Adaptive Prefetching Technique for Shared Virtual Memory

Sang-Kwon Lee Hee-Chul Yun Joonwon Lee Seungryoul Maeng

Computer Architecture Laboratory
Korea Advanced Institute of Science and Technology
373-1 Kusong-Dong, Yusong-Gu, Taejon Korea 305-701
{sklee,hcyun,joon,maeng}@camars.kaist.ac.kr

Abstract

Though shared virtual memory (SVM) systems promise low cost solutions for high performance computing, they suffer from long memory latencies. These latencies are usually caused by repetitive invalidations on shared data. Since shared data are accessed through synchronizations and the patterns by which threads synchronizes are repetitive, a prefetching scheme based on such repetitiveness would reduce memory latencies. Based on this observation, we propose a prefetching technique which predicts future access behavior by analyzing access history per synchronization variable. Our technique was evaluated on an 8-node SVM system using the SPLASH-2 benchmark. The results show that our technique could achieve 34% – 45% reduction in memory access latencies.

1 Introduction

Research on cluster systems such as NOW (Network of Workstations) has been fueled by the availability of powerful microprocessors and high-speed networks. SVM (Shared Virtual Memory) systems provide shared memory abstraction between machines using virtual memory hardware [8, 2]. But, long remote memory access latencies have been a major obstacle to performance improvement of SVM systems.

To reduce memory latencies, most SVM systems use the following approaches: (1) to convert remote accesses into local ones by caching, (2) to delay communications using relaxed memory model such as LRC (Lazy Release Consistency) [7], and (3) to reduce effects of false sharing using multiple-writer protocol. But, remaining latencies are still significant.

Since shared data are accessed through synchronizations and the patterns by which threads synchronizes are repetitive, a prefetching scheme based on such repetitiveness would reduce memory latencies. Prefetching can be used to reduce these latencies by fetching data in advance before actual data accesses.

Based on this observation, we propose a prefetching technique in which the data invalidated after a repetitive synchronization pattern are prefetched at proper times. Our prefetching technique analyzes past data access history per synchronization variable, then adaptively chooses between history mode and stride mode prefetching. If it is found that there is no particular access pattern, we don't issue prefetches.

To evaluate our technique, we ran four applications from SPLASH-2 using KDSM (KAIST Distributed Shared Memory) on 8-PC Linux cluster. The results show that our technique could achieve 34% – 45% reduction in memory access latencies for three applications. By comparing with other techniques, we illustrate that considering synchronization variables makes prediction more accurate.

The rest of this paper is organized as follows. In section 2, experimental methodology and preliminary results are given. In section 3, we discuss related work. In section 4, we describe our prefetching technique which can effectively reduce memory access latencies. In section 5, we report experimental results. Finally, conclusions are given in section 6.

2 Experimental Methodology and Preliminary Results

To examine impact of remote memory access latencies on SVM performance, we performed experiments on 8-PC Linux cluster which are connected by 100 Mbps Switched Fast Ethernet. Each PC contains 500 MHz Pentium III CPU and 256 MB main memory.

[0]This research is supported by KISTEP under the National Research Laboratory program.

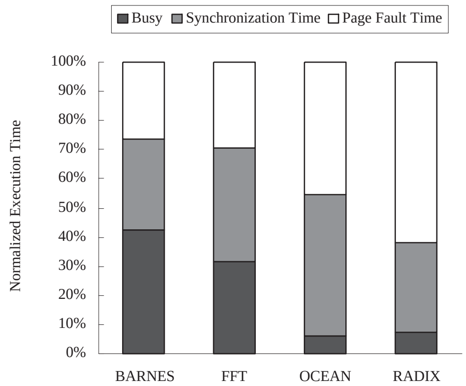

Figure 1. Execution time breakdown on top of KDSM on 8 processors

2.1 KDSM: KAIST Distributed Shared Memory

KDSM is a full-blown SVM system which has been implemented from scratch. KDSM is implemented as a user-level library running on Linux 2.2.13. Communication layer is TCP/IP and SIGIO signal handling is used for processing messages from other processes. KDSM uses page-based invalidation protocol, multiple-writer protocol, and supports HLRC (Home-based Lazy Release Consistency) [11]. Cache pages can be one of four states: RO (read-only), RW (read-write), INV (invalid), and UNMAP (unmapped). All experiments were performed using round-robin home allocation. Average basic operation costs of KDSM are as follow: 1047μs for fetching a 4KB page, 259μs for acquiring a lock, and 1132μs for barrier (8 processors).

2.2 Applications

We ran four applications from SPLASH-2 [10]. BARNES implements the BARNES-Hut method to simulate the interaction of a system of 4K bodies. FFT performs a 3-D complex Fast Fourier Transform on 256K data points. OCEAN simulates large-scale ocean movements based on eddy and boundary currents. We simulate a 258 $\times$ 258 ocean grid. RADIX performs an integer radix sort with 2^{20} keys. BARNES and FFT are restructured versions from CVM [6] distribution and exhibit highly regular page access pattern. OCEAN and RADIX are directly ported to KDSM from SPLASH-2 and not restructured. RADIX exhibits highly irregular pattern and OCEAN exhibits mixed pattern.

2.3 Preliminary Results

Figure 1 shows an execution time breakdown of applications on top of KDSM on 8 processors. Execution time consists of the following components: (1) time spent doing useful computation (2) synchronization (lock and barrier) (3) time spent handling page faults (this includes both local and remote faults, but almost all the time is spent handling remote ones). From figure 1, we can see that most applications spend much time (about 26% – 62%) handling page faults. Prefetching can be used to reduce these remote access overheads.

3 Related Work

Bianchini et al. [3] proposed the B+ technique which issues prefetches for all the invalidated pages at synchronization points.

Amza et al. [1] proposed a technique which dynamically aggregates pages into larger page group. On a page fault, all pages in a page group are fetched at the same time. This has a similar effect as prefetching.

Karlsson et al. [5] proposed a history prefetching technique that exploits producer-consumer access pattern. If no access pattern is detected, a sequential prefetching is used. This technique is a limited one in that it can detect only producer-consumer pattern. It is possible for this technique to issue useless prefetches for irregular applications because it always tries to issue prefetches.

Bianchini et al. [4] proposed the Adaptive++ technique which adaptively chooses repeated-phase mode and repeated-stride mode. Adaptive++ has a benefit over Karlsson's in that (1) it is not limited to producer-consumer pattern and (2) it does not issue prefetches when there is no particular pattern.

Mowry et al. [9] studied prefetching effectiveness when prefetching codes are inserted into program source by compiler and programmer. Since this technique analyzes source code, it can predict future access behavior with high accuracy even in a case of irregular application. This study presents the maximum performance improvements which can be achieved by prefetching.

4 Adaptive Prefetching Technique

4.1 Prefetching Heuristics

Our prefetching technique analyzes past data access history per synchronization variable, and then adaptively chooses between history mode and stride mode prefetching. If it is found that there is no particular access pattern, we don't issue prefetches. Although this technique can be applied to both barrier and lock variables, we only deal with barrier variables in this paper. Before we continue to explain our prefetching technique, we show a simple example which illustrates why we should consider synchronization variables.

4.1.1 A Simple Parallel Program Example

Figure 2 shows a simple parallel program of an iterative algorithm. This program consists of three parts which are protected by three barrier variables. Figure 3 shows run-time execution behavior of this program. If we examine page access history of each part which is protected by the same barrier variable, it is evident that almost identical access pattern repeats as `for` loop goes (see figure 4).

```
int i;
for (i = 0; i < iterations; i++)
{
  barrier(1);
  compute_a();
  barrier(2);
  compute_b();
  barrier(3);
  compute_c();
}
```

Figure 2. A parallel program example

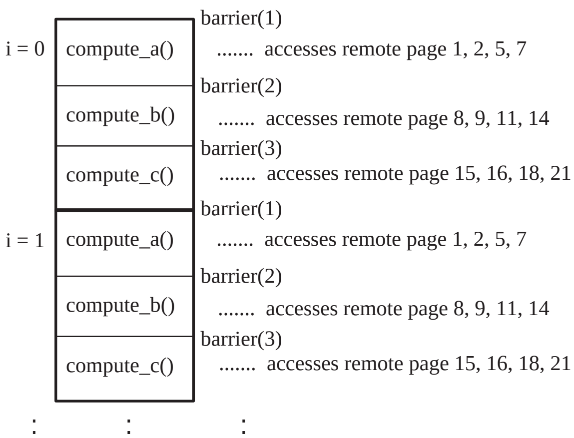

Figure 3. Run-time execution pattern

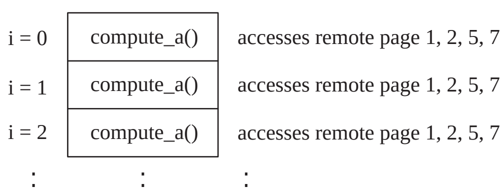

Figure 4. Access pattern of a program part which is protected by barrier variable 1

4.1.2 History Mode Prefetching

History mode prefetching works as follows: if access patterns of last two barrier phases which are protected by the same barrier variable are similar to each other, this mode predicts that future accesses are going to be similar to these patterns.

To record access history per barrier variable, we maintain three lists for each barrier variable: `last`, `before_last`, and `expected`. In a case of barrier variable `bar1`, `last` contains access history at the last barrier phase of `bar1`'s and `before_last` contains access history at the phase before the last of `bar1`'s. Each list contains the ids of faulted pages in the order of fault. But, pages faulted within critical section protected by lock are not included in these lists. Similarity of two lists is calculated like this: (|| denotes number of elements in a list):

$$similarity = \frac{|last \cap before\_last|}{|last| + |before\_last| - |last \cap before\_last|}$$

If similarity is more than 50%, we can say that they are similar to each other. We save the ids of common pages (`last` ∩ `before_last`) included in both `last` and `before_last` to `expected`, which will be used for prefetching.

4.1.3 Stride Mode Prefetching

As history mode does, stride mode prefetching analyzes access history per barrier variable. At the beginning of barrier, we sort `last` and store it into `temp` list. We calculate a frequency of the stride which occurs most frequently in `temp`. If the frequency is more than 50%, this mode predicts that future accesses are going to be separated by the same stride.

4.1.4 Choosing a Prefetching Mode

Choosing a prefetching mode is decided like this: at the beginning of a barrier, similarity and frequency are independently calculated as explained above. Comparing similarity and frequency, we choose the one which has a larger value. In a case of tie, we choose the history mode. But, we issue prefetches only when the chosen value is larger than 50%.

4.2 Implementation Details

4.2.1 Maintaining Access History per Barrier Variable

To record page access history per barrier variable, we maintain one more list `current`, which contains the ids of pages accessed in current barrier phase (let's call the current barrier variable `bar1`) in the order of fault. Cases when a page fault occurs during execution of an application are one of the followings: 1. write to a RO local (home) page, 2. read/write to an INV remote page, or 3. write to a RO remote page. Since access pattern of an application has no relation to home allocation, we must consider not only remote pages but also local pages in calculating a stride. So, we append the page id to `current` for both case 1 and 2.

When current phase of barrier `bar1` ends, all lists related to `bar1` are restructured appropriately.

4.2.2 When and How To Issue Prefetches

When the history mode is chosen, prefetches are issued through two steps. (1) At the beginning of a barrier, we issue prefetches for the first 25 pages in `expected` and delete them from the list. While other techniques usually don't wait for replies after issuing prefetches, our implementation does. Because prefetch replies consecutively arrive right after the last prefetch is issued, there is almost no overlapping between computation and communication. (2) Whenever a remote page fault occurs (except for occurring inside critical sections), we issue prefetches for the first 5 pages in `expected`, delete them from the list, and wait for replies.

When the stride mode is chosen, prefetches are only issued at the time of page fault. When a remote page fault occurs (except for occurring inside critical sections), we examine 5 pages following the page separated by the stride value. We issue prefetches only for pages which have a valid shared memory address and are in an INV state (this means that it has ever been mapped).

Cache state of prefetched pages is not set to RO but to INV for the purpose of checking that pages are really accessed. If a fault on prefetched page occurs later, we change the state from INV to RO/RW according to the fault class.

4.3 Comparison with the Adaptive++ Technique

Adaptive++ [4] technique is very similar to our technique in many respects. But, Adaptive++ assumes that most applications exhibit consecutive or alternating pattern and it pays no attention to barrier variable in analyzing access history. Therefore, Adaptive++ fails to predict future accesses for an application that uses more than two barrier variables within a loop (for example, figure 2), though the application exhibits very regular pattern.

i = 0	0(H), 2(R), 3(R), 4(R), 5(R), 6(R), ...	barrier(13)
	16(H), 24(H), 32(H), 40(H), 48(H), ...	barrier(15)
i = 1	0(H), 2(R), 3(R), 4(R), 5(R), 6(R), ...	barrier(13)
	16(H), 24(H), 32(H), 40(H), 48(H), ...	barrier(15)
i = 2	0(H), 2(R), 3(R), 4(R), 5(R), 6(R), ...	barrier(13)
	16(H), 24(H), 32(H), 40(H), 48(H), ...	barrier(15)
⋮	⋮	⋮

Figure 5. Another case that Adaptive++ fails to predict regular alternating pattern

Adaptive++ even fails to predict regular alternating pattern like figure 5 (actually, this pattern is from BARNES). The number denotes a page id and the character in parenthesis denotes whether it is a local fault (H) or remote (R). Adaptive++ fails to predict access pattern of barrier(13) because it chooses repeated-stride mode not repeated-phase mode at the beginning of barrier(13). This in turn stems from the fact that usefulness is 0 (since every fault in previous phase is local, there is no useful prefetches) and frequency of stride value 8 is more than 50%.

5 Experimental Results

To evaluate performance of prefetching techniques on the same platform, we implemented B+, Adaptive++, and our technique on KDSM. A little attention must be paid for the following details: (1) while B+ and Adaptive++ are proposed issuing prefetches for diffs under the LRC protocol, our technique issues prefetches for a whole page under the HLRC protocol. To perform experiments under the same condition, all techniques were implemented as prefetching a whole page. (2) B+ and Adaptive++ don't wait for prefetch replies. But there is almost no overlapping between computation and communication, as explained in section 4.2. So, we implemented such that all techniques wait for replies.

5.1 Prefetching Effectiveness

To evaluate effectiveness of prefetching techniques, we examine prefetching coverage and hit ratio. *Coverage* is defined as the percentage of page faults which are eliminated by prefetching pages in advance. *Hit ratio* is defined as the percentage of valid prefetches among total prefetches. A valid prefetch is a prefetch by which a page fault is successfully eliminated. Table 1 shows coverage and hit ratio of three prefetching techniques (B+ denotes B+ technique, A++ denotes Adaptive++ technique, and New denotes our technique). For both Adaptive++ and our technique, coverage and hit ratio of individual mode are also shown. Coverage and hit ratio are calculated as follow:

$$Coverage = \frac{Valid\ Prefetches}{Total\ Access\ Misses}$$

$$Hit\ Ratio = \frac{Valid\ Prefetches}{Tatal\ Prefetches}$$

B+ has best coverage among three techniques for BARNES, FFT, and RADIX. B+ has good hit ratio for BARNES and FFT, but bad for OCEAN and RADIX. Low hit ratio of RADIX is due to its irregular access pattern.

Adaptive++ has lowest coverage among three techniques. From table 1, we can see that most prefetches were issued by repeated-stride mode. This implies that repeated-phase mode fails to predict future access pattern. Though

Appl.	Pref. Tech.	Total Access Misses	Overall				History/Repeated-Phase Mode				Stride/Repeated-Stride Mode			
			Total Pref.	Valid Pref.	Covg.	Hit Ratio	Total Pref.	Valid Pref.	Covg.	Hit Ratio	Total Pref.	Valid Pref.	Covg.	Hit Ratio
BARNES	B+	20084	19497	18111	90.18	92.89								
	A++	20084	1414	1390	6.92	98.30	0	0	0.00	0.00	1414	1390	6.92	98.30
	New	20084	14558	14539	72.39	99.87	13309	13290	66.17	99.86	1249	1249	6.22	100.00
FFT	B+	19433	15960	15163	78.03	95.01								
	A++	19433	10335	10164	52.30	98.35	1711	1711	8.80	100.00	8624	8453	43.50	98.02
	New	19433	12266	12260	63.09	99.95	12168	12168	62.62	100.00	98	92	0.47	93.88
OCEAN	B+	29501	23827	11728	39.75	49.22								
	A++	29501	8806	8216	27.85	93.30	579	560	1.90	96.72	8227	7656	25.95	93.06
	New	29501	21881	19510	66.13	89.16	14587	12993	44.04	89.07	7294	6517	22.09	89.35
RADIX	B+	20156	18094	1317	6.53	7.28								
	A++	20156	21	20	0.10	95.24	0	0	0.00	0.00	21	20	0.10	95.24
	New	20156	24	23	0.11	95.83	0	0	0.00	0.00	24	23	0.11	95.83

Table 1. Prefetch Coverage and Hit Ratio

Appl.	Number of Messages				Message Size (KBytes)			
	Base	B+	A++	New	Base	B+	A++	New
BARNES	43005	45777	43053	43043	92647	98332	92745	92725
FFT	78290	79884	78632	78302	236299	239568	237001	236324
OCEAN	88488	110136	89146	93116	243096	287566	244436	252625
RADIX	46702	80261	46705	46699	112257	181073	112257	112241

Table 2. Network Traffic

BARNES has a very regular access pattern, its coverage is very low because it fails to predict future access pattern (see section 4.3). Low coverage of RADIX comes from a different reason that RADIX is very irregular. It will be helpful not to issue prefetches for such an irregular application. Hit ratio of Adaptive++ is more than 90% for all applications.

Coverage of our technique is best for OCEAN and second for BARNES and FFT. Our technique issues small number of prefetches for RADIX for the same reason as Adaptive++. For BARNES and FFT, more than 90% of total prefetches were issued by history mode and hit ratio of history mode is nearly 100%. If we analyze past access history per barrier variables, regular applications would exhibit similar access pattern at phases under the same barrier variable and, therefore, there is much possibility for the history mode to be chosen. This explains why our technique chooses history mode more frequently than stride mode and why the hit ratio of history mode is so high.

5.2 Network Traffic

Table 2 shows the total number of messages and message size sent through network for the base KDSM and for cases prefetching used. Overall, prefetching techniques increase message count and size for all applications. In particular, B+ greatly increases message count and size for OCEAN and RADIX as a result of many useless prefetches.

Prefeting Technique	Applications			
	BARNES	FFT	OCEAN	RADIX
B+	86.0	71.5	33.5	3.3
A++	3.0	22.2	15.1	0.0
New	41.1	34.1	45.4	0.0

Table 3. Percentage of Reduction in Remote Memory Latencies

5.3 Overall Performance

Figure 6 shows execution time breakdown of applications normalized to base KDSM. Prefetching overhead is time spent preparing for prefetching (e.g. maintaining access history and deciding prefetching mode). From figure 6, we can see that it is very small for all techniques. Overall, prefetching reduces time spent handling page faults except for RADIX. Table 3 shows reduction in remote memory latencies.

While B+ has best performance improvement for BARNES (18%) and FFT (11%) it has worst performance degradation for OCEAN (-17%) and RADIX (-13%). From this, we can think that effectiveness of B+ is largely dependent on whether invalidations can guide future access behavior or not. If invalidations succeed to predict future data accesses, B+ can improve application performance greatly. If invalidations fail to do, B+ causes severe performance

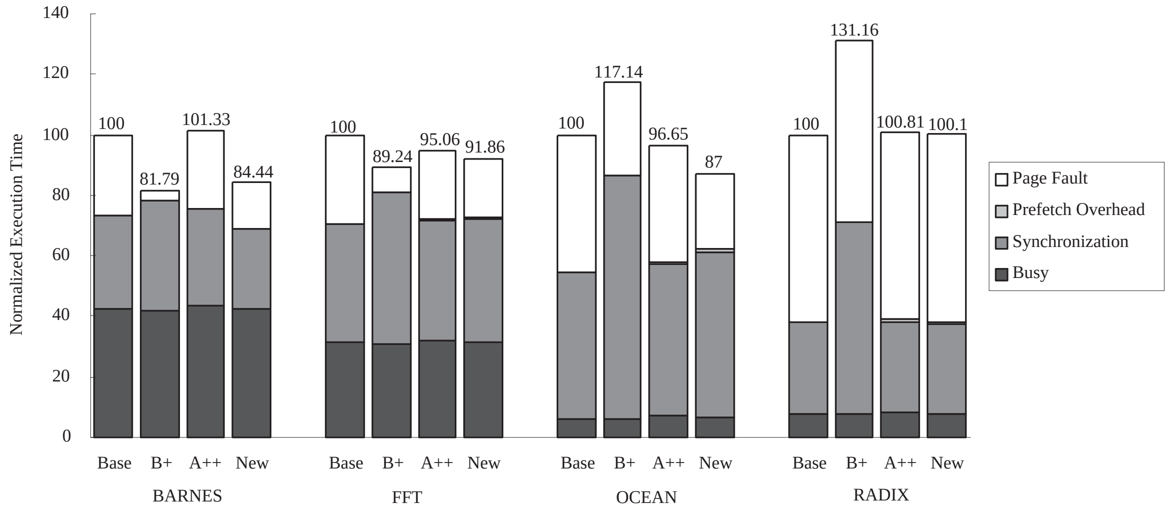

Figure 6. Normalized Execution Time on 8 Processors

degradation as a result of issuing many useless prefetches. The reason why synchronization time increases for all applications is that B+ issues all the prefetches at synchronization points and waits for replies.

While Adaptive++ has performance improvement for OCEAN (3%) and FFT (5%), it has a little performance degradation for BARNES and RADIX (-1% for both). As we saw previously, the coverage of BARNES and RADIX is so low that we can't expect any performance improvement by prefetching.

Our technique has performance improvement for BARNES, OCEAN, and FFT as much as 16%, 13%, and 8% respectively, and no performance improvement for RADIX. High coverage and hit ratio of BARNES, OCEAN, and FFT well explain this improvements.

6 Conclusions

In this paper, we proposed a prefetching technique which predicts future access behavior by analyzing access history of individual barrier variable. Experimental results showed that considering synchronization variable makes prediction more accurate. Our technique could achieve significant reduction in memory access latencies. We believe that our technique is effective in reducing long memory latencies of SVM systems.

References

[1] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the 6th PPoPP*, June 1997.

[2] C. Amza, S. Dwarkadas, P. Keleher, A. L. Cox, and Z. Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), February 1996.

[3] R. Bianchini, L. I. Kontohanassis, R. Pinto, M. Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th ASPLOS*, October 1996.

[4] R. Bianchini, R. Pinto, and C. L. Amorim. Data Prefetching for Software DSMs. In *Proceedings of the International Conference on Supercomputing*, July 1998.

[5] M. Karlsson and P. Stenstrom. Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems. *Journal of Parallel and Distributed Computing*, 43(7), July 1997.

[6] P. Keleher. CVM: The Coherent Virtual Machine. Technical report, 1996.

[7] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th ISCA*, MAY 1992.

[8] K. LI and P. Hudak. Memory Coherence in Shared Memory Systems. *ACM Transaction on Computer Systems*, 7(4), November 1989.

[9] T. Mowry, C. Q. C. Chan, and A. K. W. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *Proceedings of the 4th HPCA*, Feburary 1998.

[10] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd ISCA*, May 1995.

[11] Y. Zhou, L. Iftode, and K. Li. Perforamnce Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of USENIX OSDI*, October 1996.