Distributed Shared Memory: Experience with Munin

J.K. Bennett, J.B. Carter, A.L. Cox, E.N. Elnozahy,
D.B. Johnson, P. Keleher and W. Zwaenepoel
Department of Computer Science
Rice University
willy@rice.edu

April 20, 1992

1 Introduction

Distributed shared memory (DSM) is the provision in software of a shared memory programming model on a distributed memory machine [6]. We are exploring the use of DSM in a cluster-based computing environment of workstations and servers connected by a local internetwork.

We have concentrated so far on using DSM to program compute-intensive tasks on networks of workstations. For these applications, DSM is a superior programming model compared to message passing, because it relieves the programmer from having to worry about data motion. Second, DSM allows applications written for shared memory machines to be ported with relative ease to distributed memory machines. Finally, DSM offers a natural paradigm for integrating both the locally shared memory and the globally distributed memory of anticipated future networks of shared memory multiprocessors. In particular, in such an environment, local interprocess communication is likely to dominate and therefore should be implemented in the most efficient form possible, i.e., by using the (hardware) shared memory available. It then follows that to achieve a single paradigm for both local and remote interprocess communication, DSM is the natural choice.

2 The Performance Challenge

The cost of communication in a distributed memory environment is high relative to a shared memory machine. This is the case both in terms of message latency and in terms of processor cost to send a message. Moreover, naive implementations of DSM may lead to high amounts of message traffic, for instance as a result of spinning synchronization or false sharing. False sharing occurs when unrelated program objects, that are being modified by threads on different machines, appear in the same physical page. The challenge is to reduce communication without requiring the programmer to extensively restructure his program, which would negate many of the alleged advantages of DSM. In Munin, we have experimented with two techniques to overcome these problems: release consistency [4] and multiple consistency protocols [1].

3 Release Consistency

Release consistency was introduced by the DASH system at Stanford [4]. While still providing a programming model very close to true shared memory, release consistency masks the effects of latency and reduces the number of messages sent.

Release consistency distinguishes between ordinary memory accesses and synchronization accesses, and, among the latter, distinguishes between "acquires" and "releases". Various well-known synchronization constructs, such as locks and barriers, may be mapped into acquires and releases. With RC, modifications to shared memory by a processor p_1 must become visible to another processor p_2 only when a subsequent release of p_1 becomes visible on p_2. A program produces the same results under RC as under a more conventional sequentially consistent memory if a matching release-acquire separates all conflicting ordinary accesses. Most shared memory programs obey this restriction without need for additional synchronization.

The Munin implementation of RC is specifically tuned to reduce the number of messages sent, a key consideration for a software implementation. Unlike the DASH's hardware RC, which pipelines modifications, the Munin implementation buffers them until a release and then merges messages to the same destination in a single message. The Munin implementation thus reduces both the number of messages and the latency, whereas in the DASH implementation the concern is solely with latency.

We are implementing a new algorithm for RC, called Lazy Release Consistency (LRC) [5], which rather than "pushing" the modifications at a release, "pulls" them at an acquire. This results in additional message savings, especially in the case where the lock is re-acquired on the same machine.

4 Multiple Consistency Protocols

For each shared object, a consistency protocol can be chosen, based on the expected access pattern of the object [1]. Again, the goal is to reduce the amount of communication in keeping the object consistent.

Munin initially supported the following expected access patterns: *read-only, private, migratory, producer-consumer, reduction, concurrent-write-shared, result* [2]. After some experience with a prototype implementation, we are becoming convinced that a few protocol options suffice, with the others being implemented out of these basic building blocks. Fewer protocol choices facilitate automated choice of the protocol by the runtime system, supplanting the current annotations of shared data.

The principal protocol choices appear to be invalidate vs. update, and exclusive-write vs. shared-write. Shared-write protocols allow multiple copies of a page to be written concurrently, with the modifications being merged at the time of a release. We believe such write-shared protocols are essential in DSMs with large page sizes in order to avoid the ping-pong effect resulting from false sharing. The tradeoff between write-invalidate and write-update is well-known from shared memory cache management. Long write-runs favor write-invalidate, while high read-write ratios favor write-update.

5 Current Status

A Munin prototype has been implemented on an Ethernet network of 16 Sun-3 workstations running a modified version of the V-System [3]. Munin uses the message passing primitives of the V-System to implement remote communication, but message passing is not visible to the Munin programmer. Instead, a single global address space is provided, with thread manipulation and synchronization facilities (currently locks and barriers), as in a shared memory system.

The resulting performance is very promising. For such programs as successive over-relaxation and traveling salesman, the Munin version and a hand-coded message passing version differed by less than 10 percent in execution time. We feel that such a small performance loss is quite acceptable, given the advantages that Munin offers in terms of ease of program development [2].

Furthermore, significant reductions in message traffic and running time were achieved as a result of using release consistency and multiple protocols. Simulations with LRC using the Splash benchmark suite showed further reductions as a result of the lazy implementation.

6 Future Directions

We are building a new prototype, incorporating LRC, on SunOS and on Mach. In this prototype, we also intend to investigate the automatic selection of consistency protocols. For scientific problems, we want to explore the integration of compiler analysis and runtime optimization. We hope to port this prototype to next-generation workstations, which we expect to be small shared-memory multiprocessors. Many interesting issues will arise for a DSM on a network of shared memory machines, such as the proper (static and dynamic) division of fine-grain parallelism between shared memory, and large-grain parallelism across the network.

7 Conclusion

Munin's efficient distributed shared memory is based on a multi-protocol implementation of release consistency. Initial experience with a prototype implementation is very encouraging: the performance cost of Munin, compared to the underlying message passing system, appears to be small. This cost seems acceptable in light of the significant reduction of the complexity of programming in Munin, compared to programming in the underlying message passing system. Comparing the running times and the message traffic of Munin programs to a conventional sequentially consistent, write-invalidate scheme points out the advantages of release consistency and having multiple protocols.

References

[1] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 168–176, March 1990.

[2] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[3] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[4] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[5] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. Submitted to the 18th Annual International Symposium on Computer Architecture, November 1991.

[6] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.