An Adaptive Coherence Protocol Using Write Invalidate and Write Update Mechanisms

Siniša Srbljić

Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia

A number of different systems (multiprocessor systems, distributed systems, and nowadays Internet) replicate data (cache lines, pages, user defined data structures, and Internet objects) in order to improve the performance, but it introduces a coherence problem. Many different protocols have been introduced to maintain coherence between the copies of the same date item. The performance of such systems is very sensitive to both the data access behavior of the application and the coherence mechanism. Adaptive coherence protocols use a decision function to dynamically choose the appropriate coherence mechanism. The choice of coherence mechanism is determined from the data access behavior with the goal of improving performance. The decision function presented in this paper chooses between a write invalidate and a write update mechanism, depending on which of these mechanisms incurs a lower cumulative communication coherence overhead. Simulation results are presented which demonstrate a significant improvement over other suggested solutions.

Keywords: cache coherence, replicated data, adaptive coherence protocols, decision function, memory access behavior, performances comparison.

1. Introduction

The coherence of replicated data in distributed, multiprocessor systems, and nowadays the Internet is of critical. Among the many factors that influence the performance of such systems, the choice of a coherence protocol plays a prominent role. The most popular protocols make use of the write invalidate or the write update mechanisms. It is well known that each of these mechanisms favors certain types of access behavior. This has motivated the idea of combining more than one basic mechanism in coherence scheme implementations.

To make a protocol adapt to the current data access pattern and to dynamically choose the best mechanism, a decision function must be implemented that decides which mechanism to use. All practical implementations of such decision functions have been based on observation of a sequence of consecutive accesses to a data item. The copies of data item are updated until a certain number of consecutive writes are performed by the same node (processor) such that no intervening accesses are performed (writes or/and reads) by any other node [1-6], after which the invalidation protocol is used.

An interesting evaluation of adaptive protocols based on such a decision function has shown that gains from the adaptive approach may not be as large as one might expect [6]. These analyses also show that no single adaptive scheme proposed so far is appropriate for both sequential and concurrent sharing.

To overcome this problem we propose having a different kind of decision function. Our decision function chooses between the write invalidate and the write update mechanism based on the cumulative communication coherence overhead. For each data item we keep statistics about the cumulative overhead for both the write invalidate and the write update mechanism. The decision function dynamically chooses the mechanism whose cumulative overhead is lower. In the rest of the paper we describe the implementation of this decision function and how it is applied in the adaptive coherence protocol. We discuss how to update the counters which keep the statistics of the cumulative overheads

during the protocol execution. While the suggested solution is applicable for both distributed and multiprocessor systems, we will present it for a distributed environment only.

The performance of the proposed protocol is compared using simulations to the performance of the Illinois protocol (which is chosen as a typical example of a write invalidate protocol) [8-10], the Firefly protocol (which is a typical example of a write update protocol) [9-11], a modified RWB adaptive protocol (an adaptive protocol whose decision function switches from write update to write invalidate after three successive writes from a node without intervening writes from other nodes [2]), and an ideal EDWP (an adaptive protocol similar to the modified RWB protocol except that both intervening reads and writes from other nodes will prevent switching from the write update to write the invalidate mechanism [4]).

The next section gives the necessary background. Section 3 presents the proposed adaptive scheme. Simulation results are reported in Section 4.

2. Background

This section describes the distributed system and the communication cost parameters we assume in our simulations. A brief informal description of the Illinois protocol, the Firefly protocol, the modified RWB adaptive protocol, and the ideal EDWP adaptive protocol in a distributed environment is also given.

2.1. Distributed system model

In our model, a distributed system consists of $N+1$ nodes. Communication between nodes is achieved via messages transmitted through first-in/first-out fault-free communication channels. Consistency is maintained by exchanging messages between the nodes of the system. There are N *clients* and one *sequencer* [12]. Some requests are executed *locally* and require no communication; the other requests are called *remote*. The sequencer globally sequences all remote accesses to the copies of the same data item. Table 1 shows the costs used in the model. The communication cost is assumed to be: S for transmission of one data item, P for transmission of update information for one data item, and unity cost for transmission of a command message. For example, the cost of transmitting a read request is 1. The cost of transmitting a read response, including the requested data item, is $S+1$. The value of P is typically less than S, because updating does not necessarily change an entire data item. The cost of a broadcast to N nodes is simply N times the cost of a basic message transfer (data or command).

N	number of clients
S	communication cost for transmission of one data item
P	communication cost for transmission of update information for one data item
1	communication cost for a command message
N	multiplier for the cost of a broadcast

Tab. 1. Distributed system parameters

2.2. Firefly protocol

The Firefly protocol [9–11] is a write update protocol. On a write from the sequencer, the update information is broadcast to all clients. As explained above, the communication cost for broadcasting the update information is $N(P+1)$. The communication cost of a write from a client is $N(P+1)+1$, which consists of $P+1$ for sending of update information from requesting client to the sequencer, $(N-1)(P+1)$ for broadcasting the update information from sequencer to all other clients, and 1 for returning a write permission command message from sequencer to the requesting client.

State associated with each copy determines the presence and the validity of the copy. Based on the given state and the given operation (read or write), protocol performs appropriate action and changes the state of the copy. The Firefly protocol does not cause an invalidation, and therefore an INVALID state (copy is present but it is not valid) is not used. Copies of all data items are present in the sequencer's local memory, which is assumed to be infinite, and are in the VALID state. The client's copies can be in one of two states: VALID (consistent with the sequencer's copy) and NOTIN (the copy is not present in the local memory). After a read from a sequencer, the client obtains a copy and changes its state from NOTIN to VALID. The communication cost for reading a copy from the

Mechanism	Remote operation	Communication cost (No. of packets)
Write Update	1. Sequencer's write operation	$N(P+1)$
	2. Client's read operation to sequence	$S+2$
	3. Client's write operation with reading to sequence	$(N-1)(P+1)+S+2$
	4. Client's write operation	$N(P+1)+1$
Write Invalidate	5. Sequencer's read and exclusive read operation to client	$S+2$
	6. Sequencer's write operation	N
	7. Sequencer's read and exclusive read operation to sequencer	$S+2$
	8. Sequencer's read and exclusive read operation to remote client	$2S+4$
	9. Sequencer's read and exclusive read operation to sequencer and invalidate	$S+N+1$
	10. Client's write operation	$N+1$

Tab. 2. Communication costs for remote reads and writes

sequencer is $S+2$, which consists of 1 for sending a read request to the sequencer and $S+1$ for returning a copy and the read response command. Once a copy is in the VALID state, all successive reads can be executed locally without sending any requests to the sequencer. The communication costs for all remote reads and writes (the communication costs for write update operations) are given in Table 2.

2.3. Illinois protocol

The Illinois protocol [8-10] is a write invalidate protocol. A write from a sequencer causes an invalidation message to be broadcast to all clients at a cost of N. The state of the sequencer's copy becomes DIRTY, while the clients' copies become INVALID. The DIRTY copy is the only valid copy in the distributed system. A write to the DIRTY copy can be executed locally without requiring commands to be sent. A read of an invalid copy at a client will cause a read request to be sent to the sequencer. After a read response, the state of the sequencer's copy changes from DIRTY to VALID while the state of the client's copy changes from INVALID to VALID. The communication cost of this read is $S+2$. After the client performs a write, its copy becomes DIRTY. All other copies, including the sequencer's copy, become INVALID. The DIRTY state enables the local execution of the client's write operations. The sequencer keeps information about the location of each DIRTY copy. Reading a DIRTY copy from a remote client results in sending a read request to the sequencer. The sequencer sends a read request to the client with the DIRTY copy. The client with the DIRTY copy returns the data to the requesting client through the sequencer. All three copies will be now in the VALID state. The communication cost of this operation is $2S+4$. There are three states for both the sequencer's and the clients' copies: VALID (the copy is consistent with all copies which are in the VALID state), DIRTY (the copy is the only valid one and all others should be in the INVALID state), and INVALID (the copy is not consistent). The client's copy can also be in the NOTIN state (the copy is not present in local memory). The communication costs for remote operations for a write invalidate protocol are given in Table 2.

2.4. Modified RWB and ideal EDWP adaptive protocols

The modified RWB and the ideal EDWP adaptive protocols use both the write invalidate and the write update mechanisms. While the write update mechanism is being used, the state transitions and the execution of operations are identical to the Firefly protocol. If the write invalidate protocol is being used, then the state transitions and execution of operations are identical to the Illinois protocol. Changing the mechanism from write update to write invalidate is not difficult because the set of states for the write update mechanism is a subset of the set of states for the write invalidate mechanism. The protocol simply applies the rules of the write invalidate mechanism since all global states for the write update mechanism are legal for the write invalidate protocol. The problem occurs when switching from the write invalidate mechanism to the write update mechanism when a DIRTY copy exists. The INVALID and DIRTY states are not legal states for the write update mechanism. This can be resolved by defining the

state transitions and the execution of operations for the INVALID state to be the same as for the NOTIN state. Also, the transition from the DIRTY state is interpreted as a remote operation from the VALID state.

We have implemented a modified version of the original RWB protocol [2]. The modified version switches from write update to write invalidate if there are three consecutive writes without intervening writes from other nodes, as suggested in the EDWP protocol [4]. We introduce two intermediate states instead of one as defined in the original protocol. This enables us to send two updates before an invalidation. The original protocol invalidates all copies on the second successive write from the same node. The second modification concerns operations on which switching should occur. In the original RWB protocol all remote operations, except local reads, prevent the adaptive scheme from switching to write invalidate from write update. In our implementation of the modified RWB protocol reads are not used to prevent switching. We do not use remote reads because they are visible only to the sequencer and not to clients.

We also simulated an ideal distributed version of the EDWP adaptive protocol [4]. The difference between the modified RWB protocol and the ideal EDWP protocol is that all reads, local and remote, are used to prevent switching from write update to write invalidate. To implement such a protocol in a distributed environment, additional communication overhead is needed in order for the reads to be visible to clients. We did not include this additional communication overhead in the simulation of the EDWP adaptive protocol. This was done in order to show the maximum possible gain for the given decision function. We call this protocol ideal EDWP. The second difference is in the state transition for the case where node i performs a write and node j has a DIRTY copy. In the modified RWB protocol, both nodes i and j will have the copies in the VALID state and the adaptive protocol will switch from the write invalidate to the write update mechanism after a write. The ideal EDWP protocol will change the state from DIRTY to INVALID in node i and from INVALID to DIRTY in node j. It will continue to use the write invalidate mechanism and will not switch to the write update mechanism.

In the modified RWB and the ideal EDWP adaptive protocols, all remote operations in Table 2 are possible for both the write invalidate and the write update mechanisms. In Section 4 we will discuss the results of simulations of the described coherence protocols. The performance of these protocols is compared with the performance of the protocol proposed in this paper.

3. Proposed adaptive protocol

In this section, we propose an adaptive protocol, called APCUM (Adaptive Protocol using CUMulative cost), which uses a decision function based on the cumulative communication cost. An early attempt at defining such a protocol was reported by Bunjevac in [7]. The decision function chooses dynamically between the write invalidate or the write update mechanisms based on which mechanism incurs lower cumulative communication coherence overhead. This section describes the implementation of the decision function in a distributed environment. We also discuss the state transition and the execution of operations necessary to handle both of the mechanisms in one coherence protocol.

The main idea for implementing the proposed decision function is to introduce two counters for each data item. One counter will keep the cumulative communication coherence overhead for the write invalidate mechanism, and the other for the write update mechanism. Both of these two counters are updated, even when only running in one mechanism. We call these two counters NPI (Number of Packets using write Invalidate mechanism) and NPU (Number of Packets using write Update mechanism). To maintain the cumulative overhead, the communication cost from Table 2 corresponding to the appropriate operation is added to the counter at each remote read and write operations. The APCUM protocol switches to the mechanism whose counter has a lower value. To avoid constantly switching back and forth, we use a decision function with hysteresis which will be explained in Section 3.3.

The NPI and NPU counters must reside in an appropriate place in the system so that all remote operations are properly taken into account. Since the role of the sequencer is to globally sequence all remote operations, we place the

two counters within the sequencer. Each time the sequencer receives a remote request, it updates both counters. The difficulty in maintaining these counters at the sequencer is that, while running under one mechanism a data access may be non- local, but may be purely local under the other mechanism, and that is that case the counters need to be updated even though the actual operation is local. The contribution of these local operations must also be added to the appropriate counter. When the write update mechanism is used, the NPI counter must be updated for the local clients' read operations, and when the write invalidate mechanism is used, the NPU counter must be updated for local executions of clients' write operations. In order to keep these statistics for local executions of operations, we include two additional counters to each client: the NRO counter (Number of Read Operations) and the NWO counter (Number of Write Operations).

The next section explains how to maintain the counters while the write update mechanism is used. This is followed by an explanation of how to maintain the counters when the write invalidate mechanism is used.

3.1. Maintaining the counters during the execution of the write update mechanism

The sequencer updates the NPU counter to maintain the cumulative communication coherence overhead for the write update mechanism. All remote operations for the write update mechanism are visible to the sequencer and their respective values are given in Table 2. The sequencer adds $S + 2$ to the NPU counter each time the client sends a read request; or adds $(N-1)(P+1)+S+2$ if the client sends a read request with an update request and updating information; or adds $N(P + 1) + 1$ if the client sends an update request and updating information; or adds $N(P + 1)$ if the local sequencer's application process sends an update request and updating information.

For the write invalidate mechanism, a key question is how to estimate the contribution of reads and writes to the NPI counter. It is not straightforward to estimate the cost for the write invalidate mechanism, because not all of the writes will be remote (in contrast to when using the write update mechanism). The second write and all succeeding writes from a single node without intervening accesses from other nodes will be executed locally as was previously explained for the Illinois protocol. To decide which write will be executed locally, some additional communication and calculation is needed. To avoid additional complexity in maintaining the NPI counter, we simply add $S + N + 1$ to the NPI counter for each write operation. This value was chosen because for large N it is the largest among all values for remote writes for the write invalidate mechanism (see Table 2). Updating the NPI counter using this value will predict the upper limit for the cumulative communication cost overhead for the write invalidate mechanism. Results presented in Section 4 show that even with such a simple approximate approach which favors the write update mechanism, our decision function gives good results.

Each time the read operation is remote under the write update mechanism, it is also remote for the write invalidate mechanism. Some of the reads which are performed locally under the write update mechanism would be remote under the invalidate mechanism. As a result, we introduce the NRO counters within each client to count the number of these local reads. Only the first read from a client i after an update request from client j will increment the NRO counter. During each remote operation a copy of the NRO value is sent to the sequencer. It must be determined whether to multiply the received values by $S + 2$ or by $2S + 4$, and then add the value to the NPI counter; i.e. how many of these reads will be satisfied by the sequencer and how many reads will be sent to a remote client. We use the same approach we used for write operations and assume the worst case, so always multiply by $2S + 4$ under assumption that all reads are sent to remote clients.

As an example, assume that data is shared among several clients and that the write update mechanism is used. Also assume that only one client is performing writes while all others are reading the same data. The sequencer can update the NPU and the NPI counters based only on the remote write operations. The contributions due to reads cannot be added to the NPI counter because they are local and not visible to the sequencer. Because the contribution of writes to the NPU counter is greater

than to the NPI counter, the value of the NPU counter exceeds the value of the NPI counter. The mechanism is changed to write invalidate and the sequencer invalidates all copies. Now, the clients send read requests and the values of the NRO counters to the sequencer. The sequencer uses these NRO values to update the NPI counter which now once again has a greater value than the one in the NPU counter. As a result, the mechanism switches back to the write update mechanism. To avoid this unnecessary switch, we introduce a constant MAX_NRO. Whenever the NRO counter exceeds the value of MAX_NRO, the contents of the NRO counter are sent to the sequencer. This additional message does not affect the communication overhead by much, but avoids the unnecessary invalidation of copies of the shared data.

3.2. Maintaining the counters during the execution of the write invalidate mechanism

Maintaining the NPI counter under the write invalidate mechanism is straightforward and incurs no communication cost. Each time the sequencer receives a request, the appropriate cost for the write invalidate mechanism in Table 2 will be added to the NPI counter.

To maintain the NPU counter, the cost of using the write update mechanism must be estimated based on the execution of the write invalidate mechanism. To properly predict the contribution of reads to the NPU counter during the execution of the write invalidate mechanism, the client must send a read request along with an indication of the state of its copy (INVALID or NOTIN state). The sequencer adds $S + 2$ to update the NPU counter only when a read request is sent from a client whose copy is in the NOTIN state. In the write update mechanism, a read is remote only if the copy is not present in the local memory. Invalidations never take place; therefore, reading an INVALID copy cannot occur in the write update mechanism.

All writes under the write update mechanism incur a communication cost which must be added to the NPU counter. Therefore, to properly handle the contribution of writes to the NPU counter during the execution of the write invalidate mechanism, the sequencer must add the appropriate value to the NPU counter each time it receives a request for a write. But, the client's write operations are executed locally under the write invalidate mechanism, and hence not visible to the sequencer, so we introduce the NWO counters in each client to keep track of the number of local executions of the write operation. Each local execution of a write operation increments the NWO counter, and the contents of the NWO counter are sent to the sequencer with each remote operation. Upon receiving the NWO value, the sequencer multiplies it with $N(P + 1) + 1$ and then adds the product to the NPU counter.

As an example, assume that the APCUM protocol is using the write invalidate mechanism and that client i is the only one using the data. Upon the arrival of a read request from the sequencer, the client returns the contents of the NWO counter in the data response message. The sequencer updates the NPU and NPI counters. Based on the new values of the counters, the sequencer makes a decision which mechanism to use next.

3.3. Decision function

The following is the simplest possible decision function: use the write invalidate mechanism when NPI < NPU, otherwise use the write update mechanism. A decision function like this can result in periods where the operation mechanism is changed after each remote operation. To overcome this problem, we use a decision function with hysteresis that changes the mechanism from write update to write invalidate when NPI < NPU, and from write invalidate to write update when NPU < NPI - h, where h is some constant. This policy favors the write invalidate mechanism. We choose this policy because the analysis of real applications has shown that it is more likely that sharing will be sequential rather than concurrent, and for sequential sharing, it is better to use the write invalidate mechanism.

3.4. State transition and operation execution

The state transitions and operation execution necessary to handle both mechanisms in one coherence protocol were described in Section 2.4 where we discussed the modified RWB and the ideal EDWP adaptive protocols. The copy

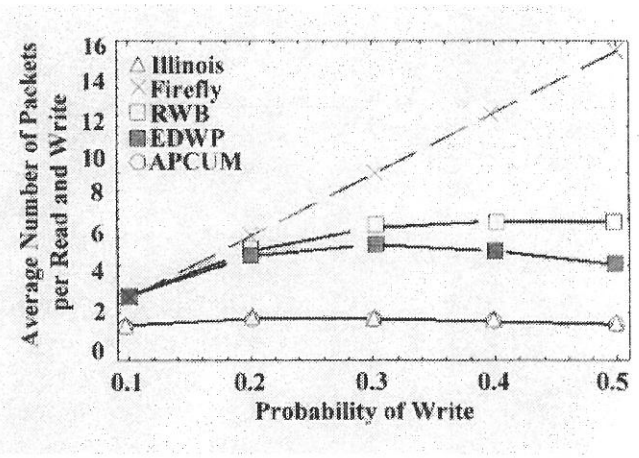

Fig. 1. Average number of packets per operation for sequential sharing data access pattern (Distributed system parameters: $N = 16, S = 4, P = 1$; Workload parameters: Burst size = $N(11, 1)$)

in the client can be in the NOTIN, INVALID, VALID, or DIRTY state whose meanings are the standard ones for coherence protocols. The sequencer's copy can be in the INVALID, VALID, or DIRTY state. Besides the state in the sequencer which indicates the consistency of the copy, there is a state which indicates which mechanism to use to maintain the consistency. The states are changed according to the values of the NPI and the NPU counters as previously described. All requests and responses from the sequencer to the clients during the execution of the remote operations bring information about which of the two mechanisms to use. According to this information the client performs the appropriate state transition and coherence action.

The performance of the proposed APCUM protocol is presented in the next section.

4. Simulation results

To assess the performance of the APCUM protocol, we compare the simulation results with those of other coherence protocols, both adaptive and non-adaptive. The simulations are driven by artificial workloads in order to obtain the results for a variety of values of the data access pattern parameters. We first describe the workload model.

The analysis of real applications shows that nodes perform operations in bursts, i.e. only one node accesses the data at a time. Therefore, we use Dubois & Katz burst model [13, 14] to define the artificial workload. The workload generates the operations in bursts, where the burst size (number of operations in access burst) is determined by a Normal (Gaussian) distribution N(mean, standard deviation). When the node completes a burst of accesses, all $N + 1$ nodes have the same probability of performing the next access burst. The probability of a write operation is also a parameter of the workload.

All results in this paper were obtained through the simulation of a distributed system consisting of 16 clients ($N = 16$). Four packets are used to send data ($S = 4$) while one packet is used to transfer update information ($P = 1$). We measure performance by measuring the average number of packets per operation. We generate three types of data access patterns: sequential sharing, concurrent sharing, and sharing which dynamically changes from sequential to concurrent and vice versa.

4.1. Sequential sharing

Figure 1 shows the results of a simulation using a workload with a high degree of sequential sharing. The burst size is generated as a Normal distribution with a mean equal to 11 operations and a standard deviation equal to 1. Therefore, the average size of the burst, i.e. the number of operations performed by a node without intervention by other nodes, is equal to 11 operations. This characterizes the sequential sharing. The simulation results are presented for different probabilities of write operations.

As one expects, the write invalidate Illinois protocol incurs lower communication cost per operation than the write update Firefly protocol.

With increasing probability of writes, the difference between them becomes bigger. Note that for the Illinois protocol only the first write (and the read operation if it is the first operation in the access burst) incurs a communication cost, while for Firefly all writes in an access burst are remote and incur a communication cost. Therefore, the communication cost for the Illinois protocol does not change much when the probability of a write operation is increased, but the communication cost for the Firefly protocol increases linearly with the probability of a write operation.

The performance of the modified RWB adaptive protocol and the ideal EDWP is very close to the Firefly protocol for small probabilities of write operations. Both of these protocols switch from write update to write invalidate if there are three consecutive write operations from one node. If the probability of a write is small, then the probability that there will be three writes in an access burst is also small. Both protocols will use the write update mechanism most of the time and their performance will be very close to the performance of the Firefly protocol. As the probability of a write increases, it becomes more likely that in a single access burst there will be three writes which will switch the mechanism to write invalidate. Therefore, the communication cost for large probabilities of write operations is better for RWB and EDWP than for the Firefly protocol. The communication cost is higher than for the Illinois protocol because the modified RWB and the ideal EDWP protocols perform two unnecessary updates before an invalidation.

The ideal EDWP protocol incurs a lower communication cost than the modified RWB for higher probability of writes, because it is more likely that the first operation in an access burst will be a write. In this case the ideal EDWP will continue with the write invalidate mechanism (see state transition explanation for EDWP and RWB protocol in Section 2.4), while the modified RWB will switch to the write update mechanism for each new access burst. Sequential sharing with a high probability of a write characterizes migratory data. The detection of migratory data [15, 16], and sending read exclusive as the first request instead of a normal shared read, can improve the performance of the EDWP protocol. The first exclusive read will force the EDWP protocol to use the write invalidate protocol all the time.

The performance of the APCUM protocol is the same as for the Illinois protocol. The cumulative communication cost for write invalidate is lower than for write update. As the access burst starts, NPI < NPU and the APCUM protocol uses the write invalidate mechanism. During most of the access burst, operations are local, so that both NPI and NPU are not changed. As one of the other nodes starts a new access burst, the NPU counter is increased for all local executions of write operations as previously described. This increases the difference between the NPI and the NPU counters and ensures that the APCUM protocol will continue to use the write invalidate mechanism which is better for the given data access pattern. APCUM can switch to write update at the beginning of the simulation, but will later switch to write invalidate for the remainder.

4.2. Concurrent sharing

Figure 2 shows the results of simulation for workloads with a high degree of concurrent sharing. The access burst sizes are much shorter and are generated from a Normal distribution with a mean equal to 1 operation and with a standard deviation equal to 0.16. All values for an access burst size less the 1 are forced to be equal to 1.

For lower probabilities of write operations, Firefly outperforms Illinois, while for larger values, Illinois incurs a lower communication cost. For small probabilities of write operations, it is likely that after a write operation one of the remaining nodes will read the same data after each change. Therefore, it is better to update all copies than to perform invalidation. For large probabilities of a write operation it is better to use write invalidate because the probability that the data will be read by more than one node after each change is very low.

The results for the modified RWB and the ideal EDWP protocols are very close to the Firefly protocol for all values of probability of write operations. These protocols stay with the write update mechanism, because it is unlikely that three successive writes will come from one node. The results for the ideal EDWP are negligibly better

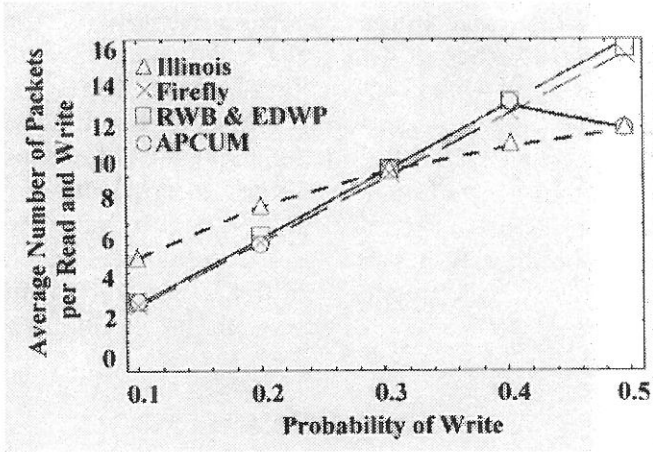

Fig. 2. Average number of packets per operation for concurrent sharing data access pattern (Distributed system parameters: $N = 16, S = 4, P = 1$; Workload parameters: Burst size = $N(1, 0.16)$)

than for the modified RWB and are not presented separately in the figure.

The results for the APCUM protocol show that its performance always follows the better non-adaptive protocol except for the case when the probability of a write is equal to 0.4. The reason that APCUM is worse than Illinois at this point is the method used to estimate the cumulative cost for the write invalidate mechanism. It is estimated by taking the worst case (as discussed in Section 3.3). The estimation of the cumulative cost by more realistic values (we can choose some average numbers for updating the NPI, not the worst case) would give results which more closely follow the best non- adaptive protocol.

4.3. Dynamically changed sharing

Both of the previous workloads are static in the sense that the data access patterns are characterized either as sequential shared or concurrent shared. Figure 3 shows the results for a workload whose type of sharing is changed with respect to time. The access burst size is generated from a Normal distribution with different values of the standard deviation parameter. The mean is fixed and equal to 1 operation. The probability of a write operation is equal to 0.1. Figure 3 presents the results for increasing values of standard deviation. All values of burst size less than 1 are forced to be 1. The left end of the graph shows the results for pure concurrent sharing. Moving toward the right increases the amount of sequential sharing.

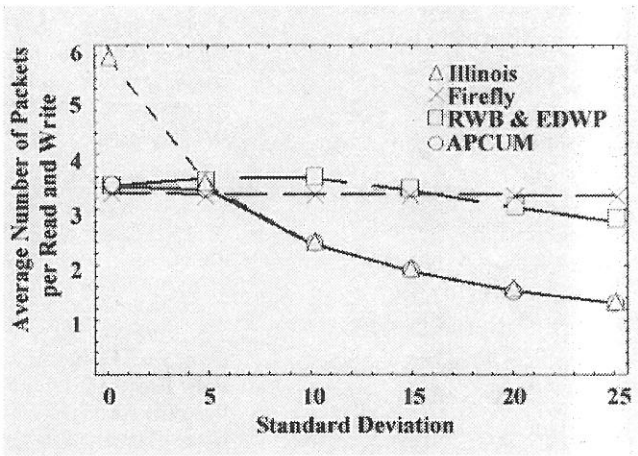

Fig. 3. Average number of packets per operation for dynamically changed sequential and concurrent sharing data access pattern (Distributed system parameters: $N = 16, S = 4, P = 1$; Workload parameters: Probability of write = 0.1, Burst size = $N(1,$ Standard Deviation$)$)

The results at both ends were discussed in the previous two sections. The performance of the APCUM protocol is always as good as the best non-adaptive protocol for a given set of parameters. The modified RWB and the ideal EDWP can only benefit from adaptability if there is a high probability of a large access burst. Then it is more likely that there will be three writes in the access burst, which will enable switching to the write invalidate mechanism.

5. Conclusions

We showed that it is possible to define a decision function for adaptive coherence protocols which for a given data access behavior ensures that the adaptive protocol will have the same performance as the best non-adaptive protocol. In particular, we proposed an adaptive scheme, APCUM, whose decision function gives good results for both sequential and concurrent sharing data accesses. We also showed that previously proposed decision function are not as good as APCUM for both sequential and concurrent sharing, particularly not for sequential sharing. Because previously proposed decision function give good results for concurrent sharing and they work better than pure write update protocols for sequential sharing, they are attractive for page replication [5]. Data access patterns at the page level show a higher percentage of concurrent sharing than for smaller data items such as cache lines. For smaller data sizes there is more likelihood of sequential sharing than there is of concurrent sharing. In such cases the previous solutions do not give good results. The solution proposed in this paper is attractive because it works well with both sequential and concurrent sharing data accesses.

Adaptive protocols can be implemented either in software (typically for distributed systems and Internet) or in hardware (typically for multiprocessor systems). The proposed counters can be easy and efficiently implemented in software. Even though today's cache line sizes (it is normal to have 128 byte cache lines) are large enough to allow the cost of implementing the counters in hardware, in our future work we are planning to use fuzzy set decision function in order to reduce the number of bits required for counters.

Acknowledgments The author thanks members of the Advanced Technology Group of AT&T (San Jose, California), NUMAChine team (Department of Electrical & Computer Engineering, University of Toronto), Department of Electronics, Microelectronics, Computer and Intelligent Systems (Faculty of Electrical Engineering and Computing, University of Zagreb), and in particular the author appreciates the contributions of Hrvoje Bunjevac.

References

[1] J. R. GOODMAN, "Using Cache Memory to Reduce Processor-Memory Traffic", In Proceedings of the 10th Annual International Symposium on Computer Architecture, Stockholm, Sweden, pages 124–131, 1983.

[2] L. RUDOLPH AND Z. SEGALL, "Dynamic Decentralized Cache Scheme for MIMD Parallel Processors", In Proceedings of the 11th Annual International Symposium on Computer Architecture, Ann Arbor, Michigan, Pages 340–347, June 1984.

[3] A. R. KARLIN, M. S. MANASSE, L. RUDOLPH, AND D. D. SLEATOR, "Competitive Snoopy Caching", In Proceedings of the 27th Annual Symposium on Foundations of Computer Science, pages 244–254, October 1986.

[4] J. ARCHIBALD, "A Cache Coherence approach for Large Multiprocessor Systems", In International Conference on Supercomputing, St. Malo, France, pages 337–345, July 1988.

[5] A. W. WILSON, R. P. LAROWE, AND M. J. TELLER, "Hardware Assist for Distributed Shared Memory", In Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, Pennsylvania, pages 246–255, May 1993.

[6] S. J. EGGERS AND R. H. KATZ, "Evaluating the Performance of Four Snoopy Cache Coherency Protocols", In Proceedings of 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, pages 2–15, 1989.

[7] H. BUNJEVAC, "Adaptive Algorithm for Distributed Shared Memory Management", In Proceedings of 15th International Conference on Information Technology Interfaces, Pula, Croatia, pages 245–250, June 1993.

[8] M. S. PAPAMARCOS AND J. H. PATEL, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", In Proceedings of the 11th Annual International Symposium on Computer Architecture, Ann Arbor, Michigan, pages 348–354, June 1984.

[9] S. SRBLJIĆ, "Modification of Cache Coherence Protocols for the Distributed Environment", Automatika, Vol. 34, No 1–2., pages 29–40, January-April 1993.

[10] S. SRBLJIĆ AND L. BUDIN, "Analytical Performance Evaluation of Data Replication Based Shared Memory Model", In Proceedings of Second IEEE International Symposium on High Performance Distributed Computing, Spokane, Washington, pages 326–335, July 1993.

[11] C. P. THACKER AND L. C. STEWART, "Firefly: A Multiprocessor Workstation", In Proceedings of Second International Conference on Architectural Support for Programing Languages and Operating Systems, Polo Alto, California, pages 164–172, October 1987.

[12] M. STUMM AND S. ZHOU, "Algorithms Implementing Distributed Shared Memory", IEEE Computer, Vol. 23, No. 5, pages 54–64, May 1990.

[13] M. DUBOIS AND J.–C. WANG, "Shared Block Contention in a Cache Coherence Protocol", IEEE Transactions on Computers, Vol. 40, No. 5, pages 640–644, May 1991.

[14] M. DUBOIS AND J.–C. WANG, "Shared Data Contention in a Cache Coherence Protocol", In Proceedings of the 1988 International Conference on Parallel Processing, Vol. I, pages 146–155, August 1988.

[15] A. L. COX AND R. J. FOWLER, "Adaptive Cache Coherency for Detecting Migratory Shared Data", In Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, California, pages 98–108, May 1993.

[16] P. STENSTROM, M. BRORSSON, AND L. SANDBERG, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing", In Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, California, pages 109–118, May 1993.

Received: September, 1996
Accepted: October, 1996

Contact address:
Siniša Srbljić
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3, 10 000 Zagreb, Croatia
E-mail: sinisa@zemris.fer.hr

SINIŠA SRBLJIĆ received B.S. degree in electrical enginering in 1981, and M.S. and Ph.D. degree in computer engineering in 1985 and 1990, all from the University of Zagreb, Croatia. He is an Assistant Professor at the University of Zagreb, School of Electrical Engineering and Computing. He was visiting the University of Toronto, Canada, from 1993 to 1995 where he worked on the NUMAchine multiprocessor project. As a visiting scientist, he was working with the Advanced Technology Group of AT&T, USA, from 1995 to 1996, on caching of Internet objects in large distributed multimedia systems. His research interests include parallel and distributed computer systems, compiler design, and performance evaluation.