Measuring Consistency Costs for Distributed Shared Data

Christopher Diaz and James Griffioen

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

Abstract. Distributed Shared Memory (DSM) systems typically support one consistency protocol [3, 5, 6]. However, recent work [1, 11, 12, 14, 17] proposes the use of adaptive consistency based on a heuristical analysis of recent access patterns. Although heuristic-based approaches can significantly improve runtime, the access pattern alone does not necessarily define the most appropriate consistency protocol. The size of updates and other factors related to the computing environment, such as heavily loaded links, heavily loaded nodes, bursty traffic patterns, and network latency all affect performance. Multiple access patterns within the application also make it difficult to select the most appropriate consistency protocol. This paper presents a measurement-based approach to the problem of selecting the most appropriate consistency protocol for the current application in the current runtime environment. We show that measurement-based analysis provides an accurate estimate of performance and therefore can be used to select the most appropriate consistency protocol, even in cases where hueristic-based approaches fail to select the appropriate protocol.

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 the *most appropriate* consistency protocol becomes very important for large-scale multicomputers.

Past DSM research has resulted in a variety of consistency protocols [5, 6, 8, 13]. However, the most appropriate consistency protocol for a given application is difficult to identify and depends on a wide range of factors. In most systems, the application is stuck with whatever consistency protocol the DSM provides.

Some recent work, however, has explored the concept of adaptive consistency in which the DSM observes the application's access pattern and then selects an appropriate consistency protocol [1, 11, 12, 14, 17]. Given an application's recent access pattern, the DSM uses heuristics to map the observed access pattern to a consistency protocol. For example, if a machine writes to shared data that is subsequently read by N other machines, the DSM may decide to use an update-based approach, such as that used in Eager Release Consistency systems [6,16], where the newly written data is immediately disseminated to the N machines. On the other hand, if a machine writes to shared data that is subsequently read or written by a seemingly random machine, the DSM may take an on-demand approach, like that in Entry Consistency systems [5] or Lazy Release Consistency systems [13] to delay the update until the data is needed.

While heuristic-based approaches can significantly improve performance, they do not always select the best consistency protocol for the application in the current environment. As one example, consider iterative applications that alternate between access patterns, such as the SPLASH2 [20] Water-Nsquared (Water) and NAS [4] Integer Sort (IS) applications. Both applications exhibit an access pattern where shared data is written and then read by a set of machines. Later, both applications exhibit another access pattern where the same set of machines wait, in turn, to write the data. A heuristic-based approach that only analyzes access patterns may choose an update-based protocol because it sees the set requires updates at least some of the time. However, as we show in section 5, the size of updates relative to the network bandwidth or congestion can influence which protocol is most appropriate, and Water exhibits sparse writes compared to IS. Consequently, an update protocol is most appropriate for Water but is often not appropriate for IS.

As a second example, again consider applications with alternating access patterns. A heuristic-based approach may change consistency protocols to react to each change in the access pattern. However, unless the consistency protocol change is made quickly, the access pattern may immediately change again, as in the IS application. Here, such a heuristic-based approach often does not use an appropriate consistency protocol.

As a third example, consider an application where each machine writes to distinct shared data, synchronizes at a barrier, then reads the data written by all other machines. A heuristic-based approach typically selects an update-based protocol for this type of access pattern. However, other factors such as network congestion, caused by the burst of communication activity at the barrier may severely degrade performance to the point where an on-demand consistency protocol would have been better.

This paper presents a new measurement-based adaptive approach to select consistency protocols. The objective is to measure consistency protocol performance to get a clearer picture of an application's performance in the current runtime environment. The DSM estimates application runtime differences between consistency protocols by measuring each protocol's consistency related activity. This approach eliminates the need for an "active user" approach, where the user actively modifies application code, executes the application once with each consistency protocol, notes the runtime for each execution, and finally chooses a protocol. We also note that the appropriate protocol may change over the course of execution because of a remapping of work to machines or a change in the algorithm. In these cases, an active user approach is impractical because the user must detect the change, stop the application, modify application code, then restart or continue the application. To handle these situations, we are currently investigating ways for the DSM to continually reevaluate and rechoose appropriate protocols as necessary.

We present three approaches for measuring consistency overhead. Although these approaches differ in their degree of accuracy, we show that all three approaches can (1) be used to accurately determine the most appropriate consistency protocol for an application in the measured runtime environment, and (2) be used to accurately estimate the expected speedup (or slowdown) of one consistency protocol over another.

The basic mode of operation is quite simple and works as follows. A long running distributed application begins executing. Initially the DSM selects a consistency protocol, say an on-demand one. After some period P, the DSM changes to another consistency protocol. Again, after another period P, the DSM changes to a third consistency protocol. The application is unaware of changes to the consistency protocol. After measuring overhead of each consistency protocol, the DSM computes the expected runtime difference of one protocol over another and selects the protocol with the lowest observed runtime in the current environment.

The paper is organized as follows. Section 2 discusses DSM consistency protocols and other approaches to provide the most appropriate protocol for an application. Section 3 describes our measurement environment and application interface. Section 4 describes the approaches used to measure and estimate consistency costs. Section 5 presents experimental results obtained from a prototype system. Finally, we summarize our work in section 6.

2 Related Work

Traditional DSMs [3, 5, 6, 9, 10] each typically provide applications with one consistency protocol. Such systems cannot provide the most appropriate consistency protocol for every application. Systems that provide more than one consistency protocol require the application programmer to specify the desired one, which requires detailed analysis of application or protocol behavior.

Recent works enhance 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 burdens such as the need to analyze an application.

Keleher [11, 12] modified CVM to provide a heuristic-based invalidate/update hybrid protocol so that after every write to shared data, CVM propagates updates 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 updates to the machine.

Monnerat and Bianchini presented ADSM [17] to enhance Lazy Release Consistency [13] in TreadMarks [3] with update protocols for lock-protected and barrier-protected data to reduce fault latency. In the case of lock-protected data, when the lock is acquired, ADSM propagates changes 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 updates to the readers. If the access pattern changes after only one or two barriers, as in IS, the protocol does not change.

Rice developed ATMK [1, 2] 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 updates for data to machines that previously requested the data. If a machine receives such an update but did not access the previous update, the machine sends a NACK to prevent future propagation. When the access pattern frequently changes, as in IS, ATMK chooses an on-demand protocol when an update-based protocol is appropriate and then chooses an update-based protocol when an on-demand consistency protocol is appropriate.

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. The replay suggests which machines may read a write, so when the write reoccurs, Tapeworm sends updates 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 determine when and where to send updates after a write. To make such annotations, a programmer must understand the access pattern in detail.

These works either use heuristic-based analysis of access patterns to choose a consistency protocol for shared data or require additional application code, the latter which we do not require. While heuristics often choose the most appropriate protocol, they sometimes do not and may cause runtime slowdown. Our system, on the other hand, estimates the runtime differences between consistency protocols to identify which protocol provides the lowest runtime.

3 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 [7] DSM system. Unify provides a segment-based, single global shared address space. Unify provides multiple consistency protocols: an update protocol like that used in Eager Release Consistency systems [6, 16], an on-demand protocol like that used in Entry Consistency systems [5], and a protocol that combines both.

To use adaptive consistency protocols, a Unify application must follow semantics similar to Entry Consistency when accessing a shared segment. In short, one or more machines may concurrently read a segment. However, to write a segment a machine must have mutual exclusive access with respect to all other machines that want to read or write the segment. To mark the beginning and end of a shared segment access, a Unify application invokes `Read_Begin(seg)` and `Read_End(seg)` to specify 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 specify the beginning and end, respectively, of a read-write access to a shared segment[1]. Access primitives represent points in the application where the DSM may enforce the current consistency protocol. For example, suppose the DSM uses an on-demand protocol. When the application invokes a BEGIN, the DSM retrieves necessary segment updates. Similarly, when the DSM uses an update-based protocol and the application invokes `Write_End(seg)`, the DSM disseminates segment updates to all other machines.

Unify provides a third consistency protocol, called a *Demand-Update Protocol*, that combines on-demand and update-based approaches. When a machine issues a `Write_Begin`, the DSM provides an on-demand protocol. However, when a machine issues a `Read_Begin`, the DSM employs an update-based protocol to disseminate updates to machines that previously read the data. The idea behind this dissemination is that if machines concurrently read the data, the first read after a write is a precursor of reads by the other machines. The dissemination then provides consistency more quickly than, say, on-demand approach that updates the machines one at a time. Our measurement system compares all three consistency protocols.

In this paper, we do not elaborate on how the DSM initially selects a consistency protocol. The DSM may select a consistency protocol or decide when to change to another protocol based on heuristical analysis as described in CVM, ADSM or ATMK. Alternatively, the DSM may change protocols after some period P, where P is defined by the system or application as a period of time or other metric. In this study, P is defined by the application as some fixed number of iterations, because behavior in an iteration tends to indicate behavior in successive iterations [18, 19]. The goal of this paper is to measure performance of each protocol in the current runtime environment and then recommend the most appropriate protocol for the application in that environment. In this paper, we also assume that the DSM employs exactly one consistency protocol at a time, but we are exploring ways to measure consistency overhead so that each shared segment may be assigned its own appropriate consistency protocol.

---

[1] In the remainder of the paper we will use the term BEGIN to represent either a `Read_Begin` or `Write_Begin` and the term END to represent the corresponding `Read_End` or `Write_End`.

4 Measuring Consistency Overhead

This section describes three ways to measure consistency overhead. In the first approach, called *Access Time (AT)*, the DSM measures the wall clock time from a BEGIN to the corresponding END. AT represents a macro analysis, measuring not only consistency costs but all other costs that occur between the BEGIN and END. The motivation of AT is to measure all of the time machine spends accessing shared data. It can be argued this is the "perceived performance" and is what we want to optimize. Unfortunately, it also measures costs not associated with the segment, such as synchronization or changes in machine load. In the case where BEGIN/END pairs are nested, the DSM measures only the outermost BEGIN and END to represent the contributions of all nested accesses to the consistency overhead.

The second approach, *Wait Time (WT)*, measures the wall clock time a machine waits for consistency activity. The idea behind WT is to measure the time a machine is idle due to consistency activity. It can be argued that this is a better measure of consistency cost, but it ignores the consistency protocol's effect on other components of performance, such as prolonged synchronization activity. When an on-demand protocol is used, WT measures the time a machine waits for updates to arrive at the beginning of a shared region access. When using on update-based protocol, WT measures the time a machine waits while the DSM reliably disseminates updates.

The third approach, *Correlated Wait Time (CWT)*, measures the wall clock time of all consistency activity 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 spent propagating updates. Update propagation may slow the sending machine and delay its forward progress. CWT includes WT time but additionally measures the time from the first to last packet a machine receives via an update protocol. When an on-demand protocol is used, CWT measures the time for a machine to send a reply for an update request. Note that a machine may concurrently receive nested updates for two pieces of shared data. CWT measures such nested updates in a manner similar to AT.

5 Analysis of Measurements

We used the SPLASH2 [20] Water-Nsquared (Water), NAS [4] Integer Sort (IS) and Successive-Over Relaxation (SOR, included in the Unify [7] distribution) benchmarks to estimate and compare runtime differences between consistency protocols. Our test environment consisted of twelve 125MHz HyperSparcs, each with 64MB of physical RAM, interconnected by a 100Mb Ethernet. We separately ran each application with an on-demand protocol (OD), an update-based protocol (UB), then a demand-update consistency protocol (DU). During each execution, the DSM measured the consistency overhead for all approaches. We then compared protocols using the actual and estimated runtime differences to see whether our measurements could identify the most appropriate protocol and see how accurately they estimated runtime differences. To obtain "total runtime performance" information, we selected a consistency protocol and ran the application for a period P, defined in this study by the application as some fixed number of iterations. This was done for each application/consistency protocol combination. During each of these runs, we enabled our code to measure AT, WT and CWT consistency costs as well as runtime. Using the AT, WT and CWT measurements, we estimated the relative performance improvement of one consistency protocol over a second protocol by subtracting the measured cost of the first from that of the second. Similarly, actual performance improvement of one protocol over a second protocol is calculated by subtracting the runtime of the first from that of the second. We then compared our estimated improvement against the actual performance improvement.

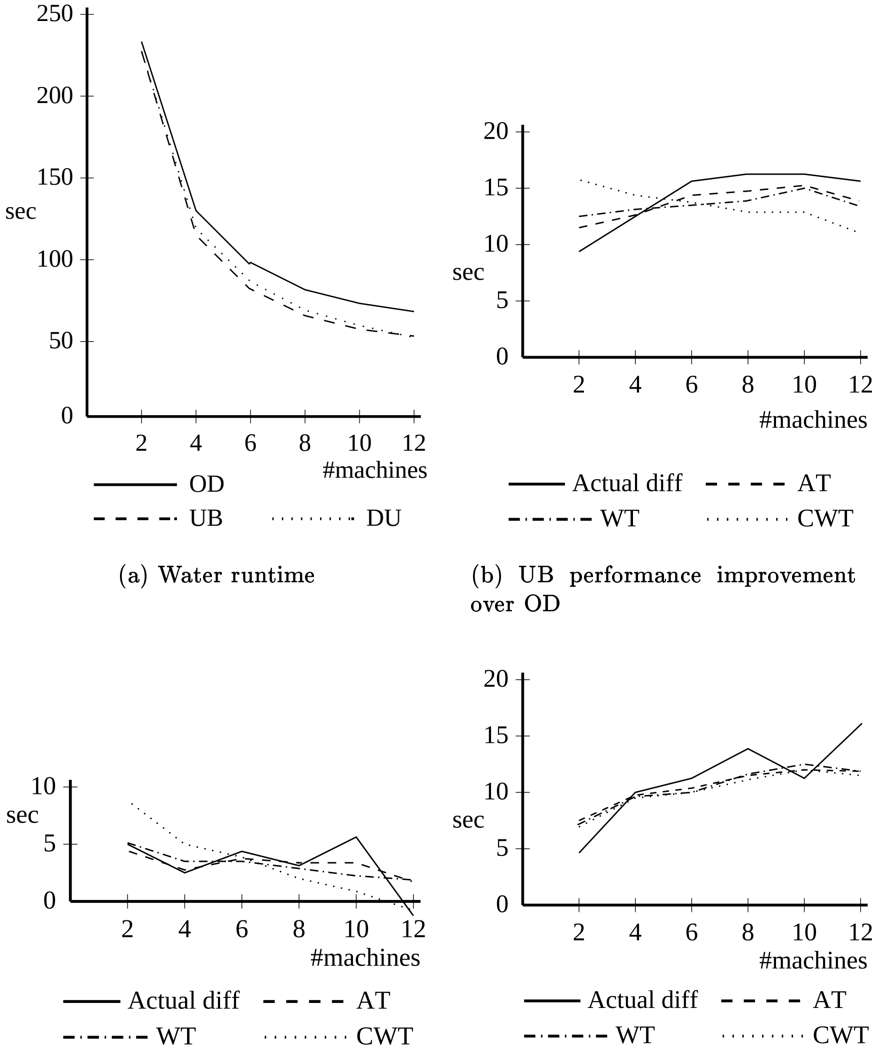
For each application, we show four graphs. The first graph shows the total runtime of each consistency protocol. The second graph shows the actual improvement UB gives over OD. It also shows the expected improvement predicted by AT, WT and CWT. The third and fourth graphs are similar to the second, but compare UB to DU and DU to OD, respectively.

In the first experiment (figure 1), Water is executed with 2197 molecules for three iterations. Figure 1(a) shows the total runtime for OD, UB and DU as the number of machines scales from two to twelve. As described in section 1, Water contains two distinct access patterns. In the first access pattern, OD suffers because to read, a machine requests updates and waits, possibly for updates to propagate among other machines first. On the other hand, UB and DU disseminate updates to machines that read the molecules, and thereby reduce wait time. In the second access pattern, machines write in turn to molecules. OD or DU appears better, because they send updates to a machine only when needed. However, the writes are sparse, so the size of updates is small. Consequently, UB does not cause excessive overhead because updates may not be overwritten before a machine needs them. As a result, UB scales better than OD, and DU scales similar to UB.

The solid line in figure 1(b) shows the actual runtime performance improvement obtained by using UB rather than OD (taken from figure 1(a)). The remaining lines in figure 1(b) show the improvement predicted by AT, WT and CWT. Note that AT, WT and CWT all predicted that UB is better than OD. Moreover, they accurately predicted how much the improvement would be. In this case, AT and WT are within 20% of the actual difference, and CWT is within about 40%.

Figures 1(c) and 1(d) show similar comparisons for the improvement of UB over DU and DU over OD, respectively. Again, AT, WT and CWT often correctly predict the most appropriate consistency protocol.

IS sorts 2^{24} keys with density 2^{19} 3 times. Figure 2(a) shows the total runtime for OD, UB and DU as the number of machines scales from four to twelve. IS contains access patterns similar to Water, except that writes to shared data overwrite a large portion of the data. As a result, UB does not scale well because the DSM sends large, unnecessary updates to many machines. While OD scales



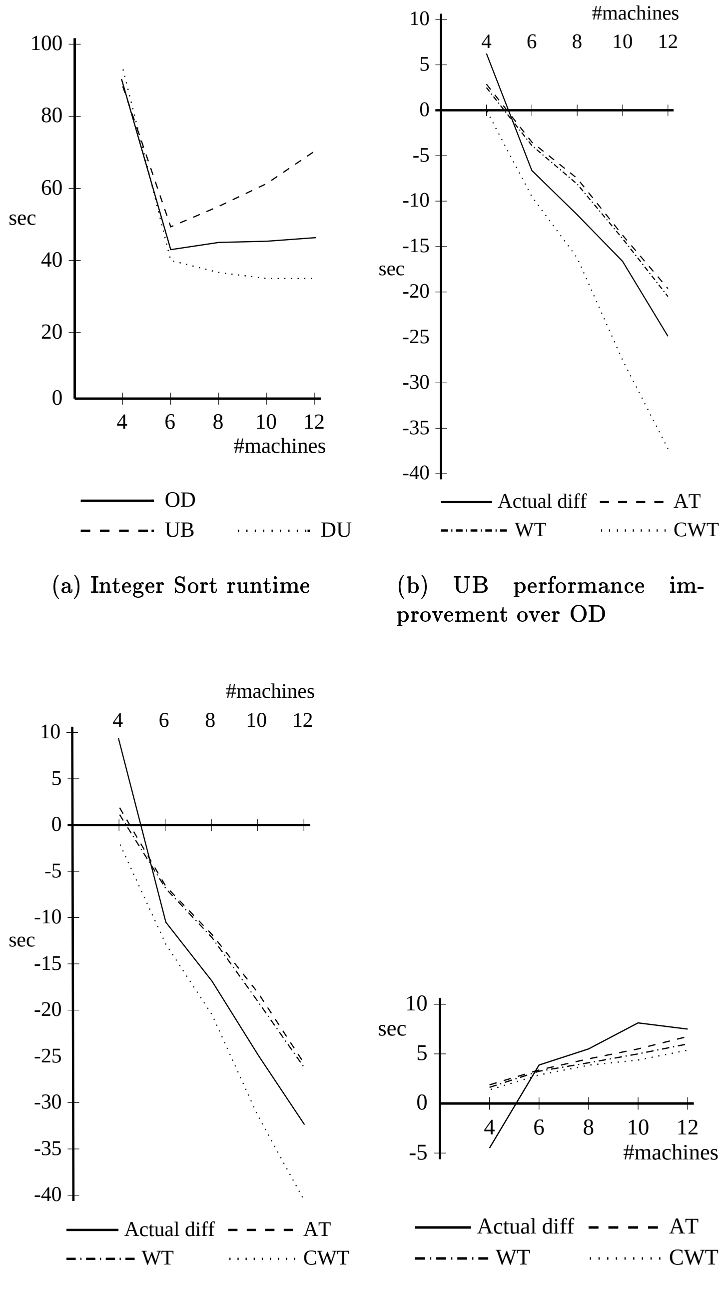
(a) Water runtime

(b) UB performance improvement over OD

(c) UB performance improvement over DU

(d) DU performance improvement over OD

Fig. 1. Measurement analysis of Water: 1(a) runtime in seconds; 1(b) actual and estimated runtime performance of an update-based protocol (UB) over an on-demand protocol (OD); 1(c) actual and estimated performance of UB over a demand-update protocol (DU), and; 1(d) actual and estimated performance of DU over OD.



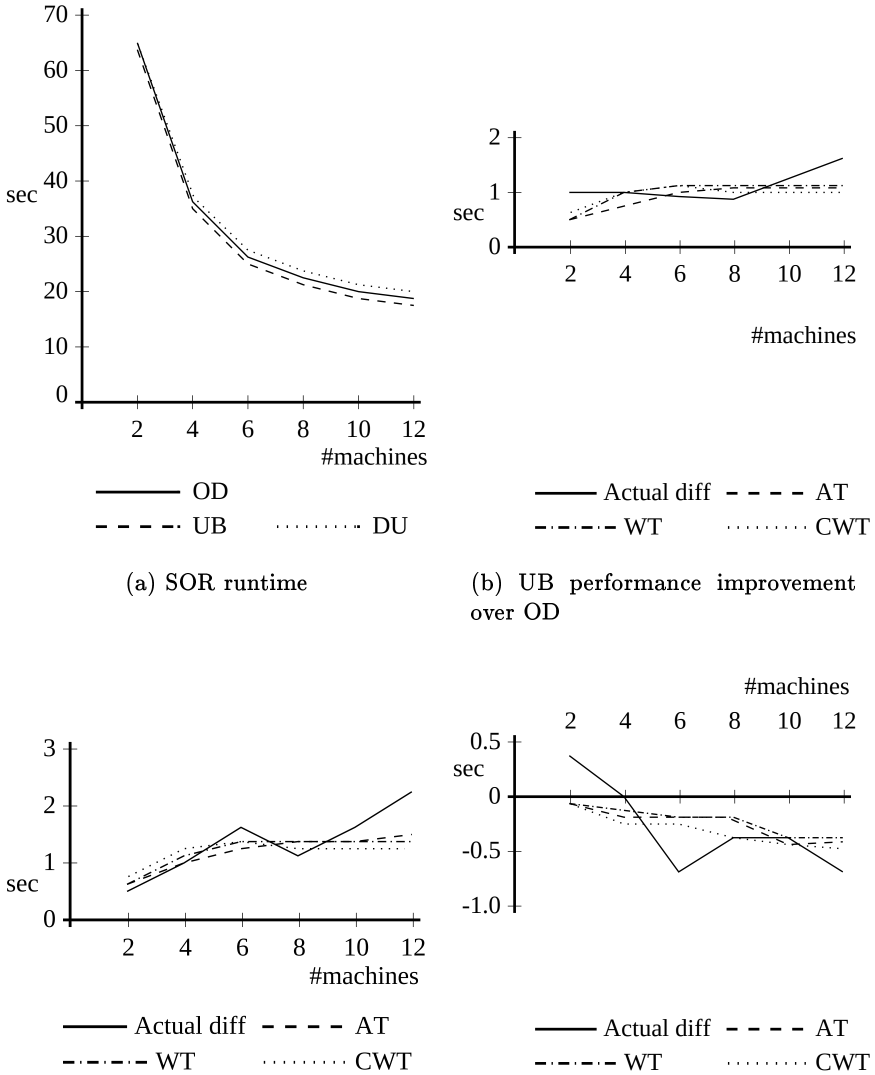
(a) Integer Sort runtime

(b) UB performance improvement over OD

(c) UB performance improvement over DU

(d) DU performance improvement over OD

Fig. 2. Measurement analysis of Integer Sort: 2(a) runtime in seconds; 2(b) actual and estimated runtime performance of an update-based protocol (UB) over an on-demand protocol (OD); 2(c) actual and estimated performance of UB over a demand-update protocol (DU), and; 2(d) actual and estimated performance of DU over OD .



(a) SOR runtime

(b) UB performance improvement over OD

(c) UB performance improvement over DU

(d) DU performance improvement over OD

Fig. 3. Measurement analysis of SOR: 3(a) runtime in seconds; 3(b) actual and estimated runtime performance of an update-based protocol (UB) over an on-demand protocol (OD); 3(c) actual and estimated performance of UB over a demand-update protocol (DU), and; 3(d) actual and estimated performance of DU over OD.

better than UB, OD suffers when each machine reads all shared data and waits for updates to propagate among machines. DU is most appropriate because it employs an on-demand protocol when machines write to shared data in turn and employs an update-based protocol when each machine reads all shared data.

Figure 2(b) shows the improvements obtained by using UB rather than OD. In this case, UB is worse than OD for more than four machines. AT and WT correctly identify OD as the better protocol, usually within 20% of the actual runtime difference. CWT mostly correctly identifies OD as the better protocol but overestimates the improvement by 50%. Figure 2(c) is similar. In figure 2(d), AT, WT and CWT usually all accurately predict DU as the most appropriate.

SOR is executed on a 1000x1000 grid for 100 iterations. Figure 3(a) shows the total runtime for OD, UB and DU as the number of machines scales from two to twelve. Although UB appears slightly better than OD, the two are so close, it is difficult to accurately predict which one is better. As a result, we see that the improvement predicted in figure 3(b) by AT, WT and CWT dances around 0 seconds. In this case, the system is best off using a *threshold* to say that the predicted improvement is too close to 0 to be accurate. On the other hand, whichever protocol is predicted will not significantly affect performance either way, so either protocol will not be too bad a choice. This situation is similar in figures 3(c) and 3(d).

Out of 51 LAN situations (three each for six Water, five IS and six SOR configurations), AT and WT both correctly identified 48 (94%), while CWT correctly identified 47 (92%). In most cases where a prediction was incorrect, the performance difference was close to zero, and one can argue that the prediction was too close to call. In these cases, an incorrect selection gives roughly the same performance as a correct selection. Furthermore, CWT correctly identifies cases for larger numbers of machines. All approaches estimate runtime improvement usually within 20% to 30% of the actual difference.

6 Summary

We present three approaches for DSM to measure consistency related activity in order to (1) accurately identify the most appropriate consistency protocol for an application in the given runtime environment, and (2) accurately estimate the expected speedup (or slowdown) of one consistency protocol over another. AT, WT and CWT are all good at identifying the most appropriate consistency protocol.

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, March 1999.

2. 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.
3. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb 1996.
4. D. Bailey, J. Barton, T. Lasinski, and H. Simon. The nas parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, Aug 1991.
5. Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the IEEE CompCon Conference*, 1993.
6. John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM symposium on Operating Systems principles*, pages 152–64, Oct 1991.
7. James Griffioen, Rajendra Yavatkar, and Raphael 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. Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures*, Jun 1996.
9. Kirk L. Johnson, M. Frans Kaashoek, and Deborah 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. Pete Keleher. Update protocols and iterative scientific applications. *The 12th International Parallel Processing Symposium*, March 1998.
12. Pete Keleher. Update protocols and cluster-based shared memory. *Computer Communications*, 22(11):1045–1055, July 1999.
13. Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the International Symposium of Computer Architecture*, pages 13–21, May 1992.
14. Peter J. Keleher. Tapeworm: High-level abstraction of shared accesses. *The 3rd Symposium on Operating System Design and Implementation*, Feb 1999.
15. Jae Bum Lee and Chu Shik Jhon. Reducing coherence overhead of barrier synchronization in software DSMs. *SC98*, Nov 1998.
16. Daniel Lenoski, Kourosh Gharachorloo, James Laudon, Anoop Gupta, John Hennesy, Mark Horowitz, and Monica 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. Thu D. Nguyen, Raj Vaswani, and John Zahorjan. On scheduling implications of application characteristics. Technical report, University of Washington Department of Computer Science and Engineering.
19. Thu D. Nguyen, Raj Vaswani, and John Zahorjan. Maximizing speedup through self-tuning of processor allocation. In *Proceedings of the International Parallel Processing Symposium*, April 1996.

20. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodoligical considerations. In *Proceedings of the 22nd International Synposium on Computer Architecture*, pages 24–36, 1995.