A Measurement-Based Algorithm for Mapping Consistency Protocols to Shared Data

Christopher Diaz James Griffioen

Department of Computer Science
University of Kentucky
Lexington, KY 40506 USA
{diaz,griff}@dcs.uky.edu

Abstract

Distributed Shared Memory (DSM) systems often provide exactly one consistency protocol for all shared data [2, 4]. Recent systems adaptively select consistency protocols using heuristical analysis of recent access patterns [1, 11, 12, 14, 17]. Although approaches based on access patterns can significantly improve application performance, there are other factors that influence the performance of a consistency protocol, such as network bandwidth, congestion, latency and topology. These factors become particularly important in a wide-area environment, where access pattern analysis alone can lead to suboptimal performance [6]. In this paper we describe an algorithm for selecting consistency protocols on a per-segment or per-object basis. It uses a low overhead measurement-based performance metric to determine the most appropriate consistency protocol for each segment. Because the measurements are based on "observed" performance, the system can react to any LAN or WAN environment in which the application is running.

1 Introduction

Compute-intensive applications such as hydrodynamics, weather forecasting and genetic analysis play a vital role in scientific communities. These applications often require hours, days, or longer to execute, even on large-scale multicomputers. Performance speedup for these types of applications requires scaling the number of machines. As the number of machines increases, providing consistency among the machines becomes more costly. Consequently, selection of *appropriate* consistency protocols for shared data becomes very important for large-scale multicomputers. However, no single consistency protocol is optimal for all applications in all environments. Unfortunately, an application is often stuck with whatever consistency protocol the DSM provides.

Recent work has explored the concept of *adaptive consistency protocols* in which the DSM observes the application's access patterns and then selects an appropriate consistency protocol [1, 11, 12, 14, 17]. While access patterns provide some hints about which consistency protocol is the best, they do not necessarily tell the whole story [6]. Other factors, such as the size of data writes, can affect performance. Additionally, elements related to the computing environment, such as congested links, heavily loaded nodes, bursty traffic patterns, and network latency, can also affect performance. These elements become more significant in a wide-area environment.

In a wide-area network (WAN), an application's overall performance is often dictated by the network characteristics, rather than by the application's access pattern. This results from the observation that a segment's consistency protocol performance may differ depending on the network environment. Moreover, the consistency protocol for one segment may interfere with the consistency protocol of another segment (for example, the segments may compete for the WAN's limited bandwidth). In comparison, the bandwidth in a local-area network (LAN) environment may be sufficient to hide the network contention. As a result, an algorithm that selects a consistency protocol for each segment cannot simply consider the segment's access pattern because the access pattern may dictate a consistency protocol that actually degrades application performance.

This paper presents a low-overhead measurement-based approach that maps segment-specific consistency protocols onto segments in such a way that performance improves or, in the worst case, guarantees the same performance with that of an appropriate single consistency protocol. The difficulty in mapping segment-specific consistency protocols arises from the fact that:

1. The number of potential protocol-to-segment mappings grows exponentially with the number of segments used, and

2. Segments may not be independent. The consistency protocol selected for segment X may have a signifi-

cant impact on the choice of consistency protocol for segment Y.

For these reasons, finding the optimal protocol-to-segment mapping requires trying *all* possible mappings and measuring their performance, which is impractical.

The following section describes a new algorithm for mapping consistency protocols to shared data on a per-segment basis. The algorithm searches for a local optima using performance measurements (that are gathered with low overhead) obtained from the running system. In cases where the segments' consistency protocols are independent (that is, do not influence one another), the algorithm will find the most appropriate mapping for the given environment. The algorithm also ensures performance will never be worse than using an appropriate single consistency protocol for all segments. Having outlined the protocol, section 3 describes how the performance measurements are obtained from the running system. Section 4 then presents our experimental results. Finally, Sections 5 and 6 describe related work and summarize our conclusions.

2 Mapping Protocols to Segments

In this section we present an algorithm for selecting the most appropriate consistency protocol for shared data on a per-segment basis. Like other systems that adaptively choose consistency protocols, we focus on long running iterative applications, because an iteration's behavior (such as which data is accessed and how many times) tends to indicate behavior of successive iterations [18, 19].

In the algorithm presented below, we assume that the overhead (delay) introduced by a consistency protocol can be accurately measured and recorded. Section 3 describes how the overhead can be measured and shows that the overhead measurement is closely correlated with the expected performance improvement. Specifically, we assume that the system continuously measures the consistency overhead introduced by each segment (*per-segment overhead*) and computes the total overhead caused by the consistency protocols (*combined overhead*) for all segments used by a machine. Given the ability to measure consistency overhead accurately, we employ a two phase algorithm that establishes a baseline performance in the first phase and then incrementally improves the performance in the second phase until a local optima is achieved.

To establish baseline performance, the DSM first selects and maps a single consistency protocol (say an *On-Demand* protocol, like that used with Entry Consistency systems [4]) to all segments. The application executes with this protocol for some period P while the DSM measures per-segment consistency overhead for the protocol. Section 3.3 gives definitions for P.

The DSM then selects and maps a second consistency protocol (in our case an *Update-Based* protocol, like that used with Eager Release Consistency systems [5, 16]) to all segments. The application executes for another period P while the DSM measures per-segment consistency overhead for the second protocol. This sequence repeats for all possible consistency protocols. After measuring the overhead of all possible "single consistency" protocols, the DSM then selects the single protocol with the minimal combined overhead as the *baseline* consistency protocol. We use this baseline as the limiting case. The second phase of the algorithm will not allow consistency overhead to exceed that for the baseline case.

The second phase attempts to further reduce consistency overhead by individually selecting an appropriate consistency protocol for each segment. The DSM uses measurements gathered in the first phase to aid the per-segment selection process. During the first phase, consistency overhead was measured for each segment using each consistency protocol. In other words, we know which protocol worked best for each segment. For the purpose of discussion, we will call the protocol that worked best the *optimal* protocol. We begin by assuming that segments are *independent*. We say two segments A and B are *independent* if and only if the consistency overhead of A cannot be influenced by the consistency protocol used by B under any choice of consistency protocols for A or B. Given this assumption, the consistency protocol for each segment can be adjusted individually. Setting each segment to its own optimal mapping will quickly lead to the overall optimal mapping. After setting each segment to its own optimal mapping, the DSM agains measures the consistency overhead of each segment for another period P. The DSM records mappings with the lowest combined overhead in the second phase.

Unfortunately, overhead dependencies between segments may exist and mappings made during the second phase may cause the combined overhead to increase rather than decrease. Alternatively, the combined overhead may decrease while overhead of certain segments increases. The algorithm then begins to search out a local optima by slowly changing the consistency protocol for some segments to their non-optimal protocol. The hope is that the small increased overhead experienced by these changed segments will allow those segments expecting large improvements to attain the expected improvements and benefit overall performance. If the segments are independent, changing some segments to their non-optimal protocol at worst degrades performance and causes the second phase to terminate.

To determine which segments to change, the DSM may take the following approach. The DSM first orders segments by the difference of their optimal and non-optimal protocol measurements, from smallest to largest. The DSM then changes those segments in the bottom δ of the order to their non-optimal protocols. Using a non-optimal protocol for these segments is least likely to degrade performance. The DSM then executes and measures the mapping for another period P. If performance is worse than the previous mapping, the second phase terminates. On the other hand, if performance improves, then setting some segments to their non-optimal protocols allowed the larger expected improvements for other segments to occur. The DSM then attempts to focus on a local optima by reducing δ so that a smaller percentage of the segments in the bottom of the order use a non-optimal protocol during the next measured period. The DSM continues to reduce δ after every period while performance improves.

Once the second phase terminates, the DSM uses the mapping with the lowest combined overhead, whether baseline or generated in the second phase. Note that the application may later change the way data is accessed or shared, for example with reassignment of work to machines. As a result, a mapping chosen by the algorithm may no longer be appropriate for at least some segments. We are investigating techniques for the DSM to reevaluate mappings after such an event so that appropriate protocols are reselected for the affected segments.

3 Measuring Consistency Overhead

In this section, we discuss the DSM environment used in our study, then explain how the DSM measures and compares consistency protocol overhead.

3.1 Measurement Environment and API

Although our measurement-based approach can be applied to the consistency protocols of any system, we will present the methods in the context of the Unify DSM system [7]. Unify provides a segment-based, single global shared address space. A Unify application can declare a segment to be as small as a single variable or large enough for several variables. Unify provides multiple consistency mechanisms. One mechanism employs an *On-Demand Protocol (ODP)* like the consistency protocol used in Entry Consistency systems [4]. A second mechanism employs an *Update-Based Protocol (UBP)* like the consistency protocol used in Eager Release Consistency systems [5, 16]. A Unify application may invoke ODP for a machine to request and wait for segment updates before accessing the segment. On the other hand, a Unify application may invoke UBP for a machine to disseminate updates after writing to the segment.

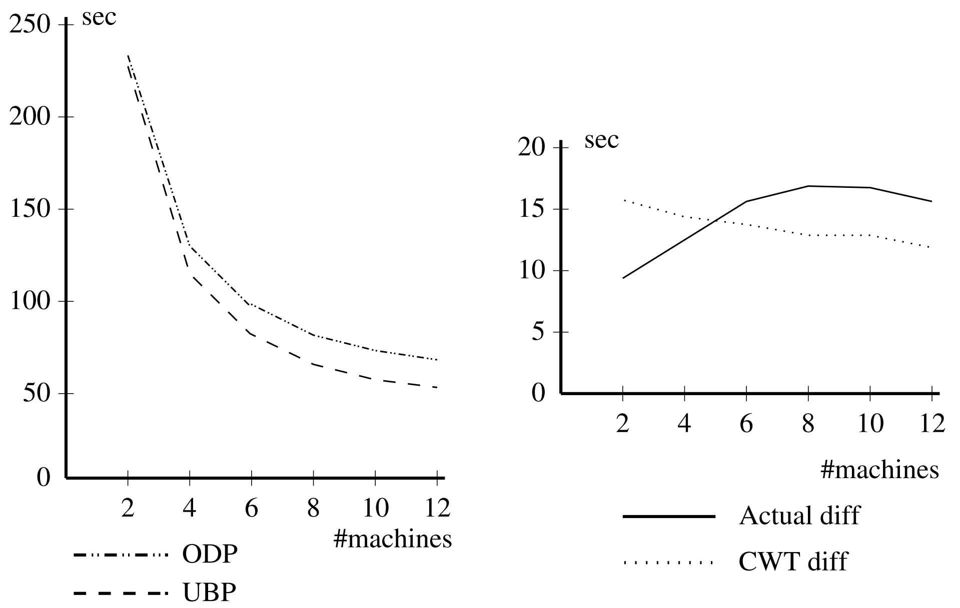
Because the DSM may dynamically select a segment's consistency protocol, we provide an API that ensures application correctness independent of the selected protocol. The API simply requires a Unify application to mark the beginning and end of shared data accesses. The application invokes `Read_Begin(seg)` and `Read_End(seg)` to mark the beginning and end, respectively, of a read-only access to a shared segment. Likewise, an application invokes `Write_Begin(seg)` and `Write_End(seg)` to mark the beginning and end, respectively, of a read-write access to a shared segment. These access primitives represent points in the application where the DSM can enforce the current consistency protocol. For example, suppose the DSM uses ODP. When a machine invokes `Read_Begin(seg)` or `Write_Begin(seg)`, the DSM retrieves necessary segment updates. Similarly, when the DSM uses UBP and a machine invokes `Write_End(seg)`, the DSM disseminates written changes to other machines.

3.2 Correlated Wait Time

Our measurement system measures consistency overhead of ODP and UBP to estimate the performance of one protocol over the other. To obtain consistency overhead, the DSM measures the *Correlated Wait Time (CWT)*[6] of each shared segment. CWT measures the wall clock time of all consistency activity related to the segment for which a machine *may* block. The idea behind CWT is to not only measure a machine's idle time, but to also measure the time a machine spends propagating shared memory modifications. Such propagations may slow the sending machine and delay its forward progress. Specifically, CWT measures the time a machine is idle due to consistency activity for a segment. When using ODP, CWT measures the time a machine waits for data to arrive at the beginning of a shared access. When using UBP, CWT measures the time a machine waits while the DSM reliably disseminates its shared data modifications. CWT also includes the time from the first to last packet a machine receives via UBP.

Note that a machine may concurrently receive updates for two segments during a time period. In such cases, the DSM divides the wall time of the period among the measurements for both segments. To compute a machine's combined overhead, the DSM totals the machine's CWT measurements of each segment.

To illustrate how CWT relates to actual performance, figure 1 shows the SPLASH2 [20] Water-Nsquared benchmark tested with 2197 molecules in a LAN of up to twelve machines. Figure 1a shows the runtime for three iterations with either an on-demand or update-based protocol for all segments. Figure 1b shows the actual performance difference between the two protocols, as well as the performance difference measured by CWT. For all numbers of machines, CWT identifies a performance gain for UBP over ODP and follows the actual difference curve once the number of machines scales.



(a) Water total runtime (b) UBP perf improvement over ODP

Figure 1: (a) Water total runtime and (b) Actual and CWT (as combined overhead) performance difference of an Update-Based Protocol (UBP) over an On-Demand Protocol (ODP) in a LAN.

CWT uses only timestamp information and thus requires little overhead. CWT does not require additional communication between machines.

3.3 Comparing Measurements

Each measurement is collected over some period P. In order to compare measurements, we need some assurance that the periods are the same duration and contain a similar number of accesses. Alternatively, we can use a period independent measure such as the average consistency time per access (that is, the *average access cost* for a protocol). We discuss two solutions based on these ideas.

First, the periods may be delineated by the application to measure similar periods, such as one iteration or multiple iterations. These measurements can be compared directly, because each period contains similar behavior [18, 19]. The second approach measures per-segment consistency overhead in terms of both time and number of segment accesses for the protocol. With these measurements, the DSM can compute a per-segment average access cost for that protocol. The average access cost allows the DSM to scale a per-segment measurement between two protocols for comparison when the number of accesses between periods is not the same.

4 Analysis

We used the SPLASH2 [20] Water-Nsquared and Explicit Hydrodynamics (Expl) [8] benchmarks with single protocol and per-segment mappings in LAN and WAN environments. Our LAN test environment consisted of twelve 125MHz HyperSparcs, each with 64MB of physical RAM, interconnected by a 100Mb Ethernet. Our WAN test environment consisted of sixteen 125MHz HyperSparcs, each with 64MB of physical RAM, in two domains of eight machines each. The domains were connected by three routers.

To illustrate the baseline cases (determined by the first phase of the algorithm), we ran the Water and Expl applications with all segments using an on-demand protocol (ODP) and then later with all segments using an update-based protocol (UBP). Figure 2 shows the total runtime for Water with 2197 molecules and three iterations if we were to do all ODP or all UBP for the entire run. In this case, UBP is the clear winner in the LAN but it is the worst protocol in the WAN. In a LAN, UBP has a 25% performance improvement over ODP at twelve machines, because UBP sends written data to machines that will likely access the data, thereby reducing fault latency. In a WAN, however, ODP has a 30% performance improvement over UBP at sixteen machines. In a WAN, UBP generates excessive network traffic that causes packet loss and router delays, increasing consistency overhead and decreasing performance as the system scales beyond ten machines.

It is important to note that if access patterns alone were used to select the consistency protocol, as is done with current adaptive systems, those systems that select an UBP approach would not scale in a WAN. Similarly, those systems that select an ODP approach would not scale best in a LAN.

In both the LAN and WAN, the per-segment measurements (PSM) found the best protocol. For Water, mapping each segment with its optimal consistency protocol (phase 2) with the PSMs produced performance the same or slightly better than the baseline performance. The reason for such behavior is that in Water, the few segments that required a protocol different than the baseline did not contribute significantly to the combined overhead.

As a second example, we analyzed the Expl application. Expl is a dense stencil kernel that computes values among nine grids. Each grid row is implemented as a segment. One machine initializes the grids before computation and gathers the grids afterwards. When the DSM uses only UBP, that machine receives updates during computation for data it does not need until the end. Consequently, UBP performance deteriorates quickly and is not represented in the graphs. Figure 3 shows the total runtime for Expl using ODP and PSM with 768x768 grids for 50 iterations in a LAN and WAN. In phase 2, PSM selects UBP for rows shared between machines and ODP for the other rows. UBP sends shared rows between machines, thereby reducing fault latency. UBP for these segments performs well in a WAN because the amount of their data is not large enough for machines to contend for the network. PSM has a 20% performance improvement over ODP, the baseline protocol, for sixteen machines in a WAN and a slightly smaller improvement for twelve machines in a LAN.

These applications have "optimal" mappings that are found in the first part of the second phase, which suggests the segments in these applications are largely independent. We are currently investigating applications with segment dependencies to test the local optima search in the second phase.

5 Related Work

Traditional DSMs [2, 4, 5, 9, 10] each typically provide applications with one consistency protocol. Such systems do not provide the most appropriate consistency protocol for every access pattern or require the application programmer to specify the desired protocol, which burdens the application programmer to analyze access patterns in detail.

Recent work enhances DSM to adaptively choose a consistency protocol for shared data at runtime to provide the most appropriate protocol for runtime speedup. Adaptive consistency may also alleviate the application programmer of the burden to analyze access patterns.

Monnerat and Bianchini developed ADSM [17] to enhance Lazy Release Consistency (LRC) [13] in TreadMarks [2] with update protocols for lock-protected and barrier-protected data to reduce fault latency. For lock-protected data, when the lock is acquired, ADSM propagates shared data modifications made when the lock was previously held. For barrier-protected data, ADSM heuristically analyzes the data's access pattern during the preceding three barriers. If during that period, only one machine modifies the data, ADSM then determines the readers in that same period. After the data is again written, ADSM propagates modifications to the readers. If the access pattern changes after only one or two barriers, the protocol does not change.

Rice developed ATMK [1, 3] to also supplement Lazy Release Consistency in TreadMarks with update protocols for lock-protected and barrier-protected data. ATMK reduces fault latency for lock-protected data with the technique used in ADSM. For barrier-protected data, ATMK heuristically analyzes the data's recent access pattern. At a barrier, ATMK propagates shared data modifications to machines that previously requested the data. If a machine receives such a modification but did not access the previous one, the machine sends a NACK to suppress subsequent propagations. When the access pattern frequently changes, ATMK may choose an on-demand protocol when an update-based protocol is appropriate and then alternatively choose an update-based protocol when an on-demand protocol is appropriate.

Keleher [11, 12] modified CVM to provide a heuristic-based invalidate/update hybrid protocol so that after every write to shared data, CVM sends the changes to machines that previously faulted on the data. In situations when a machine does not read every write to shared data, however, the protocol may send unnecessary data to the machine.

Tapeworm[14] provides the abstraction of a *tape* for an application to record a series of shared data accesses. When the application later executes the same code, the application "replays" the tape to provide Tapeworm information about forthcoming accesses. For example, the replay suggests which machines read a write that occurred before the previous barrier, so when the write and barrier reoccur, Tapeworm sends changes to those machines and reduces fault latency. However, Tapeworm requires additional application code to manipulate tapes.

Lee and Jhon [15] also supplement Lazy Release Consistency with an update protocol. Their work uses application-level annotations of both shared data writes and the machines that read those writes to specify when and where to send written changes. However, such annotations require the programmer to understand the access pattern in detail.

Nguyen et. al. [19] present an approach that at runtime determines the appropriate number of processors that

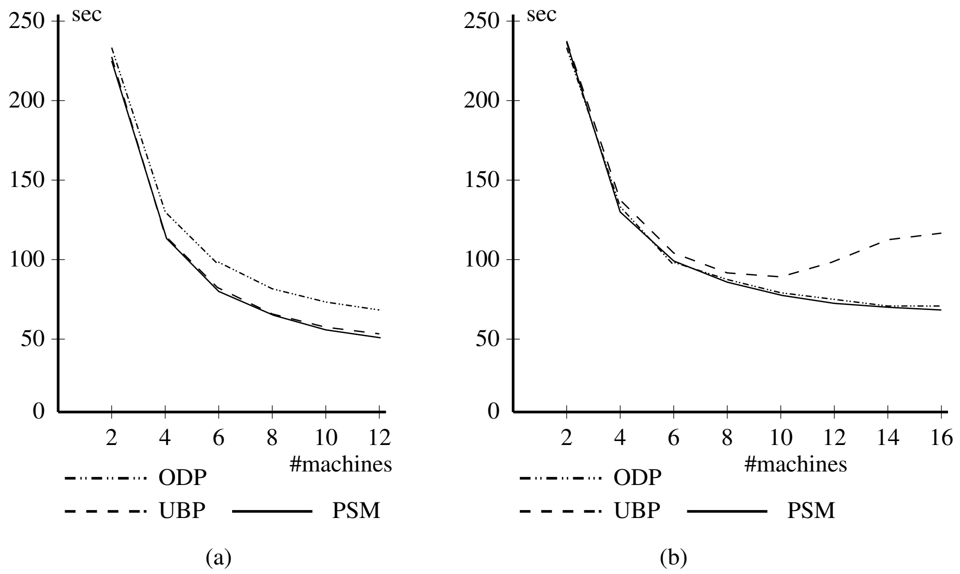


Figure 2: Water runtime with an On-Demand Protocol (ODP) used for all data, an Update-Based Protocol (UBP) used for all data, and a Per-Segment Measurement (PSM) Mapping (a) in a LAN and (b) in a WAN.

should be allocated to an application for optimal performance. Their approach measures various system overheads to gauge performance for a particular processor allocation. While our approach also measures performance, it does so to select appropriate consistency protocols rather than processor allocations.

Unlike our work, the approaches that select consistency protocols use either heuristic-based analysis of access patterns, require significant application code, or require detailed analysis of access patterns. While heuristics often choose the most appropriate consistency protocol, they sometimes do not and may cause runtime slowdown. Our system, on the other hand, evaluates performance between consistency protocols to identify an appropriate protocol for all data or for each piece of data.

6 Conclusions

We present an algorithm that uses per-segment consistency measurements to select an appropriate consistency protocol for each segment. The algorithm ensures that application performance is not worse then an appropriate single consistency protocol. We demonstrate that the algorithm adapts to LAN and WAN environments to select appropriate consistency protocols in cases where heuristic-based approaches fail to do so.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. In *Proceedings of IEEE, Special Issue on Distributed Shared Memory*, volume 87, pages 467–475, Mar 1999.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb 1996.

[3] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. In *Proceedings of the Third High Performance Computer Architecture Conference*, pages 261–271, Feb 1997.

[4] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the IEEE CompCon Conference*, 1993.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM symposium on Operating Systems principles*, pages 152–64, Oct 1991.

[6] C. Diaz and J. Griffioen. Measuring consistency costs for distributed shared data. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 2000.

[7] J. Griffioen, R. Yavatkar, and R. Finkel. Unify: A scalable approach to multicomputer design. *IEEE Computer Society Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(2), 1995.

[8] H. Han and C.-W. Tseng. Compile-time synchronization optimizations for software DSMs. In *Proceedings of the International Parallel Processing Symposium*, 1998.

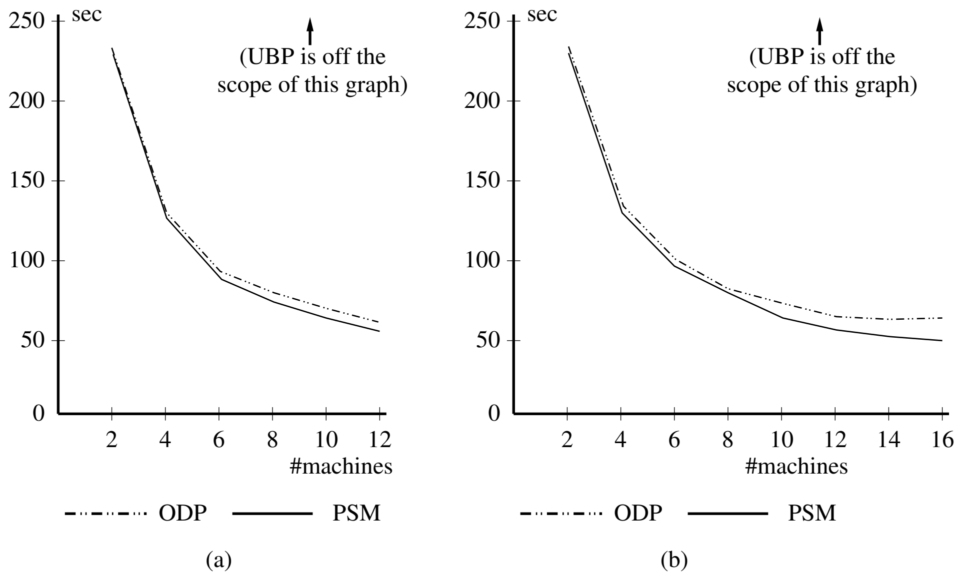


Figure 2: Expl runtime with an On-Demand Protocol (ODP) used for all data and a Per-Segment Measurement (PSM) Mapping (a) in a LAN and (b) in a WAN. Results with an Update-Based Protocol (UBP) for all data is out of scope for these graphs and not shown.

[9] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Dec 1995.

[10] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proceedings of the International Conference on Distributed Computing Systems*, Dec 1996.

[11] P. Keleher. Update protocols and iterative scientific applications. *The 12th International Parallel Processing Symposium*, Mar 1998.

[12] P. Keleher. Update protocols and cluster-based shared memory. *Computer Communications*, 22(11):1045–1055, July 1999.

[13] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the International Symposium of Computer Architecture*, pages 13–21, May 1992.

[14] P. J. Keleher. Tapeworm: High-level abstraction of shared accesses. *The 3rd Symposium on Operating System Design and Implementation*, Feb 1999.

[15] J. B. Lee and C. S. Jhon. Reducing coherence overhead of barrier synchronization in software DSMs. *SC98*, Nov 1998.

[16] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennesy, M. Horowitz, and M. Lam. The stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[17] L. R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software DSMs. In *Proceedings of the 4th IEEE International Symposium on High-Performance Computer Architecture*, Feb 1998.

[18] T. D. Nguyen, R. Vaswani, and J. Zahorjan. On scheduling implications of application characteristics. Technical report, University of Washington Department of Computer Science and Engineering.

[19] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing speedup through self-tuning of processor allocation. In *Proceedings of the International Parallel Processing Symposium*, April 1996.

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodoligical considerations. In *Proceedings of the 22nd International Synposium on Computer Architecture*, pages 24–36, 1995.