Evaluating Dynamic Prefetching in a Software DSM

Xinghua An Ting Liu
Carnegie Mellon University
Pittsburgh, PA 15213
{anxh, tingliu}@cs.cmu.edu

Abstract

While software-controlled static prefetching in software DSMs appears to achieve great performance on many applications, the results in investigating prefetching with runtime information are not satisfied due to the formidable overhead. Seeking specific prefetching mechanisms to reduce the execution time of a certain kind of applications, however, is a promising work and will impact on SVMs and other fields. In this report, an experimental work, adding dynamic prefetching in JIAJIA, a software DSM based on scope consistency [Pal 96], has been proposed and evaluated. There are two features in our prefetching algorithm: it's sensitive to stride access pattern and can be combined into JIAJIA's lock-based coherency protocol without much modification. The test results show that it has high prediction accuracy and achieves good overlap between communication and computation in benchmark applications. This work gives a good proof that study on software-based dynamic prefetching is reasonable, promising and worth to make further steps.

1 Introduction

Over the past decade, software Distributed Shared Memory (DSM) systems have been extensively studied to provide a good compromise between programmability of shared memory multiprocessors and hardware simplicity of message passing multicomputers. In DSM systems, physical memories of multiple workstations are combined to form a larger shared space. Many current software DSM systems, such as Ivy[Li 88], Midway[Bershad 93], Munin[Carter 91], TreadMarks[Keleher 94], and CVM[Keleher 96] have been implemented on the top of message passing hardware or on network of workstations.

Communication latency is generally much larger in a DSM system than it would be in a multiprocessor system, and bandwidth generally much lower. This leads to different design decisions in areas such as memory consistency and cache coherence where the increased latency increases the cost of operations. Several relaxed memory consistency models, such as release consistency (RC), lazy release consistency (LRC), and entry consistency (EC) have been implemented in software DSM systems to reduce communication latency. JIAJIA implements the scope consistency (ScC)[Singh 96], which is even lazier than lazy release consistency (LRC). Adopting the ScC greatly simplifies the lock-based cache coherence protocol of JIAJIA, and thus achieves a higher performance.

While relaxed consistency models and coherence protocols enhance software DSMs by reducing communication overhead as much as possible, prefetching provides an alternative way to reach higher performance by overlapping computation with communication. Hardware and software prefetch mechanisms speculatively bring data to a processor that is likely to be accessed soon. Some attempts have been made on software-controlled prefetching [Quoc 98]. With programmer-inserted prefetching and compiler-inserted prefetching, it appears to achieve a better performance on many applications. However, the results in investigating prefetching with runtime information are not satisfied. Although using dynamic information to issue prefetching can overcome some of the limitations of statically inserted prefetching, the overheads of this approach often involve more small-sized message communications and more complex coherency protocols that finally offset any gain in memory performance.

Basically, seeking dynamic runtime prefetching mechanism suitable for all kinds of applications is extremely difficult under current hardware and software technology. To focus on solving some specific applications, however, is promising and worthful in real world. Today, more and more scientific computation parallel algorithms have been rewritten under software DSM due to the advantage of its large shared linear address space. And statistic report shows most scientific applications have similar memory access regularity such as consecutive, stride or nested-stride based data access patterns. Thus accelerating such applications by prefetching is a valuable work and will make deep impact on the fields of both SVM and scientific computing.

In this report, an experimental work, adding a specific dynamic prefetching in JIAJIA, has been introduced and evaluated. This prefetching algorithm is proposed to be sensitive to those applications with stride access pattern (say, MM, LU and FFT). Furthermore, it can be easily added into JIAJIA's scope consistency model without much modification. The test results show that it has high prediction accuracy and achieves good overlap between communication and computation in some benchmark scientific applications.

The rest of the report is organized as follows. The following section introduces background knowledge about JIAJIA. Section 3 describes our prefetching algorithms in detail. Section 4 presents the key issues and data structures in our implementation. Section 5 gives the performance tests and analysis on those results. Conclusions are made in the end of this report.

2 Background Knowledge about JIAJIA

2.1 Memory Organization of JIAJIA

Unlike other software DSMs that adopt the COMA-like memory architecture, JIAJIA organizes the shared memory in a NUMA-like way. In JIAJIA, each shared page has a fixed home node and homes of shared pages are distributed across all nodes. References to local part of shared memory always hit locally. References to remote shared pages cause these pages to be fetched from its home and cached locally. When

cached, the remote page is kept at the same user space address as that in its home node. In this way, shared address of a page is identical in all processors; no address translation is required on a remote access. Figure 2.1 shows JIAJIA's organization of the shared memory.

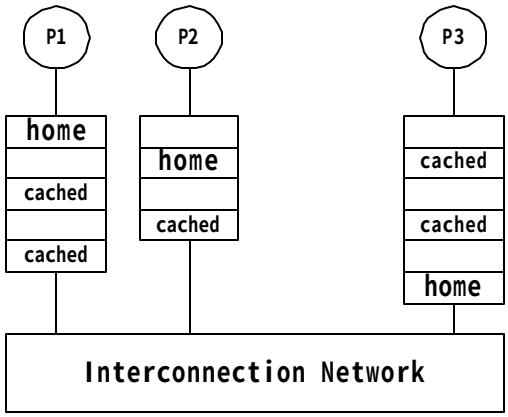

Figure 2.1. Memory Organization of JIAJIA

With the above memory organization, JIAJIA is able to support shared memory that is larger than physical memory of one machine. In other software DSMs such as TreadMarks, CVM, and Quarks, the shared space is limited by the physical memory of one machine because no cache replacing mechanism is implemented and hence each host should have the capability of holding all shared pages.

Another important feature of JIAJIA's memory organization is that it allows the programmer to flexibly control the initial distribution of homes of shared locations. The basic shared memory allocation function of JIAJIA allows the programmer to allocate a certain size of shared memory block by block across all processors.

2.2 Memory Consistency Model for JIAJIA

Systems that maintain coherence at large granularity such as shared virtual memory systems, suffer from false sharing and extra communication. Relax memory consistency models have been used to alleviate these problems, but at a cost in programming complexity. Release Consistency (RC) and Lazy Release Consistency (LRC) are accepted to offer a reasonable tradeoff between performance and programming complexity. Entry Consistency (EC) offers a more relaxed consistency model, but it requires explicit association of shared data objects with synchronization variables. The programming burden of providing such associations can be substantial.

JIAJIA implements scope consistency model (ScC), which offers most of the performance advantages of the EC model without variables. Instead, ScC dynamically detects the associations implied by the programmer, using a programming interface similar to that of RC or LRC.

A consistency scope is a limited view of memory with respect to which memory references are performed. That is, modifications to data performed within a scope are

only guaranteed to be visible within that scope. We can think of a consistency scope as consisting of all critical sections protected by the same lock. Additionally, barriers define a global consistency scope, which includes the entire program. For each consistency scope there is an open operation that opens the scope and a close operation that closes the scope. The interval during which a consistency scope is open at a given process (e.g. a given critical section protected by the lock) is called a session. Any modifications made within a consistency scope session become visible to processes that then enter new sessions of that scope (acquire that lock or pass a barrier). Modifications made outside the scope session are not guaranteed to be visible. Figure 2.2 illustrates consistency scopes. Memory is accessed through different perspective or scope is built out of sessions, which occur on different processes. Sessions belonging to different scopes can interleave in time or overlap in the code or memory. In addition to the individual consistency scopes (whose sessions may be delineated by locks and unlocks, say), there is also a global consistency scope with regard to which all memory references are performed. The sessions of the global scope are typically delineated by barriers.

To define the model we need an additional definition to distinguish a reference being performed with respect to a process:

A write that occurs in a consistency scope is performed with respect to that scope when the current session of that scope closes.

The consistency rely for Scope Consistency are:

1. Before a new session of a consistency scope is allowed to open at process P, any write previously performed with respect to that consistency scope must be performed with respect to P.
2. A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened.

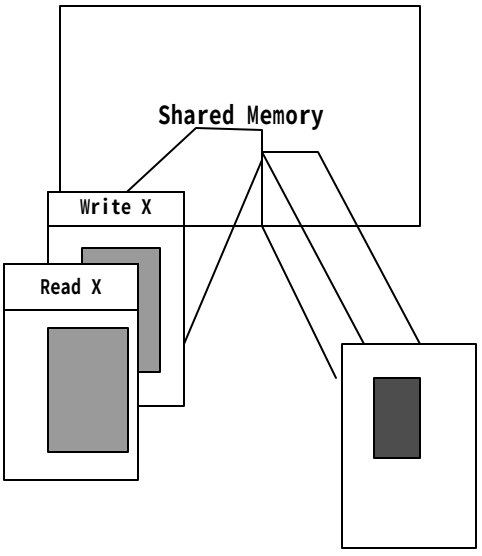

Figure 2.2 Consistency Scopes

3 Prefetching Algorithm

3.1 Address-based Prefetching

Hardware and software prefetch mechanisms speculatively bring data to a processor that is likely to be accessed soon [Koppelman 00]. Two common prefetching methods are sequential and stride [Fu 92]. Sequential prefetching is designed to exploit reference (memory address) sequences of the form *a… a+1…a+2…* The prefetching mechanism may react to the use of an address *a* by fetching just *a+1* or for systems where the fetch latency is high, by fetching *a+1, a+2…a+d,* where *d>0* is the prefetching degree. Such sequences are of course quite common, generated, for example, by sequential access to an array.

In adaptive sequential prefetching (ASP) [Dahlgren 95] the degree is adjusted based on the success of past prefetches. The hardware keeps track of the number of prefetched line that have been used. After some number of prefetches, the number of used prefetches is checked and the degree adjusted.

Stride prefetching is designed to exploit reference sequences of the form *a…a+s…a+2s…*, where *s* is the stride of the prefetch. Such sequences might be generated by accessing an array at some stride (every *s*' th element) or by sequential access to an array of large elements, with only a part of each element being accessed.

These address-based prefetching algorithms, however, can't be directly used in page-based software DSMs. The overhead will be extremely high if keeping track of memory access operations by software. Since most SVMs implement remote memory access by catching page faults, we can only exploit such information to make address prediction. In the next subsection, our brand-new prefetching algorithm based on page prediction is proposed and introduced.

3.2 Page-based Dynamic Prefetching Algorithm

Our new algorithm aims at exploiting the page access patterns corresponding to the consecutive stride based data access patterns.

We still assume the reference sequences pattern as ***a..a+s..a+2s…*** The corresponding page references are ***b_1…b_2…b_3… (b_i*** is page number), and page size is d.

Theory: Assume $s = d \times p + r$ (0≤r<d), here p and r are both integers, then the difference of two consecutive page numbers b_i and b_{i+1} is either p or p+1. Formulize as: $b_{i+1} - b_i$ **= p or** $b_{i+1} - b_i$ **= p+1.**

Proof: Assume the base address of page b_i is $b_{1(a)}$. According to the given conditions, ***a*** falls in page b_1, assume its offset within b_1 is n. n satisfies: 0 ≤ n < d. Then the address of the second data ***a+s*** satisfies:

$$a+s = (b_{1(a)} + n) + (d \times p + r) \qquad (1)$$

Here r and n satisfy 0 ≤ r < d and 0 ≤ n < d, according to the inequality property, we can reason that 0≤ r + n < 2d, bring it into (1), we conclude that:

$$b_{1(a)} + d \times p \le a + s < b_{1(a)} + d \times p + 2d = b_{1(a)} + d \times (p+2) \qquad (2)$$

When $a+s \in [b_{1(a)} + d \times p,\ b_{1(a)} + d \times (p+1)\ )$, ***a+s*** falls in page b_1 + p.

When $a+s \in [b_{1(a)} + d \times (p+1),\ b_{1(a)} + d \times (p+2)\ )$, $a+s$ falls in page b_1 + p+1.

Thus we have proved $b_2 - b_1$ = p or $b_2 - b_1$ = p+1.

Using the same method, we conclude that for all consecutive access pages b_i and b_{i+1}, either $b_{i+1} - b_i$ **= p or** $b_{i+1} - b_i$ **= p+1 holds.**

Implementing this simple theory, we can do a perfect prediction for the access pattern of the page sequences. The process is as following:

1) Prefetch will begin when two consecutive page faults have been detected. The page numbers b_1, b_2 are recorded.

2) Calculate $b_2 - b_1$, let $k = b_2 - b_1$

3) Prefetch three consecutive pages $b_2 + k - 1$, $b_2 + k$ and $b_2 + k + 1$.

4) int i = 3, unsure = 1, state;

```
while( hit in the prefetching queue)
{
   if (unsure)
   {
         if (bi == bi-1 + k – 1)
                {
                   set unsure = 0;
                   set state to –1;
                   prefetch page bi + k – 1 and page bi + k;
                 }
          else if (bi == bi-1 + k )
                   {
                    set unsure = 0;
                    set state to 1;
                    prefetch page bi + k and bi + k +1;
                   }
                else
               prefetch page bi + k –1 and bi + k and bi + k +1;
     }
     if (!unsure)
     {
          if (state == -1)
            prefetch page bi + k – 1 and page bi + k;
          if (state == 1)
            prefetch page bi + k and page bi + k+1;
       }
        i++;
}
```

Analysis:

In step (2), k = p or k = p+1. But we can't tell the exact value of k at this time. If k = p, then the next data will fall in page $b_2 + k$ or $b_2 + k + 1$; If k = p+1, then the next data will fall in page $b_2 + k - 1$ or $b_2 + k$. So if we are unsure of the value of k, to avoid miss, we have to prefetch the next 3 pages $b_2 + k-1$, $b_2 + k$ and $b_2 + k + 1$. Three pages prefetching will continue until variable unsure change to 1 (see step(3)). After that, we can continue prefetching 2 pages each time.

Step(3) also shows that our prefetching will stop once the next data missed in the prefeching queue.

3.3 Simplified Algorithm

Page-based prefetching has some constraints:

1) This prefetching algorithm will at least prefetch two pages each time in order to achieve the maximum hit rate. This will cause three problems: the first is at least one page is waste each time since the data could only reside in one page, if the data sequence's length is L, then at least L pages will be prefetched unused. These will lead to at least double communication time than prefetching one page at a time.

1) Two pages prefetching also has the potential for false sharing, which will cause more latency.
2) You can't increase the prefetch degree. For example, if you try to extend the prefetch degree to 2, assume now you hit page b_i, then next time you have to prefetch $b_i + p$, b_i+p+1, b_i+2p, b_i+2p+1, b_i+2p+2, there will be 3 pages waste and 5 times communication and more potential of false sharing. You can imagine, when the degree extends to n, one time prefetching number of pages will reach $O(n^2)$, Compare this data, one degree prefetch is best in this algorithm, but the performance improvement of prefetching will be limited.

For convenience, we introduce a new array $c_1,c_2,c_3\ldots$, where $c_i = b_{i+1}-b_i$.. As we have proved, $c_i = p$ or $c_i = p+1$. Consider the page access pattern deeper, we found that the sequence $c_1,c_2,c_3\ldots$, always appears in two patterns:

1) p, p…p, p+1, p, p…p, p+1….

or

2) p+1, p+1…p+1, p, p+1, p+1…p+1

Let $n = \left\lfloor \frac{d}{r} \right\rfloor$, if $n \geq 2$, then the sequence of $c_1,c_2,c_3\ldots$ will appear as: pp…p, p+1, pp…, where the average number of every sequential p (which are concatenated by single p+1) is n. The probability that p will happen is near n/(n+1), with the increase of n, n/(n+1) will increase rapidly until it approaches to 1. On the other hand, if n = 1 $c_1,c_2,c_3\ldots$ will appear as: p+1, p+1…p+1, p, p+1, p+1…p+1, p… , in this case, we will consider $m = \left\lfloor \frac{r}{d-r} \right\rfloor$. The average number of every sequential p+1 is m. The probability that p+1 will happen is near m/(m+1). Similar to the above analysis, with m

increasing, m/(m+1) will increase rapidly until it approaches to 1. Then we conclude that the chance that we will keep prefetching pages with same stride is high. So most of the time, we can keep prefetching pages with the same stride successfully.

Then we get our optimized page-based prefetching algorithm:

1) Observe page fault and calculate stride p between two sequential fault pages
2) Not begin prefetching until k consecutive fault pages have the same stride p.
3) Prefetch the next page according to p.
4) If hit, turn to (3)
5) If miss, calculate the difference between the missed page b_i' and the newly prefetched page b_i, if $b_i' - b_i = 1$ or -1, discard b_i, turn to (3) (notice, the next prefeching page number is $b_i' + p$ not $b_i + p$).
6) If $b_i' - b_i \neq 1$ or -1, stop prefetching.

Notice, in step (2), k is a parameter we can adjust to decide when to begin our prefetching. We find that when k = 3, the prefetching effect is best. When the data access doesn't have an obvious stride pattern, we won't call the prefetching mechanism, in this way, we are protected from doing useless prefetching, and thus we can achieve at least the same performance with JIAJIA without a prefetching mechanism.

In step (6), as we have proved, the stride of two pages only have two possible consecutive values, p or p+1, when detecting a distance not satisfying this rule, we can conclude that the data access pattern has changed. The prefetching should stop now.

3.4 Keeping Memory Consistency with Prefetching

Scope consistency model updates values only when every process reaches the same scope boundary. So prefetching might violate memory consistency when data are prefetched in the current scope and accessed in the next scope. For example, we can see the following code segments that when P1 executes *Max = max(a[0],a[1],a[2])*, it will possibly prefetch *a[3]*. But at this time *a[3]* has not been updated by P2 yet. Then memory consistency will be violated in the line *Max = (Max, a[3])*.

P1:	***P2:***
…….…….	*…….…;*
Max = max(a[0], a[1], a[2])	*a[3] = max(a[3], a[4], a[5]);*
Jia_barrier();	*Jia_barrier ();*
Max = max(Max, a[3]);	*…..…*
…..…	

Notice that such inconsistency only occurs at the scope boundary. So we simply discard those pages already prefetched or currently under prefetching if they are announced to be in invalid state during the scope update procedure. Reaping benefit from scope consistency and lock-based coherency protocol, we can keep the consistency model easily compared to ERC and LRC models.

4 Implementation Issues

In this section, we first give a brief introduction to the page fault handling procedure, and then illustrate some key data structures, functions and tricks used in our implementation.

4.1 Page Fault Handler

The whole page fault handling procedure is shown in Figure 4.1. First, the miss page number is calculated and added to the history page table. Then it checks whether the page is in the prefetch queue. If hit, it simply calls *mmap* to map the prefetched page in its virtual address. Otherwise it sends a page request message to page's home memory and wait until the page is available. After mapping the miss page, it calls our prefetch algorithm to prefetch pages to be possibly accessed soon.

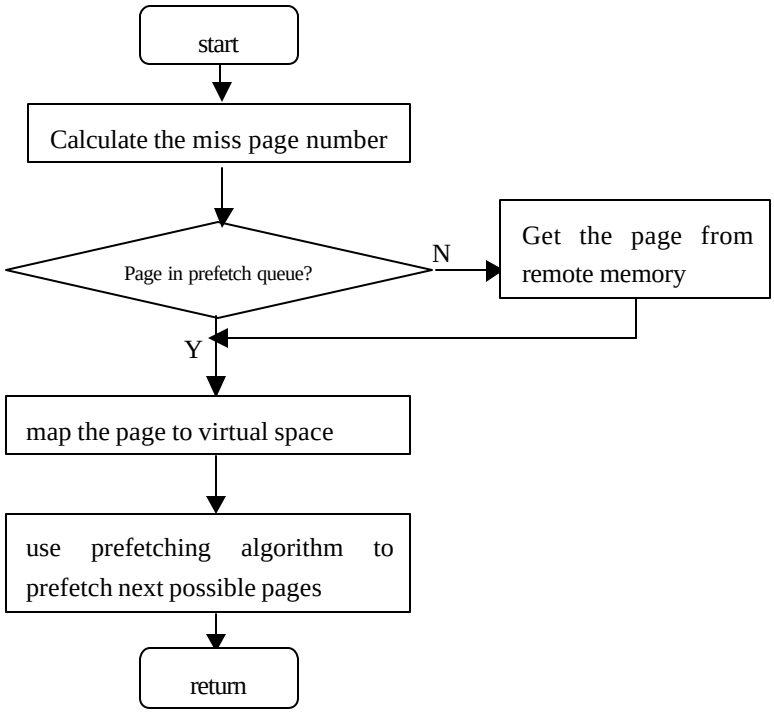

Figure 4.1 Page Fault Handling Procedure

4.2 Distinguishing Different Shared Variables

In most applications, stride pattern only occurs within a single variable (say, array variable) 's data reference sequence but seldom occurs among different variables' reference sequence. Then how to distinguish different shared variables so that our prefetch prediction algorithm can make sense according to the page number? In JIAJIA, shared variables are allocated only by calling jia_alloc. So we modified this function to record each variable's start page number and its range. For a given page number, it's

easy to decide which variable it belongs to by calculating which variable range it falls in.

4.3 Prefetch Queue

A prefetch queue contains all the pages that have been prefetched yet not used. Each element in the queue has three possible states:

READY - the page is in the queue

WAITING - the page has been requested for prefetching but not ready now.

EMPTY - the cell isn't occupied

It's obvious that when a page prefetch request is sent out, the page number will be added into the queue and its state will be set to *WAITING* and when the page is prefetched from remote memory, its state will change to *READY*. But when a page should be removed from prefetch queue? There are three cases in which the element should be removed.

- When the page in the prefetching queue is hit, it will be mapped to the virtual address and then should be removed from the queue.
- When a page isn't hit in the prefetch queue, those pages belonging to the same variable as this page should be removed. For stride pattern, it's reasonable to treat those pages as mispredicted ones and thus should no longer exist.
- When the page has been updated by other processors at the boundary of a scope, it should be removed.

When the queue is full, adding a new page to the queue will fail and no prefetch request will be launched because we have no reason to eliminate any pages currently in the queue.

4.4 Tracing Memory Reference

Is it a good idea for the prefetch queue to record only page numbers and their states? Consider that when a page has been prefetched, it can be directly mapped into the virtual address space. And the prefetch queue can also use page number to unmap a page that should be removed from the queue. However, if we directly map a prefetched page into a virtual address and assume it will be accessed in the next memory reference, we can't get this page number because accessing this page won't incur a page fault any more. That means a successful prediction will diminish our future prefetching prediction accuracy due to the incomplete memory reference sequence. In our implementation, we provide extra memory to store those prefetched pages and don't map them into virtual address until the application accesses this page.

5 Experimental Results and Analysis

To evaluate the performance of our stride-based prefetching in JIAJIA, we ported several widely accepted benchmarks for DSM system, including LU from SPLASH2, IS from NAS and a matrix multiplication (MM) written by ourselves.

Ideally, the evaluation should be done on a cluster of workstations or PCs connected

by Myrinet or Ethernet. Being lack of such equipment, however, we only found five PCs donated by my friends and did our experiments on SCS LAN. Fortunately, the result shows that our prefetching strategy has effectively overlapped the computation and communication in applications with stride-based data access pattern and thus enhanced the whole DSM performance. Those PCs are connected by 1000Mb Ethernet(I guess so) . Each node has a Pentuim III 1GHz processor with 512MB local physical memory, running under RedHat Linux 7.0.

5.1 MM

The MM is a matrix multiplication program using inner product algorithm. Arrays are distributed uniformly among all processors in row order to reach good data locality. Table 5.1 shows the execution time with no prefetching and stride-based prefetching respectively under different number of processors and matrix sizes.

Data in Table 5.1 depict that JIAJIA with prefetching does perform better over the original one. The optimization become more obvious as the problem size increases. Furthermore, we can see that the prefetching hit rate is extreamly high (around 98%) due to the strict stride access pattern in MM. That is a good proof for the prediction accuracy of our algorithm.

Basically, the execution time saved by prefetching strategy depends on the overlapped part of communication and computation. Therefore, total communication time should be the maximum saved time by prefetching and the absolute speedup of execution time might not be a proper gauge here. In order to evaluate the effectiveness of our prefetching, we calculated the proportions of saved communication time (overlapped by computation) to the total communication time (shown in Figure 5.1).

This figure depicts that normally over 50% communication time has been overlapped by computation, which seems a little far beyond our initial expectation. As the number of processors increases, more and more data need to be transferred among those nodes. We can see from the high hit rate that almost all the communication are successfully predicted and prefetched by our algorithm. So it's reasonable to expect a better *saved/total comm. time* ratio if running under more processors. But why the ratio still remains the same or even worse? In fact, it's impossible to overlap communication and computation perfectly. Consider that if a processor sends a prefetching request to the other, the other processor has to stop its computation and handle the remote request. We use signal to invoke such service in our implementation and thus we can't execute computation thread and network service simultaneously. Therefore, each processor will suffer from heavy prefetching requests when communication becomes frequent. That's an obstacle to the scalability of prefetching if no mechanisms support true parallelism between computation and network communication service.

The other wired result is the performance on 5 processors isn't so good as other cases. As we know, in this case each processor will fetch 4/5*N rows (we can reasonably consider them as 4/5*N pages because each float element takes 4 bytes and each row contains at least 1024 elements, which has already exceeded a page size 4096) from remote processors to compute the value of an element. However, the local memory may not hold all the prefetched pages when N becomes large. Then some pages have to be

pulled out and be prefetched when computing the next element and thus it needs extra prefetch operations or communication, which leads to the reason discussed above. Anyway, the speedup and absolute performance is acceptable.

Matrix Size	Processors											
	2			3			4			5		
	no	pre	comm.	no	pre	comm.	no	pre	comm	no	pre	comm.
1024	4.92	4.65* (98.4%)	0.380	3.73	3.39 (98.83%)	0.506	3.20	2.81 (98.9%)	0.570	2.95	2.52 (99.0%)	0.609
2048	35.91	34.76 (99.6%)	1.524	26.25	24.38 (99.7%)	3.218	21.8	19.67 (99.7%)	3.914	18.8	16.46 (99.8%)	5.252
4096	271.54	266.21 (99.8%)	7.160	196.82	190.6 (99.8%)	10.501	155.6	147.08 (99.8%)	12.069	123.21	114.65 (99.8%)	23.008

no ---- execution time (in seconds) without prefetching

pre ---- execution time (in seconds) with pref etching

comm. ---- total communication time (in seconds)

* ---- prefetching hit rate is shown in parentheses

Table 5.1 Execution Time of MM Without Prefetching and With Prefetching

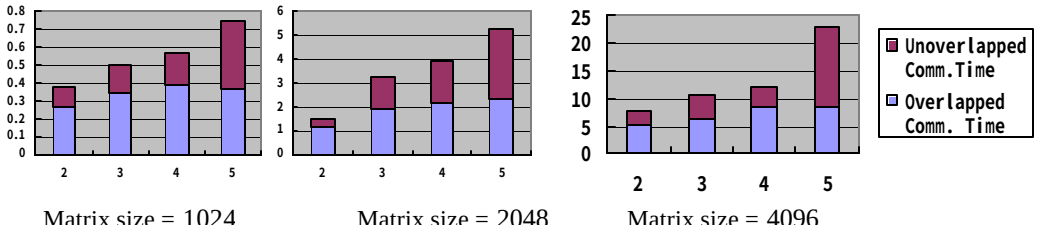

Figure 5.1 Saved Communication Time (Overlapped with Computation) vs Total Communication Time (in seconds)

Besides, software DSM page granularity seems not to be a neglectable factor as well as those parameters discussed above. We tested 1024*1024 MM with different page sizes (shown in Table 5.2 and Figure 5.2) to evaluate how page size affects our prefetching strategy (say, it might reduce prefetching frequency as page size increases) and thus the execution time.

Page Size	Processors							
	2		3		4		5	
	no	pre	no	pre	no	Pre	no	Pre
4096	5.24	4.89	4.11	3.68	3.61	3.32	3.71	3.45
8192	4.92	4.65	3.73	3.39	3.20	2.81	2.95	2.52
16384	4.82	4.59	3.53	3.27	2.97	2.66	2.73	2.47

Table 5.2 Execution Time (in seconds) of 1024*1024 MM under Different DSM

Page Size

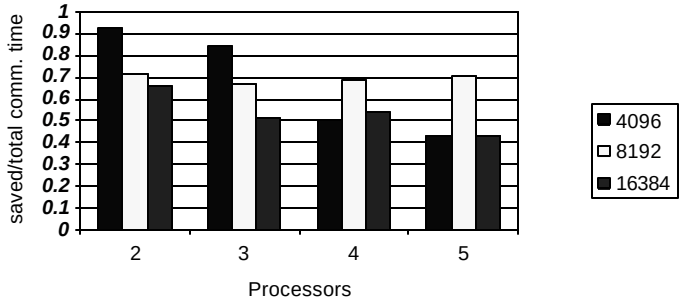

Figure 5.2 The Ratio of Saved Comm. Time to Total Comm. Time under Different DSM Page Size

The data shows the total execution time decreases as page size increases. For MM problem, large page size will reduce remote memory access times (page fault) and thus reduce execution time. But there is tradeoff between page size and *saved/total communication ratio*. Small page size incurs frequent prefetchings while large size will quickly exhausts the space of prefetching queue. We can conclude from the figure that 8192Bytes is more proper than 4096 and 16384 and consistent with JIAJIA's default page size.

5.2 LU

The LU program factors a matrix into the product of a lower triangular and an upper triangular matrix. To exploit temporal locality the matrix is split into B*B blocks of submatrixes. The execution time is shown in Table 5.3. Because data access pattern in each loop of LU is nearly stride but isn't between two consecutive loops, our prefetching prediction accuracy (the ratio pages by prefetching to total pages by remote access) isn't as high as MM. The *saved/total communication ratio* is similar to MM though, which contributes to our conservative prefetching algorithm.

Matrix Size	Processors					
	2		3		4	
	no	Pre	no	pre	No	Pre
512	8.21	8.08 (83.3%)	6.22	6.10 (78.6%)	5.94	5.72 (81.3%)
1024	51.46	50.86 (90.3%)	36.18	35.87 (82.5%)	28.28	27.50 (74.5%)
2048	402.38	394.78 (93.1%)	278.1	272.2 (90.9%)	207.80	203.69 (81.16%)

Table 5.3 Execution Time (in seconds) of LU

5.3 IS

IS ranks an unsorted sequence of keys using bucket sort algorithm. Keys are distributed among processors and communication happens only at the end of each local sort. So no stride-based shared memory access pattern exists in IS. We choose this program to test whether our prefetching algorithm will greatly reduce JIAJIA's performance under its worst case.

Test results (Table 5.4) show that there is no much performance difference between prefetching and no prefetching. As we mentioned before, our conservative prefetching algorithm won't send prefetching requests unless it believes current data access pattern is a stride. Thus there are no actual prefetching operations performed in IS.

Size	Processors					
	2		3		4	
	no	Pre	no	Pre	No	Pre
2^{22}	1.42	1.377	1.792	1.834	1.953	1.996
2^{23}	2.622	2.570	2.443	2.494	2.591	2.619
2^{24}	6.061	5.890	5.833	5.805	5.708	5.699

Table 5.4 Execution Time (in seconds) of IS

6 Conclusions and Future Work

Memory prefetching provides an efficient way to reduce application's overall execution time by overlapping communication with computation under DSM systems. Software-based dynamic prefetching, however, is not satisfied due to its formidable overhead. Avoiding seeking for a general solution, we focus on evaluating the performance of some specific prefetching mechanism, which is suitable to a certain class of applications, in a software DSM JIAJIA based on scope consistency. In this report, we proposed a page-based, stride-sensitive prefetching algorithm. It can be combined into JIAJIA's scope consistency model without much modification of coherency protocol. Then we ported three scientific applications from benchmark to evaluate the performance of our prefetching algorithm. The experimental results show that this algorithm has high prediction accuracy in applications with stride access pattern. And in average more than 50% communication time is well overlapped with computation time in MM and LU. For IS, an application with none-stride access pattern, our conservative prefetching strategy also achieves similar performance compared to the original JIAJIA system.

Our current work is still very far to reach the final goal - software-based dynamic prefetching. Inspired by this work, we are confident to make further steps in this research. We aim our future work at three aspects: The first is to add and evaluate more prefetching algorithms (say, algorithm sensitive to nested-stride access pattern) in JIAJIA. The second is to seek for proper and efficient prefetching algorithms for RC-based or EC-based SVMs. The third is to study on the scalability of software based prefetching (say, can the scalability problem discussed in 5.1 be alleviated to some extent?)

Acknowledgement

First of all, we would like to thank Prof. Todd C. Mowry for his kind help on our project proposal and providing us related materials. Thanks also go to Hua, Yiheng and Xuerui , who provided us their computers to perform our tests. Finally, we thank JIAJIA develop group for giving us their source codes.

References

[Bershad 93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In Proceedings of the 38th IEEE Computer Society International Conference, pages 528--537. IEEE, February 1993.

[Carter 91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of MUNIN. In Proceedings of the 13th ACM Symposium on Operating Systems Principles, pages 152--164, October 1991.

[Dahlgren 95] F. Dahlgren, M. Dubois, and P. Stenstrom, Sequential Hardware Prefetching in Shared-Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, vol. 6, pp. 733--746, July 1995.

[Fu 92] John W. C. Fu, Janak H. Patel, and B. L. Janssens. Stride Directed Prefetching in Scalar Processors. In MICRO-26, 1992.

[Hu 99] Weiwu Hu, Weisong Shi, Zhimin Tang . JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In Proceedings of the High Performance Computing and Networking (HPCN'99) , LNCS 1593, pp. 463-472, Springer, April, 1999, Amsterdam, Netherlands.

[Keleher 94] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In Proceedings of the 1994 Winter Usenix Conference, pages 115--131, January 1994.

[Keleher 96] Keleher, P. The Relative Importance of Concurrent Writers and Weak Consistency Models. In Proceedings of the 16 th International Conference on Distributed Computing Systems. 1996.

[Koppelman 00] D.M. Koppelman, Neighborhood prefetching on multiprocessors using instruction history. To appear in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, October 2000.

[Li 88] Kai Li. IVY: A shared virtual memory system for parallel computing. In International Conference on Parallel Processing, pages 94--101, 1988.

[Pal 96] Jaswinger Pal, Singh Liviu Iftode and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. Technical Report TR-509-96, Princeton, NJ, January 1996.

[Quoc] Charles Quoc Cuong Chan. Graduate Department of Computer Science, University of Toronto.