Software 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 have developed a transaction-based approach to distributed shared memory(DSM) that supports object caching and generates path expression prefetches. A path expression specifies a path through the heap that traverses the objects to be prefetched. To our knowledge, this is the first prefetching approach that can prefetch objects whose addresses have not been computed or predicted. Our DSM uses both prefetching and caching of remote objects to hide network latency while relying on the two-phase transaction commit mechanism to preserve the simple transactional consistency model that we present to the developer. We have evaluated this approach on a matrix multiply benchmark. We have found that our approach enables to effectively utilize multiple machines in a cluster and also benefit from prefetching and caching of objects.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Distributed Programming; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications

General Terms Design, Algorithms

Keywords Transactional memory, Distributed shared memory, Prefetching objects, Path-expression prefetch

1. Introduction

Price decreases in commodity hardware have led to the widespread adoption of cluster computing. Developing software for these clusters can be challenging. While previous generations of high-performance computers commonly provided developers with a shared memory, modern clusters typically do not provide the developer with a shared memory. Instead, the underlying hardware supports communication between processing nodes through message passing primitives. As a consequence, the already challenging task of developing parallel software has become even more difficult. Developers must now reason about communication patterns, write code to traverse and marshall possibly complex data structures into messages, write communication code to interface with MPI or PVM to route these messages from producers to consumers (Gropp et al. 1996; Geist and Sunderam 1991), and write code to unmarshall these message back into data structures.

In response to this trend, researchers have developed software distributed shared memories to provide developers with the illusion of a simple shared memory abstraction on message passing machines (Li 1988). A straightforward implementation of a distributed shared memory can provide developers with a simple memory model to program. However, accessing remote objects in such implementations requires waiting for network communication and therefore is expensive. In response to this issue, researchers have developed several distributed shared memory systems (Keleher et al. 1994) that achieve better performance through relaxing memory consistency guarantees. However, developing software for these relaxed memory consistency models can be challenging — the developer must often read and understand sometimes complicated memory consistency properties to understand the possible behaviors of the program.

In recent years, a general recognition of the importance of programmer productivity has shifted the focus in computing research from solely performance to the more holistic focus of high productivity computing which encompasses programmer productivity. Both the Chapel and Fortress high performance computing languages include language constructs that specify that code should be executed with transactional semantics. These transactional constructs were included to potentially simplify software development by enabling developers to control concurrency without having to reason about potentially complex locking disciplines.

2. Approach

Our transaction-based approach to distributed shared memory presents a simple programming model to the developer. It uses a set of language extensions to a subset of Java to support transactions. Each shared object is annotated with `shared` keyword while each block of code with transactional semantics is annotated with `atomic` keyword represented by a pair of braces(e.g. { }). The shared memory extensions are similar to those present in Titanium (Yelick et al. 1998) though our use of transactions introduces additional constraints on when the application may access shared objects. Figure 1 shows an example of a Java source code using these keywords.

Our compiler generates C code from the Java source code. Figure 1 shows the C code generated for the `run` method of the `bar` class. Local objects and shared objects are accessed in the context of a transaction. Whenever a transaction writes to a local object, the compiled code first checks if there is a copy of the object's state and then makes a copy if necessary. These copies are used to revert the local objects back to their original states if a transaction aborts.

When an object is newly allocated on a machine, it permanently resides on that machine. This is called the authoritative copy of the object and it contains the most recently committed version of the object. We use the standard two-phase commit protocol (Gray and Reuter 1993) to commit changes to an object and use object version numbers to track committed changes to objects. The version number is incremented every time the authoritative copy of the object is changed. In the first phase of the two-phase commit, each participant verifies that the transaction has only accessed the latest versions of the objects and in the second phase it votes to abort if the transaction accessed an old version of any object. If all participants vote to commit, the coordinator sends a commit command. Thus, a transaction commits if it only accessed the latest versions

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.
ACM 978-1-60558-397-6/09/02.

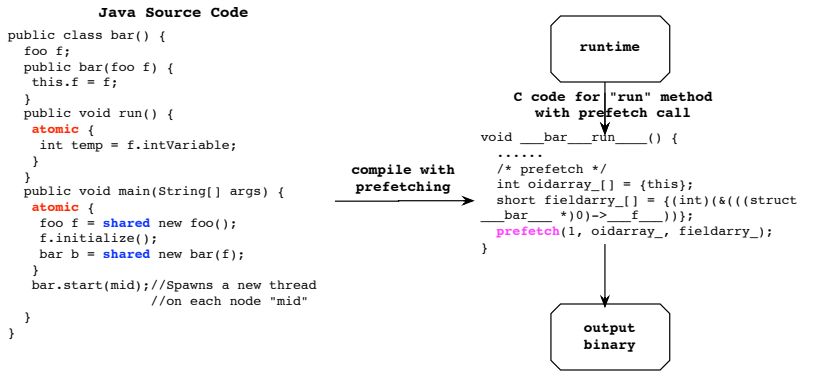

Figure 1. Block diagram of our approach

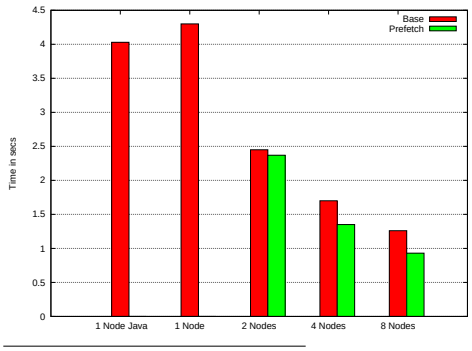

Figure 2. Matrix Multiply

of objects. If any machine votes to abort, the system re-executes the transaction.

One of the primary challenges in designing 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. Our approach prefetches and caches remote objects and relies on the transaction commit checks to safely recover from mis-speculations.

Traditional approaches to prefetching have had limited success hiding the latency of remote object accesses in the distributed environment because they require the computation to first compute or accurately predict an object's address before issuing a prefetch for that object. Our approach describes prefetches in terms of paths through the heap enabling it to prefetch objects whose addresses are not yet known. Path expressions represent a base object followed by a list of field offsets or array indices. The base object identifier component of the path expression gives the object identifier of the first object in the path expression. The sequence of field offsets and array indices describe a path through heap from the first object.

We have developed an unsound, intraprocedural static analysis that uses a simple probabilistic model to generate a set of path expressions that the program may access and the corresponding estimated probabilities. These probabilities represent how likely the objects, in a given path expression, will be accessed. It is acceptable for the analysis to be unsound because prefetches do not affect the program's correctness. Our runtime issues a prefetch call for several path expression prefetches. In Figure 1 we can see that our static analysis generates a prefetch call for the object referenced by the `f` field in the `run` method. The runtime system processes a prefetch call is the following manner: It processes as much of the prefetch request as possible locally before sending the prefetch request to the remote machines. When the remote machine receives a prefetch request it begins with the object identifier. It looks up the object identifier first in its local distributed heap and then (optionally) if necessary in its object cache. Once it locates the object, it looks up the next object identifier by using the field offset or array index from the path expression and sends the prefetch response to the original machine with copies of the objects. The remote machine repeats this process until it has served the complete request using list of fields offsets or array indices from the path expression.

3. Experience

We use a cluster of 8 identical 3.06 GHz Intel Xeon servers running Linux version 2.6.25 and connected through a gigabit switch to evaluate our approach. We have implemented the DSM system, path expression prefetching, the language extensions, and the prefetch analysis. Figure 2 shows the plots for the matrix multiply benchmark for a matrix of 600x600 elements . This figure shows the time taken for 1 Node Java which is a single-threaded non-transactional Java version compiled into C code, and time taken for 1, 2, 4, and 8 nodes that are implemented with transactional semantics in our DSM. All our numbers are averaged over ten executions with one thread running per node. The `Base` bars presents results without caching or prefetching and the `Prefetch` bars presents results with both caching and prefetching enabled. The prefetching versions are generated automatically using our static prefetch analysis. We observe significant speedup as we increase the number of nodes. We also observe a speedup of 4.33 times with the 8 node prefetching version as compared to the standard 1 Node Java implementation and a speedup of 35.5% with prefetching as compared to the non-prefetching 8 node version.

4. Conclusion

We have presented a new transaction-based distributed shared memory system with support for object caching. We have presented a new path expression-based prefetching algorithm that is the only prefetching algorithm to our knowledge that can prefetch objects before the object's address is computed or predicted. We have implemented the prefetching analysis, the language extensions, and the distributed shared memory system in our compiler. We have observed speedups as the number of machines increase and also observe benefits from prefetching objects.

References

G A Geist and V S Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 258–261, 1991.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

K Li. Ivy: A shared virtual memory system for parallel computing. In *Proceedings of the 1998 International Conference on Parallel Processing*, pages 94–101, 1988.

Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Gra ham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(10-13), September-November 1998.