Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory

Junpei Niwa, Tatsushi Inagaki, Takashi Matsumoto, Kei Hiraki
Department of Information Science, Faculty of Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
{niwa, inagaki, tm, hiraki}@is.s.u-tokyo.ac.jp

Abstract

We have proposed an "Asymmetric Distributed Shared Memory: ADSM", that provides users with efficient shared memory model. The ADSM is a hybrid system that needs not only the operating system support but also the compiler support. The ADSM executes a load instruction as the shared-read with the assistance of virtual memory mechanism. As for the shared-write, the ADSM executes a sequence of instructions for consistency management after the corresponding store instruction.

We describe the algorithm to reduce overheads for consistency management. The algorithm coalesces a sequence of instructions for consistency management using the information of affine memory accesses.

The coalescing algorithm is evaluated using the SPLASH-2 benchmark. The performance evaluation shows that the coalescing algorithm achieves the execution time improvement compared to the not-optimized result, ranging from 76% to 85% .

1 Introduction

A shared memory model can reduce the cost of programming effort in distributed systems. The main reason is that shared data are easily accessed with a single address space, that is to say, the programmers take no account of the location of data.

On distributed systems such as networks of computers, it is necessary to provide the shared memory model by software. The main reason is that the hardware approach to implement the shared memory model requires high cost.

We have proposed a software shared memory model, Asymmetric Distributed Shared Memory (ADSM)[6]. In the ADSM, the shared-read is executed as a single load instruction, using the virtual memory mechanism. The shared-write is executed as the corresponding store instruction and the instructions for consistency management inserted by the compiler.

In this paper, we propose an algorithm to coalesce a sequence of instructions for consistency management. It is a loop-level algorithm that finds affine memory accesses using the information of the induction variable and the loop-invariant variable.

To evaluate the performance of the algorithm, we have implemented three protocols: LRC, SAURC (Software emulated AURC) and HYBRID(hybrid of LRC and SAURC). We have implemented a prototype of the compiler and the runtime system for the ADSM on a multicomputer Fujitsu AP1000+. We report that overall execution times, as well as detailed breakdowns of elapsed times, the number of instructions for consistency management and the number of messages.

2 Features of the ADSM

The ADSM is an all-software, page-based shared memory system that realizes user-level protected high-speed high-functional communications /synchronizations. The ADSM requires no hardware support like AURC[1]. It can be implemented on workstation clusters or multicomputers with conventional network interfaces.

In the ADSM, the action to the shared-read is different from that to the shared-write.

- Shared-read:
  The shared-read is based on a cache-based shared virtual memory system[4]. The shared-read is executed as a load instruction from the shared page. Only when the processor reads from the shared page that is not allocated or invalid, instructions for consistency management are invoked by the read trap routine.

- Shared-write:
  The shared-write is realized as consistency management instructions (such as additional memory management and remote memory access). The compiler translates the shared write into a sequence of instructions. That is to say, instructions for consistency management are explicitly inserted after the corresponding store instruction by the compiler.

Since instructions for consistency management are inserted by the compiler, we can perform following optimizations:

1. The compiler can generate the instructions for consistency management according to the consistency protocol. That is to say, the ADSM supports consistency protocol selection per page[5].

2. When a sequence of the shared write is performed to the contiguous location and there are no synchronization points, the sequence of corresponding instructions for consistency management is coalesced. The coalescing optimization reduces the runtime overhead.

3. When there are communication packets whose destination are the same node, the number of communication is reduced, combining the packets by the compiler as well as the operating system. The combining optimization reduces the overhead associated with update memory accesses.

3 Protocol Implementation

In order to detect and collect updated data efficiently, we use the *write history* which is represented as the the tuple of the store address and the store size.

3.1 LRC Protocol

Our protocol is similar to the Lazy Release Consistency(LRC)[2] in the TreadMarks[3].

On an acquire point, the requesting processor invalidates all pages according to the *write-notice*[2]. The page fault handler obtains only the updated data rather than the whole page. The updated data cannot be discarded as long as there are nodes that may need them, which causes a severe memory consumption problem. Garbage collection must be performed frequently.

The instructions for consistency management record the corresponding *write history*. Our protocol computes the updated data using the recorded *write history*. The computation of the updated data is proportional to the amount of the created *write history*.

3.2 Software emulated AURC(SAURC) Protocol

SAURC is a protocol which emulates Automatic Update Release Consistency(AURC) protocol[1] without special hardware support. The instructions for consistency management record the corresponding write history and the corresponding data(we call them update information). The update information is not propagated to the home for each shared-write. In order to reduce the number of messages, a series of update information is combined when the home node is the same. After the home is updated using the update information, it is discarded.

3.3 HYBRID Protocol

In LRC protocol, more than one remote processor may have to be visited in order to obtain updated data at the page fault. In AURC protocol, the processor has only to visit the home at the page fault. But the whole page must be fetched from the home. We propose a hybrid protocol of LRC and SAURC in order to solve these problems.

Although HYBRID protocol is almost the same to SAURC, the home does not discard the propagated update information as well as SAURC but records them. At the page fault, the processor has only to visit the home. The home sends to the faulting processor only updated data using the recorded update information.

4 Reducing the overhead of the shared write

We use an interprocedural pointer analysis[9] for detecting the write to the shared locations. The pointer analysis is a context-sensitive algorithm keeping track of which pointers are actually referenced by a procedure. By the pointer analysis, the compiler detects the variable which contains the shared location and inserts the instructions for consistency management after the corresponding store.

When a sequence of the shared-write is performed to the contiguous location and the program does not reach to the synchronization points, the corresponding sequence of instructions for consistency management is coalesced.

Each loop has a *depth*. It is 1 when the loop is outer-most and it is n when the loop is inner-most of the n-th loopnest. We represent instructions for consistency management as a tuple $I(A, S)$. A denotes the corresponding memory location and S denotes the corresponding size.

From the program, we enumerate the n-th nested loopnest that contains the shared write which is always executed. For each loopnest,

1. We insert instructions for consistency management after the corresponding store.

2. $depth := n$.

3. If $depth = 0$ then end.

4. Take a loop L whose level is *depth*.

   (a) In the loop L, when there is a series of instructions for consistency management whose location is contiguous or the same, the series of instructions for consistency management is coalesced.

   (b) When the synchronization code or the return code exists in the loop L or inner-loops, goto 4;

   (c) For each instructions for consistency management represented as $I(A, S)$.

      i. When A is loop-invariant, the instructions for consistency management are hoisted out from the L to the outerloop of L.

      ii. When A is an induction variable and the variable's stride(A_{stride}) is equal to or smaller than S, the instructions for consistency management are hoisted from the L to the outerloop of L. Let c be the loop count and A_{low} be the lower bound of the A. The hoisted instructions for consistency management are represented as $I\left(A_{low}, (c-1) * A_{stride} + S\right)$.

5. $depth := depth - 1$, and goto 4.

Here is the example code.

```
for (i = 0; i < n; i = i + 1) {
    a[i] = a[i] + alpha * b[i];
    I (&a[i], sizeof (double));
}
```

⇓ Coalescing Optimization

```
for (i = 0; i < n; i = i + 1) {
    a[i] = a[i] + alpha * b[i];
}
I (&a[0], n * sizeof (double));
```

5 Performance Evaluation

We have implemented a prototype of the compiler and the runtime system of the ADSM on a multicomputer Fujitsu AP1000+[8].

Each node consists of 50MHz Super SPARC with 20 Kbytes I-cache, 16 Kbytes D-cache and 16MB memory. Nodes are interconnected by a 2-D torus network whose bandwidth is 25 Mbytes/sec per link.

Because the CellOS of the AP1000+ does not support user-level page fault handler, we cannot detect page fault via interrupts. In order to detect that the processor attempts to access the shared page that is not allocated or invalid, we insert the code checking the corresponding page's validity before each shared access.

Because it is impossible to handle messages via interrupts on the AP1000+, request messages from remote nodes are serviced through polling mechanism. We insert polls at every loop backedge and every function call.

We evaluate the performance on LU-Contig of SPLASH-2[10] using 8 nodes. The problem size is a 256×256 matrix with 16×16 blocks. Table1 shows the results of LU-Contig. We evaluate the optimization method of coalescing the instructions for consistency management.

For each table, LRC(*) shows the results when LRC is selected. SAURC(*) shows the results when SAURC is selected. HYBRID(*) shows the results when HYBRID is selected. *(NO) shows the results when the compiler does not perform an optimization(coalescing the instructions for consistency management). *(O) shows the results when the compiler performs the optimization. The "task" shows the computing time. The "message" shows the time spent in handling remote requests by polling and also shows the ratio of the number of messages handled by polling to the total message number. The "PF" shows the page fault time and also shows the number of the page fault. The "CM" shows the consistency managing time and also shows the number of instructions for consistency management. The "sync"shows the synchronization time that includes lock/unlock and barrier. The "GC" shows the time spent in garbage collection and also shows the number of garbage collection in LRC.

For each protocol, the optimization achieves the execution time improvement in comparison to the not-optimized result, ranging from 76% to 85%. When the number of instructions for consistency management is reduced, "CM" time is reduced and the amount of the created *write history* is also reduced. Many operations executed in "PF" time,"sync" time and "message" time are depend on the amount of the created *write history*. Therefore when the number of instructions for consistency management is reduced, "message" time, "PF" time "CM" time and "sync" time are all reduced. "task" time is also reduced because the time to call the function which perform instructions for consistency management is added to the "task" time.

Because LU-Contig is an application that the processor accesses the contiguous locations, there are many opportunities to coalesce a sequence of instructions for consistency management. In our implementation on the AP1000+, SAURC is best consistency protocol.

6 Related Work

6.1 OS-based Software DSM

A number of software page-based shared memory models have been proposed[4, 2, 1].

Writing to the shared location is executed as a single store instruction. At this point, the ADSM is different from the existing OS-based Software DSM. The consistency management instructions are executed in the write trap routine. In order to execute consistency management instructions, the write trap routine must be invoked[4]. In LRC, in order to minimize the number of trap invocations, techniques such as diff scheme which computes the page's difference between the synchronization points have been proposed[2]. But this diff scheme can be expensive because the whole page is searched independently of the amount of updated data.

AURC is LRC supported not by using diff scheme but by using automatic update hardware[1]. Automatic update is a mechanism that automatically propagates updates at fine granularity when a mapping has been established between the local written page and a remote page. AURC has a *home* for each shared page. By automatic update hardware, the shared writes are propagated to the home without software overhead. The home is always kept up-to-date. The consistency of other copies is managed like LRC invalidation scheme. When the page fault occurs, fetching the whole page from the home is only required.

6.2 Compiler-based Software DSM

In the Midway which implements Entry Consistency (EC)[11], each shared data is bound to the synchronization variable. On a lock acquire point, EC only propagates the updated data associated with that lock variable.

The compiler detects the shared-write. Every shared address has a *software dirty bit* that reflects whether or not the address has been written. After each write to shared memory, the compiler inserts instructions which set a software dirty bit with that address. No optimization is performed. Searching the software dirty bit region, the processor determines the updated data.

In Shasta[7], by using instrumentation, the compiler inserts the instructions for consistency manage-

Table 1: LU-Contig(256×256 matrix) time in seconds

protocol	total	task	message	PF	CM	sync	GC
LRC(NO)	81.89	1.87	0.61	2.27	3.42	3.17	72.57
			88/2720	104	699040		11
LRC(O)	11.95	0.63	0.76	3.33	0.24	2.60	4.38
			178/790	232	49640		1
SAURC(NO)	13.56	1.82	1.14	0.83	4.92	4.86	0.00
			262/437	210	699040		0
SAURC(O)	3.12	0.62	0.17	0.29	0.56	1.49	0.00
			217/392	210	49640		0
HYBRID(NO)	27.47	1.83	4.03	4.28	5.99	11.34	0.00
			258/440	204	699040		0
HYBRID(O)	4.11	0.62	0.39	0.51	0.71	1.87	0.00
			212/356	210	49640		0

ment, which implements an Eager Release Consistency. It performs various optimization(e.g. batching miss checks), but it does not perform optimizing methods of coalescing a sequence of instructions for consistency management in the loopnest.

7 Conclusion

We describe the optimization methods on the ADSM: (1)protocol selection and (2)reduction of the overheads for consistency management.

We have implemented three consistency protocols:(1)LRC protocol, (2)SAURC protocol and (3) HYBRID protocol. Our compiler generates the instructions for consistency management according to the consistency protocol.

We introduce an algorithm to coalesce a sequence of instructions for consistency management in the loopnest with the assistance of the affine memory accesses.

Our experiments with SPLASH-2 benchmark on a multicomputer Fujitsu AP1000+ shows that our optimization method of reducing overheads for consistency management is effective.

References

[1] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proc. of the 2nd Inter. Symp. on HPCA*, February 1996.

[2] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th ISCA*, pages 13–21, May 1992.

[3] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[4] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 ICPP*, pages 94–101, August 1988.

[5] Takashi Matsumoto. Fine Grain Support Mechnisms. In *IPSJ Computer Architecture SIG Notes*, volume 89-ARC-77, pages 91–98, July 1989. (in Japanese).

[6] Takashi Matsumoto, Taketo Komaarashi, Shigeru Uzuhara, Shozo Takeoka, and Kei Hiraki. A General-Purpose Massively-Parallel Operating System: SSS-CORE – Implementation Methods for Network of Workstations –. In *IPSJ Operating System SIG Notes*, volume 96-OS-73, pages 115–120, August 1996. (in Japanese).

[7] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–185, October 1996.

[8] Osamu Shiraki, Yoichi Koyanagi, Nobutaka Imamura, Kenichi Hayashi, Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Architecture of highly parallel computer ap1000+. In *Third Parallel Computing Workshop*, pages P1–G–1–P1–G–8, nov 1991.

[9] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Prgrams. In *Proc. of '95 Conf. on PLDI*, pages 1–12, June 1995.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd ISCA*, pages 24–36, June 1995.

[11] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proc. of the 1st Symp. on OSDI*, pages 87–100, November 1994.