Symbolic Prefetching in Transactional Distributed Shared Memory

Alokika Dash
Electrical Engineering and Computer Science
University of California, Irvine
adash@uci.edu

Brian Demsky
Electrical Engineering and Computer Science
University of California, Irvine
bdemsky@uci.edu

Abstract

We present a static analysis for the automatic generation of symbolic prefetches in a transactional distributed shared memory. A symbolic prefetch specifies the first object to be prefetched followed by a list of field offsets or array indices that define a path through the heap. We also provide an object caching framework and language extensions to support our approach. To our knowledge, this is the first prefetching approach that can prefetch objects whose addresses have not been computed or predicted.

Our approach makes aggressive use of both prefetching and caching of remote objects to hide network latency. It relies on the transaction commit mechanism to preserve the simple transactional consistency model that we present to the developer. We have evaluated this approach on several shared memory parallel benchmarks and a distributed gaming benchmark to observe speedups due to prefetching and caching.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Distributed Programming

General Terms Design, Algorithms

Keywords Symbolic prefetching, Transactional memory, Distributed shared memory

1. Introduction

Developing software that makes efficient use of clustered computing while managing program complexity can be challenging. While previous generations of high-performance computers commonly provided developers with a shared memory, modern clusters typically do not provide shared memory. Instead, the underlying hardware supports communication between processing nodes through message passing primitives.

Researchers have developed software distributed shared memories to provide developers with the illusion of a simple shared memory abstraction on message passing machines. A straightforward implementation of a distributed shared memory can provide developers with a simple memory model to program. One of the primary challenges in designing object-based distributed shared memory systems is hiding the latency of accessing remote objects. Previous work on transactional distributed shared memory primarily focused on providing transactional guarantees and largely overlooked a promising opportunity for utilizing the transaction commit mechanism to safely enable optimizations. Also, many traditional approaches to prefetching [Speight et al.2002, Joseph et al.1997] have had limited success hiding the latency of remote object accesses in the distributed environment because they require to compute an object's address or accurately predict an object's address before issuing a prefetch for that object. Our approach describes prefetches in terms of objects through the heap enabling it to prefetch objects whose addresses are not yet known.

We present a static analysis to generate prefetches for a distributed shared memory that is based on the transactional memory model [Harris et al.2006, Bocchino et al.2008]. This new unified compile-time analysis allows for software prefetching of arrays and linked structures for our transactional distributed system that aids in system performance. We have focused on small clusters of servers interconnected with ethernet networks.

2. Prefetch Design

Transactional distributed shared memory creates a new opportunity to safely and speculatively prefetch and cache remote objects without concern for memory coherency — the transaction commit process ensures that only transactions that access the latest versions of objects can commit. Many traditional address-based prefetching approaches were largely designed for hiding the latency to access local memory — such prefetching incurs large latencies when accessing remote linked data structures because the computation must wait to compute an object's address before prefetching the object. In effect this requires waiting for a round trip communication for each object to be accessed in a remote linked data structure.

We introduce a new approach to prefetching objects in the distributed environment that leverages the computational capabilities of the remote processors. Our approach communicates symbolic prefetches, which describe a path through the heap that traverses the objects to be prefetched. Symbolic prefetches have the form:

$$\textit{symbolic prefetch} := \textit{base object identifier}(.\textit{field} \mid [\textit{integer}])^*$$

The base object identifier component of the symbolic prefetch gives the object identifier of the first object to be prefetched. The list of field offsets and array indices describe a path through heap from the first object. We present a compiler analysis that enables our implementation to efficiently generate prefetches for complex linked data structures.

Our prefetching scheme enables prefetching multiple objects even multiple references away with a single round-trip network communication. The roundtrip network latency on a gigabit LAN between commodity workstations is approximately 100 μS. On a modern 3 GHz processor, this corresponds to waiting 300,000 clock cycles. Therefore, the guiding principle for our design is to avoid waiting on network responses whenever possible.

Let us consider the following example segment:

```
1 LinkedList search(int key) {
2   LinkedList ptr=head;
3   while (ptr!=null&&ptr.key!=key)
4     ptr=ptr.next;
```

Copyright is held by the author/owner(s).
PPoPP'10, January 9–14, 2010, Bangalore, India.
ACM 978-1-60558-708-0/10/01.

```
5   return ptr;
6 }
```

Without prefetching, completely searching a remote linked list of length n requires making n consecutive round-trip message exchanges. If we add a prefetch for `ptr.next.next.next.next.next` between lines 3 and 4, the runtime will have prefetch requests in flight for the next linked list node and the subsequent four nodes that follow that node[1]. The example prefetch enables the `search` method to potentially execute five times faster. Longer symbolic prefetches can further increase the potential speedup.

2.1 Prefetch Analysis

Our analysis is an intraprocedural static analysis that uses a simple probabilistic model to generate prefetches that a program may access with probabilities that tell how likely the objects , represented by a path in the heap, will be accessed. It is a backward flow analysis that computes a set of tuples containing a symbolic prefetch and a corresponding probability for each program point. The probabilistic model is naive — it makes assumptions of independence that are not true in general. However, the results need not be precise, but simply provide a rough approximation of the real program's data access patterns.

Our analysis associates a probability with each conditional branch. By default, we assume that loop branches take the true branch with an 80% probability and other branches take the true branch with a 50% probability. We ensure the termination of the analysis by introducing a minimum symbolic prefetch probability μ. If a symbolic prefetch has has a probability less that μ at a program point, the analysis drops that symbolic prefetch.

2.2 Prefetch Placement

There is a trade-off between placing prefetches early to minimize the time that the application waits for data and waiting long enough to make sure the program is likely to use the prefetched data. This trade off can depend on the specific architecture of the machine and the application — bandwidth constraints can be satisfied by delaying prefetches, while latency constraints can be satisfied by moving prefetches earlier in the execution.

We instrument the analysis in the previous section to record the mapping which maps the symbolic prefetch at the source of the edge to the corresponding symbolic prefetch at the target of the edge. Prefetches are placed on edges where the probability of using the objects specified by a symbolic prefetch crosses the developer specified threshold. In order to avoid redundant prefetches we check whether the symbolic prefetch has already been prefetched by using a set at each program point. This set is the intersection of the set of prefetched symbolic prefetches along each incoming edge to a node. We use a fixed point algorithm to compute these sets for all program points. At each edge, our prefetch placement algorithm places prefetches for the set of symbolic prefetches that cross the threshold but have not already been prefetched.

2.3 Prefetch Runtime Mechanism

Our analysis places prefetch calls which take as input an array of base object identifiers for each prefetch, and an array of unsigned shorts that stores a sequence of the combination of field offsets and array indices for every prefetch at that site. The runtime maintains a prefetch queue. The main execution thread enqueues prefetch requests into this queue – each prefetch of the prefetch call is translated into a prefetch request. The runtime maintains a separate thread called prefetch thread that processes the prefetch requests from the prefetch queue. The prefetch thread processes as much of the prefetch request as possible locally before sending the request to the remote machines. The local processing starts by looking up the base object identifier component of the prefetch request in both the local heap and the object cache – in many cases the local heap and cache may already contain many of the objects in the request. If the object is found locally, the local runtime system uses the field offset (or array index) to look up the object identifier of the next object in the path and remove the first offset value from the symbolic prefetch. The runtime repeats this procedure to process the components of the prefetch request that are available locally.

The runtime then prunes the local component from the prefetch request to generate a new prefetch request with the first non-locally available object as its base object identifier. The local machine next sends the prefetch requests to the remote machines. Each request contains the machine identifier that should receive the response.

When the remote machine receives a prefetch request it begins by looking up the base object identifier in its local heap and then (optionally) if necessary in its object cache. Once it finds the object, it looks up the next object identifier by using the field offset or array index from the symbolic prefetch. It repeats this process until it has served the complete request. While serving the request it keeps sending the copies of the objects to the original machine.

[1] The prefetch look-ahead distance is not fixed. Instead it depends on the analysis's estimation of how likely the prefetched values are to be used.

3. Conclusion

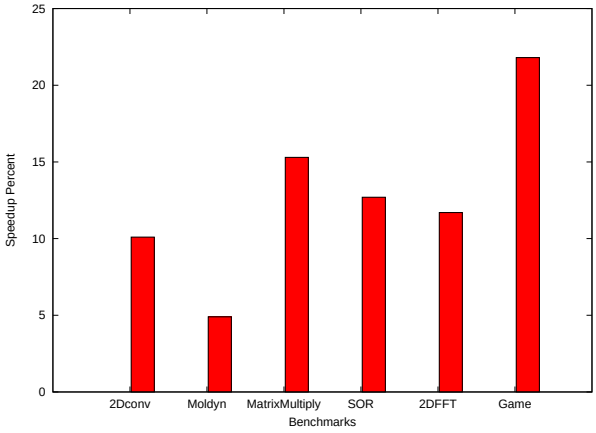

Figure 1. Speedup % due to prefetching

We have presented an analysis to generate symbolic prefetches for objects. Using this analysis we are able to generate useful prefetches successfully for our benchmarks. We observe gains upto 22% for our benchmarks as shown in Figure 1 and upto 99% hits in the cache due to prefetching.

References

[Harris et al.2006] Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. 2006. Optimizing memory transactions. SIGPLAN Not. 41, 6 (Jun. 2006), 14-25. DOI= http://doi.acm.org/10.1145/1133255.1133984

[Bocchino et al.2008] Bocchino, R. L., Adve, V. S., and Chamberlain, B. L. 2008. Software transactional memory for large scale clusters. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Salt Lake City, UT, USA, February 20 - 23, 2008). PPoPP '08. ACM, New York, NY, 247-258. DOI= http://doi.acm.org/10.1145/1345206.1345242

[Speight et al.2002] Speight, E., Burtscher, M.: Delphi: Prediction-Based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems. In: Intl. Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, pp. 49.55 (June 2002)

[Joseph et al.1997] Joseph, D. and Grunwald, D. 1997. Prefetching using Markov predictors. In Proceedings of the 24th Annual international Symposium on Computer Architecture (Denver, Colorado, United States, June 01 - 04, 1997). ISCA '97. ACM, New York, NY, 252-263. DOI= http://doi.acm.org/10.1145/264107.264207