF) and prefetching (PRE).

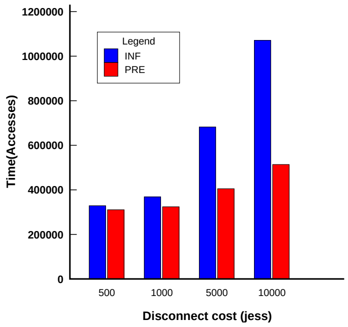
interval	∞	1	2	4	16	64
`jess`	1146	128	146	165	225	461
`mtrt`	2498	61	182	518	1278	2072

Table 4: Effect of increasing the prefetch interval on cold misses (lookahead=512).

In order to gauge the effect of longer disconnections, we varied the cost of a single cache miss. This is shown in Figure 6. The cost was measured in terms the running time of program plus the time program spent waiting for objects during the disconnects. We varied the disconnect cost from 500 to 10K accesses. In real communication systems and modern processors, failures could last much longer than 10K accesses; however, we decided to use a conservative estimate and simulate a somewhat faulty link. Under these conditions, we were able to achieve speedups of 52%. As the expected length of disconnection increases, each missed object has a greater effect on the overall running time.

5.3 Lookahead

The lookahead applies to how many future object references the prefetcher will try to gather. As the lookahead grows, so do inaccuracies due to uncertainties in the shape graph. In order to see how a changing lookahead would affect the quality of our prefetching, we examine the benchmark `jess` with several different lookahead values. Table 5 measures the disconnected misses (misses-dis), the overall cold misses (misses-cold), the total running time measured in accesses (runtime), the number of prefetched items discarded without use (discarded), and the average number of accesses between prefetch and use (wait). As expected, increasing the lookahead decreased the number of misses (both while disconnected and cold), while decreasing the total running time. Increasing the lookahead should allow the prefetcher to gather more objects earlier, reducing both the cold misses and the overall running time. However, lookahead follows the law of diminishing returns. Most



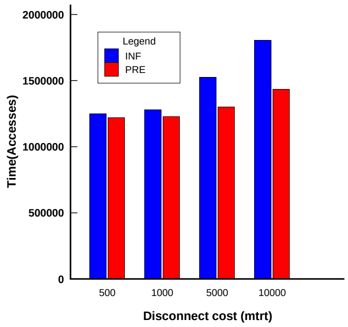


Figure 6: Effect of varying disconnect cost on running time.

lookahead	0	64	256	512	1024	2048
misses-dis	189	151	120	118	100	100
misses-cold	1182	151	128	129	131	131
runtime	401K	375K	358K	357K	346K	346K
discards	0	10K	6K	7K	8K	8.4K
wait	0	79	82	83	83	83

Table 5: Increasing lookahead beyond a certain point increases discard rates while not decreasing the total running time (example `jess`).

of the missing objects have been found with a lookahead of 512, and that after this point, the prefetcher collects many unused objects. We note that the prefetcher could rarely see more than 512 future objects. The wait time, also, leveled out somewhere between a lookahead of 512 and 1024 objects.

5.4 Prefetching and caching

So far we have examined prefetching by itself, without considering its impact on the reduction of the program's available memory. In this section, we examine the behavior of the prefetcher when integrated with a fixed local cache. Previous studies [CFKL01] have shown the need for an integrated prefetching/caching strategy.

We compare an infinite cache, with pure LRU and with prefetching/LRU. The prefetcher places objects into an LRU cache marked with the actual prefetch time. In this way, all prefetched objects will be newer than all currently stored objects, but will not keep newly accessed objects from entering the cache.

Aggressively prefetching can be detrimental as it may pollute the cache with future references. In Figure 7, we examine the effect of the limited cache on running time, by allowing the prefetcher to replace a proportion of the cached objects on each prefetch attempt. Under very limited cache and high lookahead (512), each prefetch may replace up to 50% of cache, we run 9% worse than standard LRU. However, at a lower ratio (1/8) of lookahead to cache size, a speed up of 171% is reached (for jess).

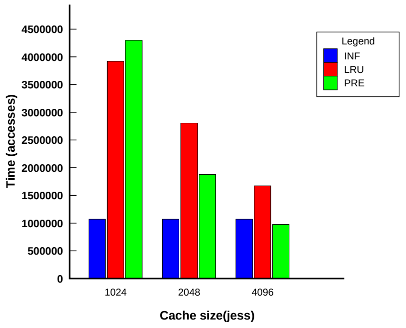
Benchmark	`jess`			`mtrt`		
Objects	26462			209630		
Cache	1024	2048	4096	10240	20480	40960
dis-cache	34386	10941	6707	24132	21073	30748
dis-pre	73350	5335	1243	48432	16496	13213

Table 6: Fixed cache and prefetch overhead in terms of discarded objects.

In order to measure the overhead in terms of unused objects, we set the cache size between five to twenty percent of the total objects used during program. Table 6 shows the effect of increasing cache size on the total cached and preloaded discards under these conditions. Cached discards (dis-cache) were objects forced from the cache after having at least one cache hit. Preloaded discards (dis-pre) were discards of an object before any hits and represent the overhead of the prefetcher. As expected, the prefetcher performed better with larger caches. In fact, increasing the cache size by 4 times for `jess` decreases the discarded prefetches by 59 times. Combining the prefetcher, which models future accesses, with cache management will be subject of further study.

6 Conclusions and future work

We have introduced a technique for accurate object prefetching based on the actual program code that manipulates the data structures. We use a compiler optimization technique to create static reachability graphs for use at runtime. In comparison to other prefetching techniques based on past history, the technique reduces cold misses the very first time the program is run and is quite accurate. It is also applicable to dynamically constructed data structures, and does not need to be trained with historical data. We have examined the technique on several benchmarks using simulated communication system with disconnections. Our initial results show speedup in the presence of disconnections between 6% and 171% percent depending on the expected length of disconnections.



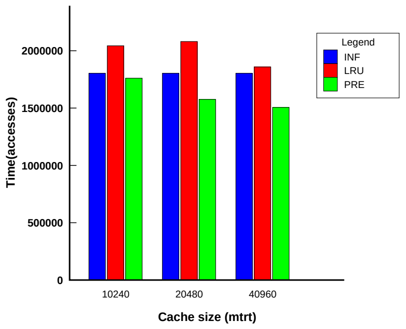


Figure 7: Effect of varying cache size on running time (lookahead=512).

Combining the shape graph technique with statistical models [CKV93, JG99, Kna99] based on past behavior may improve the accuracy of the prefetcher in a similar way that statistical branch prediction has been successful for instruction prediction. Our current technique fetches complete arrays without regard to access patterns. Improvements could be made by including more intelligent prefetching for arrays [MLG92]. The prefetcher currently runs at periodic intervals. While this is effective, the overhead of actual prefetcher can be high. We are examining techniques to limit the prefetching overhead both in terms of bandwidth and time, by choosing to prefetch only at selected high return points.

In this paper, we have focused on demonstrating our general technique for disconnected access. A more complete model of prefetching costs including communication and latency will be the subject of future work.

In a complete mobile object system, prefetching must be balanced with cost of copying vs. remote invocation and the cost of reconciliation of disparate versions. We are currently building such a system. Kan [Jam00] is a system for distributed programming that combines object-oriented programming with powerful feature such nested-transactions, asynchronous method calls, and automatically distributed objects. We are adding prefetching and reconciliation to Kan in order to support distributed programming for mobile systems.

References

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA*, pages 35–465, Denver, CO, Nov 1999. ACM.

[BS96] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA Object-Oriented Programming Systems, Languages, and Applications*, San Jose, California, Oct 1996. ACM.

[CAZ99] F. Corbera, Rafael Asenjo, and Emilio L. Zapata. New shape analysis techniques for automatic parallelization of C codes. In *International Conference on Supercomputing*, pages 220–227, Rhodes, Greece, Jun 1999. ACM.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, 1993.

[CFKL01] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *SIGMETRICS*, Ottawa Canada, May 2001. ACM.

[CKV93] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *SIGMOD Conference on Management of Data*, pages 257–266. ACM, May 1993.

[CWT$^+$01] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *International Symposium on Computer Architecture*, Jul 2001.

[DLMM94] M. Day, B. Liskov, U. Maheshwari, and A. C. Myers. References to remote mobile objects in thor. *ACM Letters on Programming Languages and Systems*, Mar 1994.

[GA94] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Summer USENIX Conference*, pages 8–12. USENIX, Jun 1994.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages POPL*, pages 1–15, St. Petersburg, Florida, Jan 1996. ACM.

[GK94] Carsten A. Gerlhof and Alfons Kemper. Prefetch support relations in object bases. In *6th Intl. Workshop on Persistent Object Systems (POS)*, pages 115–126, Tarascon, Provence, Sep 1994. Springer-Verlag.

[GKLS94] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected operation in the thor object-oriented database system. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Application*, Santa Cruz, CA, Dec 1994.

[Jam00] Jerry James. *Reliable Distributed Objects: Reasoning, Analysis, and Implementation*. PhD thesis, University of California, Santa Barbara, 2000.

[JG99] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers vol 48 (2)*, pages 121–133, Mar 1999.

[JTK97] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, Mar 1997.

[KCH99] Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing transfer delay using Java class file splitting and prefetching. In *OOPLSA*, Denver, CO, 1999.

[Kna99] Nils Knafla. *Prefetching Techniques for Client/Server, Object-Oriented Database Systems*. PhD thesis, University of Edinburgh, 1999.

[KP97] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St. Malo, France, Oct 1997. ACM.

[KS91] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 213–225, Pacific Grove, CA USA, Oct 1991. ACM.

[MDK96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *USENIX symposium on Operating systems design and implementation*, pages 3–17, Seattle, WA USA, Oct 1996. USENIX.

[MLG92] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*. ACM, Oct 1992.

[MYC00] Tulika Mitra, Chuan-Kai Yang, and TziCker Chiueh. Application specific file prefetching. In *IEEE Multimedia '00*, New York City, Jul 2000. IEEE.

[NMM98] Dor Nurit, Rodeh Michael, and Sagiv Mooly. Detecting memory errors via static pointer analysis. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 27–34, New York, NY, Jun 1998. ACM.

[PB99] Shirish H. Phatak and B. R. Badrinath. Data partitioning for disconnected client server databases. In *Workshop on Data Engineering for Wireless and Mobile Access (MOBIDE)*, pages 102–109, Seattle, WA, Aug 1999. ACM.

[PST+97] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *16th ACM Symposium on Operating Systems Principles*, pages 288–301, Saint Malo, France, Oct 1997. ACM.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI2000),*, Vancouver, British Columbia,, Jun 2000. ACM.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, May 1996. ACM.

[VL97] Steven P. VanderWiel and David J. Lilja. When caches aren't enough: Data prefetching techniques. *Computer (Vol. 30, No. 7)*, pages 23–30, Jul 1997.

[VL00] S.P. VanderWiel and D.J. Lilja. Data prefetch mechanisms. In *ACM Computing Surveys*. ACM, Jan 2000.

[WSR00] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proc. of CC 2000: 9th Int. Conf. on Compiler Construction*, Berlin, Germany, Mar 2000. Springer-Verlag.

[ZS01] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In

Intl. Symposium on Computer Architecture (ISCA 2001), Göteborg, Sweden, Jul 2001. ACM.