SPREE: Object Prefetching for Mobile Computers

Kristian Kvilekval and Ambuj Singh

Department of Computer Science, University of California, Santa Barbara
{kris,ambuj}@cs.ucsb.edu *

Abstract. Mobile platforms combined with large databases promise new opportunities for mobile applications. However, mobile computing devices may experience frequent communication loss while in the field. In order to support database applications, mobile platforms are required to cache portions of the available data which can speed access over slow communication channels and mitigate communication disruptions. We present a new prefetching technique for databases in mobile environments based on program analysis. SPREE generates maps of a client program's use of structured data to be used by our prefetching runtime system. We apply SPREE in the context of mobile programming for object structured databases demonstrating an effective way to prefetch/hoard over unreliable networks with speedups up to 80% over other techniques.

1 Introduction

Mobile platforms are everywhere. As these devices permeate our lives, we expect more integration and combined utility with traditional computing. However, mobile environments differ greatly from traditional environments in many ways. Low bandwidth, limited power, and poor connectivity are some of the challenges faced by designers of mobile systems. Emerging mobile database applications, such as mobile Geographical Information Systems (GIS), are especially difficult due to their large data requirements and expectation of consistent communication. In particular, scheduling limited resources for demanding applications over inconsistent communication links is especially difficult.

Resource scheduling is one of the fundamental problems faced by designers of any system, but it especially critical on smaller platforms. Knowledge of the future is the key to efficient scheduling of resources. In this paper, we focus on the prefetching or *hoarding* of data over unreliable links. Knowledge of future events can come from diverse sources, but has been traditionally in the form of programmer annotations or analysis of past events. Programmer annotations can be difficult to construct and may be error prone. Past behavior is not always available nor is it always a good indicator of future behavior. However, the knowledge of future access patterns is present in the client programs that use the database

* This work was funded in part by the following grants: NSF ANI01-23985, NSF IIS98-17432, and NSF EIA00-80134

system. In recent years, there has been an increasing interest in object database languages with the acceptance of object-oriented and object-relational database systems. These systems already reduce the “impedance-mismatch” between the program code and data storage often experienced in traditional SQL environments. Emerging complex data languages such as JDO [28], OQL [23] provide new opportunities to explore the benifits that program analysis can bring to traditional databases.

Our model consists of clients accessing structured data over unreliable links. Our prefetching approach (SPREE: Shape-assisted PREfetching Engine) 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 by a program and prefetching only 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 after the program 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. At runtime, the server generates the set of prefetchable objects periodically based on client parameters. In order to determine the set of likely objects for a particular program point, the server is signalled with the client’s program point and syntactically visible object references. The prefetcher uses these references and the method’s shape graph to determine the set of possible future references.

The contribution of this paper include the introduction and analysis of a new prefetching technique for mobile computers based on shape analysis. This technique uniquely removes both *cold* misses as well as other cache misses as compared to other techniques. Cold misses are especially critical when the overall cache miss rate is already low or when communication blackouts are common. Our technique is applicable to mobile database applications that are trying to operate over faulty channels or where hoarding is critical to the application. We demonstrate the technique with extensive simulation over faulty communication channels for a variety of benchmarks. We show SPREE is useful for hoarding with speedups between 9% and 80% over an infinite cache, and also demonstrate the system with intermittent connectivity where we show speedups between 2% and 80% over a recursive object prefetcher. We also examine several techniques to reduce the overhead of prefetching both on the client side and on the server reducing both the computational load and bandwidth utilization.

2 Previous Work on Prefetching

Past research in prefetching has spanned from memory cachelines to file systems and web-pages to object databases. 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 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.

Several systems have attempted prefetching/hoarding of complete files for mobile disconnected systems. The CODA file system [18] 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 [20] 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-based, attribute-based, and code-based. History-based techniques monitor the user and/or program's 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 for databases have included full replication, application specific partitioning, and object clustering based on past behavior. The full replication approach [24] has a high overall cost in terms of duplicated space. Thor [10, 13] 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 [25] 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* [11], provides precomputed page answers in order to support prefetching from an object-oriented database. Rover [16] 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 [19]. His work focused on analyzing the structure of object relationships and past workloads in order to predict page access patterns. SEOF [1] is a page level predictor that uses a two-level queue to filter object scans from the prefetcher. The first level queue tracks objects that would cause a page miss. Once the queue threshold is reached any missing object from the page will cause the entire page to be prefetched. Our work differs in that our technique is an object level predictor (not dependent on the quality of object clustering) and does not require historical data to determine object relationships but deduces them automatically from the code.

Cahoon and McKinly [5] examine dataflow analysis for prefetching object structures for Java. This works resembles our work, but is targeted to speeding up scientific applications and uses a different technique. Our method can be seen encompassing theirs by limiting lookahead to at most one reference and is focused on supporting disconnections instead of redcuing memory latency.

3 Program Analysis for Prefetching

Shape analysis [14] is a program analysis technique based on an abstract interpretation of a program to determine the "shape" of possible runtime data structures. Shape analysis produces a set of *shape graphs* 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 symbolic abstract 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 have previously been used to determine static properties of programs and for many compile time optimizations including removing synchronization primitives [3, 26], parallelization of codes [8], and type safety. Other uses include null analysis, pointer aliasing, cyclicity detection, reachability, and typing [12, 22, 29].

```
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;
```

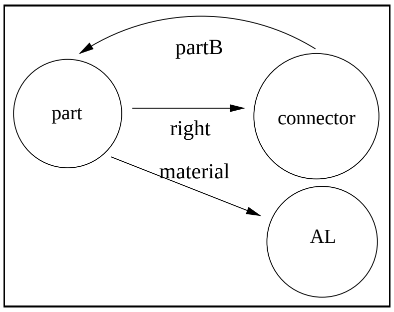

Fig. 1. Code and shape: only items used in code (`part`,`material`) are in the shape

In order to understand what a shape graph is, we present a typical program fragment of integrated databases in Figure 1. The shape graph shown on the right of the figure represents the code lines on the left. It navigates the 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 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. 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` through the field `right` to `connector` through `partB` and back to `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 Analysis and Graph Construction

Our analysis is summarized as follows: We extend the well-known shape analysis with the addition of a single parameter representing the earliest time the program could follow a pointer. This value will be used by the runtime system for scheduling. In more detail, the abstract state of the program can be represented by a mapping of program variables M to abstract runtime locations and an abstract heap H representing the interconnections between the abstract runtime locations.

A program variable v can point to a number of heap location at runtime. During shape analysis, we work with a set of abstract locations. The set of abstract locations R_v will contain those locations determined by the analysis to be reachable through object manipulations.

The abstract state M is a mapping of program variables to abstract runtime locations $M \in \mathcal{P}(Var*, R_v)$ where $Var*$ is the set of program variables. As we are interested only in object interrelationships, M will contain only program variable names that are pointer variables. Let H represent the state of the runtime heap. The heap will capture the object interrelationships and allow a simple graph representation. The abstract heap, H, is a set of tuples of the form (R_m, R_n, f, rw, l) where both R_m and R_n are sets of abstract locations, f is some field name from the program (i.e., the expression $X.f$ occurs in the program). rw has the value of either r or w, depending on whether the edge is the result of a field read or write, and $l \in N$ is earliest expected access of field f in terms of program intruction count. For example, in Fig. 2, M contains the pair (a, R_a), and the heap H contains the tuple $(R_a, R'_a, f, r, 10)$. A shape graph is a set of program variable mappings (M), combined with the abstract heap H in a tuple (M, H).

Shape graphs are created and extended by simulating the actions of the program through abstract interpretation, which creates and connects heap tuples. Simple program actions, such as a field access instruction, create heap tuples. When two variable are determined to point to the same abstract location, we unify their heap representations. Unification is a recursive operation that begins with unifying the abstract locations and continues by unifying the compatible heap tuples that stem from the originally unified location. Given two abstract locations, R_a and R_b that are to be unified, we first unify their abstract locations and then recursively unify their compatible tuples in the heap.

We say that two heap tuples may *unify* when they belong to the same shape graph and they have a common field access. In other words, two tuples (R_a, R'_a, f, rw_1, l) and (R_b, R'_b, g, rw_2, m) are *unifiable* if $f = g$. Note in Fig. 2 that R_a and R_b are unifiable as they both have a common field F.

We define a unification operation ($\sqcup$) over the tuples of H as follows: Given tuples $t_a = (R_a, R'_a, f, rw_a, l)$ and $t_b = (R_b, R'_b, f, rw_b, m)$, we have $t_a \sqcup t_b = \{(R_a \cup R_b, R'_a \cup R'_b, f, rw_a \sqcup rw_b, \min(l, m))|\} \cup \{t_{a'} \sqcup t_{b'}\}$. Note that $r \sqcup r = r, w \sqcup w = w, r \sqcup w = r$. Also note that the operation will recursively find and unify compatible tuples of unified abstract locations. Finally, let T_a be the set of tuples that emminate from the abstract location R_a. The unification operation $\sqcup$ is defined over sets T_a and T_b as simply the union of unification of their constituent tuples.

We define the unification of program variables in terms of the unification of their respective abstract locations:

$$\begin{aligned}
a \sqcup b &\equiv R_a \sqcup R_b \\
&= (R_a \sqcup_M R_b, R_a \sqcup_H R_b) \\
R_a \sqcup_M R_b &= \begin{array}{l} M \setminus \{(a, R_a), (b, R_b)\} \\ \quad \cup \{(a, R_a \cup R_b), (b, R_a \cup R_b)\} \end{array} \\
R_a \sqcup_H R_b &= H \setminus \{T_a \cup T_b\} \cup \{T_a \sqcup T_b\}
\end{aligned}$$

A graphical example of the unification process is shown in Fig. 2. The process begins at the root of two different shape graphs. First the abstract locations R_a and R_b are unified, then the common edges of the graphs are unified. The expected access time is the minimum of unified edges. The R/W field has not been shown. The resulting unification leaves a new shape graph in heap.

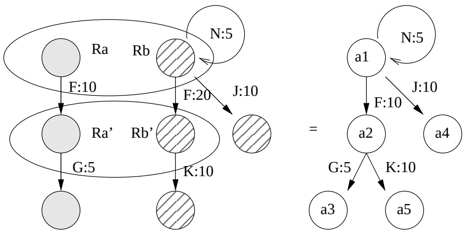

Fig. 2. Unification of graphs: Common edges are recursively unified and contain the earliest expected access

Our dataflow analysis ranges over all labelled program statements S. Given the control flow graph of the program $flow(S)$, we define the analysis based on a set of transfer functions f_l^{SA} mapping a set of shape graphs to another set of shape graphs. ι is the initial shape graph (an empty heap).

$$SG_{in}(l) = \begin{cases} \iota & \text{when } s = \text{init}(S) \\ \bigcup \{SG_{out}(l') | (l, l') \in \mathit{flow}(S)\} \end{cases}$$
$$SG_{out}(l) = f_l^{SA}(SG_{in}(l))$$

We define the set of transfer functions f^{SA} below. We are interested only in field operations and variable assignments of pointer variables.

1. $[a = b]^l$ Assignment: Unify the two variables.
$$SG = a \sqcup b$$
2. $[a = x.f]^l$ Program variable a takes the value of a field read: create a new heap tuple and unify the variable with the field.
$$SG(H) = SG(H) \cup (R_x, R'_f, f, r, l)$$
$$SG = a \sqcup x.f$$
3. $[x.f = a]^l$ A field write takes the value of program variable a: create a new heap tuple and unify the variable with the field.
$$SG(H) = SG(H) \cup (R_x, R_a, f, w, m)$$
$$SG = a \sqcup x.f$$
4. $[x = \text{new } T]^l$ Allocation site: create a new abstract location in the heap.
$$SG(M) = SG(M) \cup (x, Rx)$$

The static call-graph is used to drive the interprocedural analysis. The call-graph is partitioned into strongly connected components (SCC), then topologically sorted so that leaf methods are analyzed first. The method contexts (locals, globals, return value, and exceptions) for each method are propagated bottom-up through all possible call sites. Note that all future objects will either be linked from syntactically visible variable or a global. The shape graphs are propagated from callee to caller during this phase through the unification of shape graphs. Method call-sites force shape-graph unification of the caller's actual parameters with copies of the callees formal parameters. 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 methods) as all methods will share a single shape context. We also "boost" global variables into the callers callframe so that we may prefetch structures referenced by global variables in later callframes as early as possible.

Object-oriented languages usually permit dynamic dispatch at runtime. This implies that in many cases the actual method receiver cannot be determined at compile time. We further unify all possible target method graphs into the caller's graph causing more uncertainty in the graph. Rapid Type Analysis [2] is applied to each call site in order to reduce the number of possible targets for each call site. This method has been shown to greatly reduce the expected number of target methods with little cost.

3.2 Prefetching at Runtime

In this section we present our prefetching algorithm, which is based on the shape analysis of the previous section. In order to prefetch at runtime, the algorithm uses both an actual runtime value combined with the shape graph for the associated program point. Prefetching can be done either at the client side or the server side. We have chosen to investigate server side prefetching, though the techniques are also immediately applicable to the client side.

From any point in the program, we can follow the associated shape graph to generate a set of the possibly accessed objects in the database. 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. Currently, we generate shape graphs for all method entry points. Each shape graph represents how the method will manipulate structures referred to in the future by its visible references (the object, arguments, globals) in the method body and its sub-method invocations.

The client programs are automatically instrumented to signal the prefetching runtime at appropriate points. We permit prefetching to occur only at method entry, however this choice was arbitrary. Finding more precise prefetch points could be an interesting avenue of future research. Having an arbitrary number of prefetch points must be balanced against the cost (size) to maintain shape graphs for those points. This will be further discussed in section 4.1.

```
/* Input:  An initial object o
           A shape graph root ) */
Queue ServerPrefetch(Object o, SGNode sgroot)
 PriorityQue search
 Queue  fetch
 Set    seen
 int    distance = 0
 SGNode sgnode = sgroot

 search.push ( [o, sgnode, distance] )
 while not empty search
   [o, sgnode, distance] =  search.popmin ()
   if not seen.contains ( (o, sgnode) )
     seen.add ( [o, sgnode] )
     for each edge e of sgnode
       next = access_field(o, e.field);
       search . push ( [ next, e.target, distance+e.dist ] )
       fetch . push ( next )
 return fetch
```

Fig. 3. Prefetching algorithm for iterating over shape graph and object graph

At runtime, the client is responsible for sending initial object references and a program location (usually the name of the currently executing method) to the remote store to allow prefetching. The frequency that the client requests are sent over the link and the time spent processing prefetched objects represent the entire overhead of the client.

The server is responsible for interpreting the shape graphs of the program based on the values sent by the client. Upon receiving a prefetch request, the server will walk the shape graph with a real object reference (o) and an initial abstract location (rv) representing the root of the shape graph. Our pseudocode for a single object reference is represented in Fig. 3. The algorithm traverses the object graph based on program's field accesses represented by the shape graph. We search through the object graph in a breadth-first manner based on earliest expected access through the field. The computational cost of prefetching equals the cost of interpreting the shape graph over the input object graph. The prefetcher ensures that the pair (o,rv) is visited at most once in order to prevent infinite loops. Array objects are broken into the objects contained, each object being given an equal probability of access. Combining our object prefetcher with array prefetching techniques may prove valuable.

4 Experimental Setup

In this section we discuss our analysis and simulation model. Our analysis and simulation are written in Java. Shape analysis is particularly effective when used with the rich type structures often employed in object oriented programs. The analysis is a whole-program analysis that runs completely at compile time. Our analysis interprets the Java code at the byte-code level and can work for libraries even without program source code. We use a conservative model of dynamic class loading by loading all available classes in order to perform the analysis even when Java reflection is used. A simple class-pruning technique by Bacon [2] is used to limit the examined classes. The prefetcher does, however, assume that the class libraries at runtime are the same that were used during the analysis. This restriction could be removed by always performing the analysis at load time. Bogda [4] has investigated providing incremental shape analysis for Java. We would expect in a real system with an optimizing JVM would perform the analysis during loading where the shape analysis could also be used for other compiler optimizations. The result of the analysis is a set of shape graphs for the static prefetch points. The runtime system may not necessarily use all the graphs available, but may choose which points are the most profitable.

Fig. 4 shows our model of computation. The local cache represents the limited local storage of the mobile computing device which accesses data from a larger remote store. All program accesses are checked and allowed only through the local cache. Those references that are not available from the local cache are obtained from the remote store through a simulated communication link and placed in the local cache before the program is allowed to proceed. Communication is modeled as a single communication line between the program and

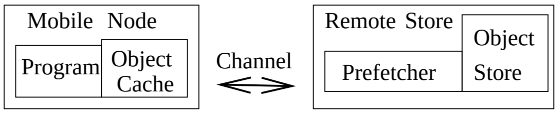

Fig. 4. Program model: programs access memory through a limited cache. The cache communicates with the heap over an unreliable link

the object repository with a fixed latency and bandwidth per experiment. At specific program points, the client signals the prefetcher, which repopulates the local cache. Both accessed objects and prefetched objects use the same channel. We simulate disconnections by periodically interrupting access to the repository. While disconnected, any object reference not found in the local cache must wait for reconnection. The communication time for shape graphs is not included in our simulation, as we expect that the shape graphs are likely to to be stored in the remote store, or communicated only once per program connection time.

For the simulation, each benchmark was instrumented to call the runtime system at all object reference instructions. This allows the simulator to capture all memory references described above. The client code was also instrumented to initiate prefetching on the server at method entry. Disconnection delays are simulated with disconnection events. The disconnection events are exponentially distributed (Poisson process), with each disconnection lasting a Gaussian distributed period of time. An object reference not found in the local store is forced to wait the entire remaining disconnection period. Communication costs were modeled by adjusting to the total runtime of the program by the time spent waiting for objects.

4.1 Benchmarks

We instrumented the programs shown in the first column of Tab. 1. We used the object-oriented benchmark `OO7` [7] that had been recoded as Java for our study. Though not originally database applications, we adapted several SPECJVM98 to provide wide variety of programming styles: `jess` is an expert system shell, `db` is a small database, and `mtrt` is a multi-threaded raytracer. Other SPECJVM98 Benchmarks `compress` and `mpegaudio` were omitted due to their integer nature and small number of objects. Our experiments needed access to the source code in order to adapt the code to use a simulated database, we therefore also omitted `javac` and `jack`. The benchmarks were modified (one additional source line) to clear the local cache once the internal data structures had been constructed. This allowed our simulation to initialize the external data storage, clear the local cache and restart as if it was operating on an external data source.

Tab. 1 shows the total number of classes examined, the number of classes instrumented, the methods instrumented, the number of graphs, and the size of the resulting Java serialized graph structures. The analysis time was considered negligible as all benchmarks were analyzed is less than 2 seconds on a Pentium

Table 1. Analysis : classes examined and instrumented, methods, # shape graphs, size of graphs, Runtime: objects initialized and used

Analysis						Runtime			
Name	total	classes	methods	graphs	size	Allocations	Initialized	INF	SPREE
`007`	650	35	73	173	88K	266240	228535	107550	1
`jess`	1921	151	623	1910	1308K	26435	11196	1182	27
`db`	1771	3	28	119	41K	542	528	51	1
`mtrt`	1823	57	157	553	206K	209630	179527	2498	3

466 Mhz. Note that our analysis has been optimized for neither size nor speed, but these figures show that both costs are quite small.

4.2 Simulation Parameters

We measured time in our simulation as a function of the total number of the memory accesses that the program had made. This is reasonable, as few programs spend the majority of their time computing solely in registers [27]. We tested our use of program-accesses-as-time by accurately measuring the time between groups of accesses for non-prefetching programs. Tab. 2 shows that accesses do actually follow a regular, constant pattern. Both `007` and `mtrt` had no console output and were very stable. Fig 5 shows the the interval time for blocks of 5000 accesses across the entire program run for `007`. Though not exactly constant, it is quite stable meaning that time can be measured by simply counting the number of accesses for this benchmark. The time vs. accesses graphs `jess`, `db` and `mtrt` were also similar.

We used an average object size of 64 bytes, over a 10 Mbit communication link with <1ms latency. The client cache was limited between 10% and 100% of the working set and the server lookahead was limited between 16 and 1024 objects.

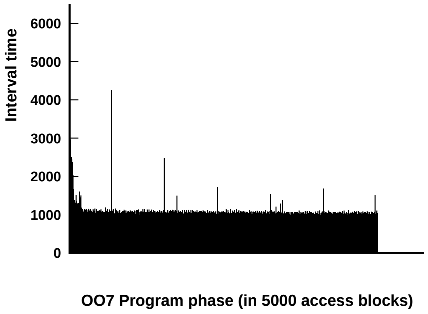

Fig. 5. Time for groups of 5000 accesses: a stable value implies that accesses can be used to measure time

Table 2. Measuring time using memory accesses

Name	Accesses	sample	min	max	mean	med	dev
`007*`	2547836	5000	981	8069	1109	1058	382
`jess`	304901	500	66	1060	164	134	96
`db`	3982	50	12	2116	79	26	295
`mtrt*`	1212727	5000	1483	7459	1931	1767	565

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. Fig. 6 shows two histograms of cold misses without prefetching vs. time. As seen in the graphs, the clustering is both program and program phase dependent. The histograms show `OO7` on different time scales. The graphs reveal a self-repeating fractal structure; determining the expected clustering parameters warrants further study. However, there is a phase-dependent clustering of cold misses, and periodically prefetching should work well if we prefetch during periods of calm.

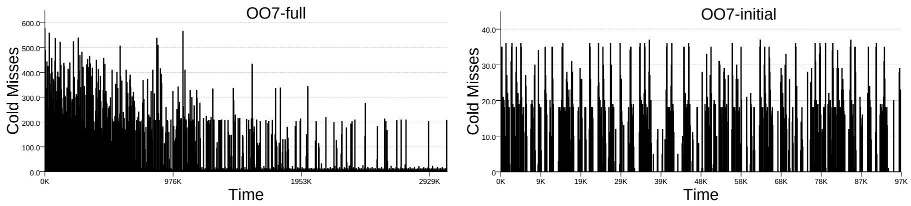

Fig. 6. Cold misses vs time: clustered cold misses provide prefetching opportunities

We examined the performance of SPREE under varying conditions in order to test its effectiveness and compare it to an infinite cache, a standard LRU cache, and a simple recursive prefetcher. In our experiments, we varied the following parameters: disconnection frequency (*mtbf*), duration of disconnection (*mean*), the size of the local cache (*cache size*) with respect to the entire repository. Several parameters were used to control the frequency vs. accuracy of the prefetching: The first group of parameters controlled how often the prefetcher was signalled by the client reporting its progress. Clients were responsible for keeping the repository up-to-date in regard to their progress and currently active objects which drive the prediction process. In this group, the frequency of prefetching invocation (*interval*) controls the number of method invocations between prefetch requests while *callheight* controlled the maximum level of the callgraph that prefetching would occur. On the server side, at each prefetch point, the maximum number of prefetched objects collected from the object graph was controlled through the parameter *lookahead*. Unless specifically mentioned, our standard interval was 1, the lookahead was 64 objects, the cache 10 % objects of initialized objects, and the mean communication failure cost 5000 accesses (or 1 ms over 10 Mbit links).

5 Experimental Results

We first examine our prefetcher without regard to caching issues to better understand the parameters affecting the pretching accuracy and overhead. We then examine the prefetching under memory contraints and compare it to the recursive object prefetcher.

In order to determine the accuracy of the SPREE, we examined an infinite cache with infinite lookahead. Once an object has been seen, it will be cached for the lifetime of the program. As mobile devices become more powerful and/or for small programs, this is a realistic model. We expect the prefetcher to reduce or eliminate the number of cold misses suffered by the program.

In Tab. 1, we show the total number of allocated objects created during the entire run (Allocations), the objects created during initialization phase (Initialized), which approximates the size of the external store, the cold misses with an infinite cache/no lookahead(INF) and the cold misses with an infinite cache/infinite lookahead(SPREE). The small number of prefetcher(SPREE) misses are objects initialized by the JVM and therefore seen by neither the prefetcher nor the reachability analysis but accessed during the program.

Disconnection time will adversly affect execution time. In order to gauge the effect of longer disconnections, we varied the mean duration of a each disconnection in Fig. 7. The graph shows relative running of SPREE compared to the normalized running time of INF. The cost was measured in terms of the running time of program plus the time program spent waiting for objects not available in the cache during the disconnects. We varied the mean disconnect period from 500 to 10K accesses (approximately 100 μsec to 2 millsec over a 10 Mbit link) . 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 increasing speedups. As the expected length of disconnection increases, each missed object has a greater effect on the overall running time.

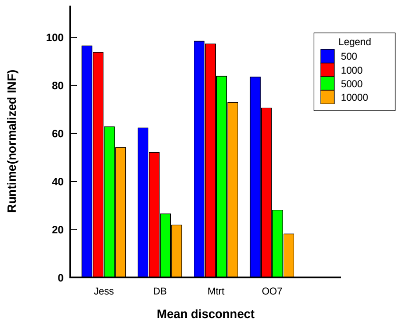

Fig. 7. Varying mean disconnect period on running time

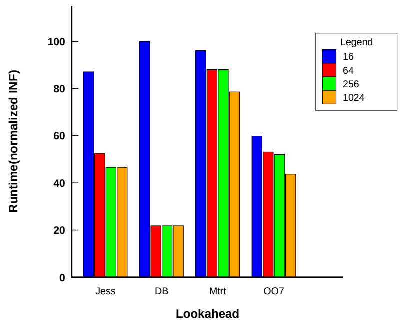

Fig. 8. Varying lookahead on running time

The server side parameter, *lookahead*, specifies how many future objects the prefetcher will try to gather and is directly related to the server overhead. As the lookahead grows, so do inaccuracies due to uncertainties in the shape graph.

By limiting lookahead, we should be able to both reduce inaccuracies and reduce server overhead. However, limiting lookahead also affects how far into the future we are exploring. If the expected disconnection period is long, then the program might benefit from greater lookahead.

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. Fig. 8 shows that increasing lookahead between 16 and 1024 objects can reduce overall running time. 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 of the missing objects have been found with a lookahead of 64, and that after this point, the prefetcher collects many unused objects. We note that the prefetcher could rarely see more than 64 future objects for all benchmarks.

5.1 Prefetching with Constrained Memory

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 [6] have shown the need for an integrated prefetching/caching strategy.

We compare an infinite cache (INF) with pure LRU and with prefetching/LRU. The client 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.

With small lookahead we saw increasingly good performance with increasing cache as seen in Fig. 9. However, aggressively prefetching can be detrimental as it may pollute the cache with future references. Fig. 10 examines 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 (10%) and high lookahead (1024), cache pollution resulted in worse performance than standard LRU in several cases (db, mtrt).

We examined a heuristic technique that suggests ignoring prefetches near the bottom the static call-graph might improve the overhead. In Fig. 11, we ran each benchmark with a cache limited to approximately 10% of the initialized objects. Limiting prefetch requests to the upper part of the static call-graph did not adversely affect the efficiency of the prefetcher, while lowering significantly the overhead (see Tab. 3). The number of calls made to the prefetcher drops signifigantly after just pruning the leaf procedures.

Another possibility to reduce the overhead of the prefetcher was to disable prefetching where the results of the prefetching were data dependent. For example, prefetching a single call to a retrieve function on a hash table would return the entire hash table as all elements are equally likely. Currently these program points are culled by limiting the lookahead.

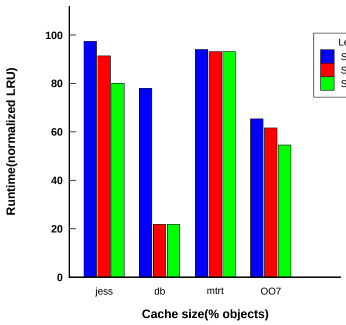

Fig. 9. Varying cache size on running time

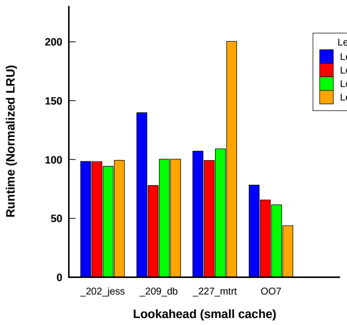

Fig. 10. Varying lookahead on running time

In our last experiment we compared our prefetcher with a simple recursive prefetcher that prefetches the next level of outgoing object links. This prefetcher design uses the naive strategy that if an object is accessed, then it is likely that one of its neighboring objects will be accessed next. Though simple, this technique has been applied to context specific web-prefetching i.e. sending all embedded image links when a specific web page is requested.

Table 4. SPREE(S) vs. REC(R) prefetcher with disconnections. Time overall and bandwidth utilization

	overall	idle	prefetch	transfer	avg
`007`-S	214344116	2049	3620	4344689	488.0
`007`-R	219476192	3470	540640	3935439	485.5
`jess`-S	770056K	231626	64345	28018	982.1
`jess`-R	845066K	290432	14343	20878	2890.5
`db`-S	561109	3722	230	354	9442.8
`db`-R	686062	3542	396	510	7578.6
`mtrt`-S	31135K	1135043	67030	27580	3162.7
`mtrt`-R	56079K	1152618	44000	37630	6721.4

We compared our prefetcher to a recursive prefetcher in Tab. 4 with disconnections lasting approximately 1 second (5M accesses). While the number of disconnections experienced by the applications was small(<20), the effect on overall runtime was great. SPREE was able to provide speedups between 2% and 80%. Bandwidth utilization for `007`, `mtrt` and `db` were comparable for both techniques, while `jess` used considerably more under SPREE. Upon examination, the majority of prefetchable objects for `jess` were not used due to its dynamic nature(rule engine). Decreasing bandwidth utilization for very dynamic programs will be the subject of future research. The `007` benchmark was I/O bound, most prefetch were converted to standard transfers as they were needed almost immediately.

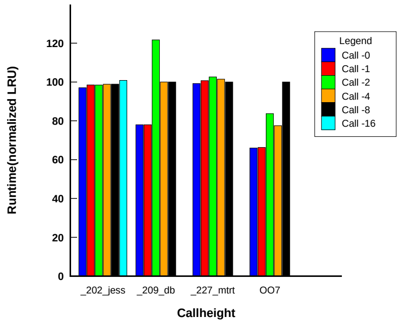

Fig. 11. Call-graph limiting on prefetches

Table 3. Increasing the callheight on cold misses (cache=10%)

call	INF	0	1	2	4	8
`jess`	1182	229	242	244	244	253
calls	NA	126K	56K	44K	24K	18K
`mtrt`	2498	1293	1359	2326	2451	2470
calls	NA	347K	36K	4K	1379	1
`OO7`	105K	17K	17K	30K	16K	NA
calls	NA	1M	364K	129K	43K	NA

In summary, our experiments demonstrate SPREE to be an effective and accurate prefetcher under a variety of conditions. When operating under few memory constraints, we are able to prefetch far into the future accesses of a program effectively hoarding data for future use when disconnected. When operating over noisy or unreliable links, SPREE smoothed out these short disconnections ($<$ 1 ms) providing significant speedups over non-prefetching systems depending on the rate and duration of disconnections.

We also examined several parameters to control overhead both on the client and on the server of the prefetcher. While no single parameter was best for both memory constrained/limitless devices, controlling the prefetcher interval was effective for limited memory devices while pruning by call-height was more effective for limitless memory. On the server side, we determined that lookahead between 16 and 64 was best over our set of benchmarks. We compared SPREE to a recursive prefetcher that also required no prior training. When operating with disconnections, SPREE was able to improve between 2% and 80% over a recursive prefetcher.

6 Future Work and Conclusions

We have introduced SPREE(Shape-assisted PREfetching Engine), a new method of accurate prefetching and/or hoarding for object programs operating on databases. Data-flow analysis is used to create static reachability graphs for use at runtime. Compared to other prefetching techniques, SPREE reduces cold misses the very first time the program is run and is accurate. While other prefetchers often must be trained in order to perform well, our technique also is applicable to dynamically constructed data structures, for which training is impossible.

Combining the shape graph technique with statistical models [9, 17, 19] 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. Shape-graph prefetching may be applicable to the work on smart memories [15, 21] in order to reduce memory latency for pointer based programs. Smart memories allow simple code to be executed at the level of memory by placing memory, a local interconnect and a processor core on a single memory chip. As our graph interpretation algorithm is quite simple, program information could be preloaded allowing the memory to calculate the future memory references.

Our technique is applicable to mobile distributed databases and other applications. We have examined the technique on several benchmarks using a simulated communication system with disconnections. Results show speedups upto 80.8% in the presence of disconnections over competing techniques. SPREE is unique in that we can support large lookaheads and eliminate cold misses without training.

References

1. J.-H. Ahn and H.-J. Kim. SEOF: an adaptable object prefetch policy for object-oriented database systems. In *International Conference on Data Engineering*, pages 4 –13. IEEE, Jan 1997.
2. 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.
3. J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA*, pages 35–465, Denver, CO, Nov 1999. ACM.
4. J. Bogda and A. Singh. Can a shape analysis work at run-time? In *Java Virtual Machine Research and Technology Symposium (JVM'01)*, Monterey, California, Apr 2001. USENIX.
5. Cahoon and McKinley. Data flow analysis for software prefetching linked data structures in java. In *International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, Sep 2001. ACM.
6. P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *SIGMETRICS*, Ottawa Canada, May 2001. ACM.
7. M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, 1993.
8. F. Corbera, R. Asenjo, and E. 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.
9. K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *SIGMOD Conference on Management of Data*, pages 257–266. ACM, May 1993.
10. 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.
11. C. A. Gerlhof and A. 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.
12. R. Ghiya and L. 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.

13. 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.
14. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, Utah, Jun 2001. ACM.
15. C. J. Hughes and S. V. Adve. Memory side prefetching for linked data structures. *Journal of Parallel and Distributed Computing (JPDC)*, May 2001.
16. A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, Mar 1997.
17. D. Joseph and D. Grunwald. Prefetching using Markov predictors. *IEEE Transactions on Computers vol 48 (2)*, pages 121–133, Mar 1999.
18. J. 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.
19. N. Knafla. *Prefetching Techniques for Client/Server, Object-Oriented Database Systems*. PhD thesis, University of Edinburgh, 1999.
20. 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.
21. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, Jun 2000. ACM.
22. D. Nurit, R. Michael, and S. 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.
23. ODMG. *Object Query Language*, 2003.
24. 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.
25. S. 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.
26. E. Ruf. Effective synchronization removal for Java. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI2000),*, Vancouver, British Columbia,, Jun 2000. ACM.
27. Y. Shuf, M. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of java workloads: A structured view and opportunities for optimizations. In *Proceedings of SIGMETRICS*, Cambridge, MA, Jun 2001. ACM.
28. Sun Microsystems. *Java Data Objects*, 2003.
29. 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.