Hardware Spatial Forwarding for Widely Shared Data *

Marius Pirvu
Department of Computer Science
Texas A&M University
College Station, TX 77845
pirvum@cs.tamu.edu

Laxmi Bhuyan
Department of Computer Science
Texas A&M University
College Station, TX 77845
bhuyan@cs.tamu.edu

ABSTRACT

Applications with widely shared data do not perform well on cc-NUMA multiprocessors due to the hot-spots they create in the system. In this paper we address this problem by enhancing the memory controller with a forwarding mechanism capable of hiding the read latency of widely shared data, while potentially decreasing the memory and network contention. Based on the influx of requests, the memory anticipates the next read references and forwards the data in advance to the processors. To identify the set of processors the data is to be forwarded to we use a heuristic based on the spatial locality of memory blocks. To increase the forwarding effectiveness and minimize the number of messages, we incorporate simple filters combined with a feedback mechanism. We also show that further improvements are possible using a combined software-prefetching/hardware-forwarding approach. Our experimental results obtained with a detailed execution driven simulator with ILP processors show significant improvements in execution time (up to 37%).

1. INTRODUCTION

Recent research has shown that applications with widely shared data memory access patterns perform poorly on cc-NUMA multiprocessors. As the machine's size increases, such data could easily create hot-spots in the interconnect or saturate the memory subsystem. This determines an increase in the latency of the memory accesses, thus limiting the scalability of these applications.

Most of the previous work in this area has focused on improving the latency of a cache miss by using custom switches in the interconnection network. Iyer and Bhuyan [13] propose to introduce small caches in the crossbar switches of the interconnect to capture the widely shared data as it flows from memory to processors. Subsequent read requests (from other processors) hit in the switch cache, thus being served faster. A similar idea is used by Kaxiras and Goodman in their GLOW cache coherence protocol extensions [15]. A k-ary logical tree structure is embedded in the network. The root of the tree represents the memory module while the leaves stand for the processors. Any intermediate node acts as a "GLOW agent". A GLOW agent caches directory information and optionally data. The widely shared data must be identified (either statically or dynamically) and the GLOW agents are informed about it. From this moment on, multiple requests for a cache line generate only a new request towards the home node, thus reducing the network traffic. Hence, an effect similar to combining [11] is achieved.

In this paper, we study an alternative solution: Rather than reducing the cache miss penalty we concentrate on improving the cache hit ratio. We enhance the memory controller with a simple, yet efficient, forwarding engine. Based on the full-map directory, the engine anticipates the requests coming from processors and pushes the desired data in their caches ahead of time, thus reducing the miss ratio.

Our technique exploits the spatial sharing pattern similarities of memory blocks, and thus is amenable to regular applications with spatial locality. To increase the forwarding accuracy we provide filters based on simple heuristics. Moreover, we devise a feedback mechanism that helps the memory controller learn from mistakes and turn off forwarding for the cases where our basic assumptions about spatial similarity are incorrect.

Since a multiprocessor is a complex system, our solution to memory latency hiding does not intend to be a panacea. Rather, we confine ourselves to just some aspects. For instance, we target only the latency related to data transfers. Data fetching tends to be much more of a hindrance when compared to instruction fetching [3]. Also, we are interested solely in improving the read latency because, with a release memory consistency model [9], the write latency can be almost completely hidden. Finally, as mentioned previously, we limit our scope to widely shared data. Even with these restrictions, our experiments based on an accurate execution driven simulator with ILP processors indicate that substantial improvements in execution time (up to 37%) are possible.

Our method is germane to both data prefetching [5, 20, 22, 18, 27, 14, 8, 2] and producer initiated communication (also

*This research has been supported by NSF Grants CCR 9622740 and CCR 9810205

known as data forwarding) [26, 16, 1]. Hence, we think that a short comparison is required.

Memory controller forwarding differs from producer initiated communication in that it does not require any specialized compiler intervention and/or modifications to the instruction set. On the other hand, traditional producer initiated communication mechanisms rely heavily on the compiler to insert *deliver* instructions [17] into the code. The performance of such a method depends greatly on the compiler's ability to statically detect producer-consumer pairs. This is the most important obstacle to producer initiated communication, because the current compiler technology makes this task very difficult and sometimes impossible. Furthermore, manufacturers are very reluctant when it comes to modifying the instruction set or adding new features to a CPU from which only a multiprocessor machine could benefit. Changes at the memory controller, like in our forwarding mechanism, are by far more appealing. Moreover, hardware modifications are sometimes preferable because they can improve the execution time of the large base of existing binaries without the need for recompilation.

Compared to data prefetching memory controller forwarding has several advantages:

1. Before a prefetch message is issued to the memory, the presence of the desired line must be checked against the cache. Such an operation might hinder the read/write operations that want to access the cache at the same time. The lack of tag-cache bandwidth is acute in an aggressive superscalar processor where several load and store operations can be issued in a single cycle [28, 31]. Prefetching exacerbates this problem even further [25]. The straightforward solutions of duplicating the tags or creating multiple cache ports are far from being satisfactory. They are expensive in terms of circuitry and therefore designers legitimately try to circumvent them [30]. Our method avoids this nuisance by placing the forwarding mechanism at the memory controller. One might argue that not much has been saved because the checking of the cache has been replaced by the checking of the directory. However, adding 2-3 extra cycles to a miss (150-300 cycles) can be considered as negligible, while increasing the latency for a cache hit even with a few cycles can have a dramatic effect on execution time. Furthermore, the directory lookup can be totally overlapped with the memory access.

2. Memory controller forwarding can decrease the memory contention for widely shared data, because only one memory access is performed in order to forward data to several processors.

3. Another characteristic of memory controller forwarding is the potential for reducing the network traffic. Prefetching is essentially based on request/reply pairs of messages. Since our mechanism is embedded in the memory controller we are able to eliminate the request message. This is a positive aspect because the lack of interconnect bandwidth is known to adversely impact the performance of prefetching [29].

To summarize this paper makes the following contributions:

- Proposes a novel hardware forwarding technique based on spatial locality.
- Devises filters and feedback mechanisms to increase forwarding efficiency.
- Thoroughly evaluates the proposed architectural enhancements pointing out their advantages and limitations.
- Studies the synergic effect of software-prefetching and hardware-forwarding and compares the advantages/disadvantages of these two methods.

The rest of the paper is organized as follows. Section 2 details the proposed architectural improvements. The simulation test-bed and the set of benchmarks are described in Section 3. Section 4 comments on the experimental results, and finally, Section 5 concludes the paper.

2. SPATIAL FORWARDING

Our forwarding mechanism is based on a simple engine embedded in the memory controller which speculatively forwards widely shared memory blocks ahead of time to the processor caches. In this section we describe our technique. First, we introduce a heuristic for determining the set of processors the data is to be forwarded to. Then, we enhance our method with simple filters and a feedback mechanism to improve the forwarding accuracy. Finally, we discuss hardware implementation issues.

2.1 Description of the Prediction Heuristic

As the name suggests, spatial forwarding is oriented towards regular applications that have spatial locality. The basic idea behind the spatial forwarding scheme is that, *if a line is shared by a group of processors, the next line will be shared by the same group.* This is almost always the case with regular applications. Conceptually, the forwarding mechanism is straightforward: when the memory sees a read request for line l, it examines the directory for a previous line, say $l-k$, and forwards the current line to all processors that cached line $l-k$, but do not have line l yet. We call the line whose sharing pattern is copied (i.e. $l-k$), the *stencil line* and k is referred to as the *stencil distance* (see Figure 1).

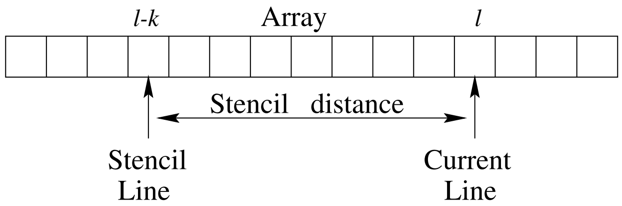

Figure 1: Exemplification of the stencil distance

To better illustrate our scheme we will use the well known algorithm for matrix multiplication as a working example. The way data are partitioned among processors is illustrated in Figure 2 and the algorithm (written in a C-like language) is presented in Figure 3. To boost spatial locality of data

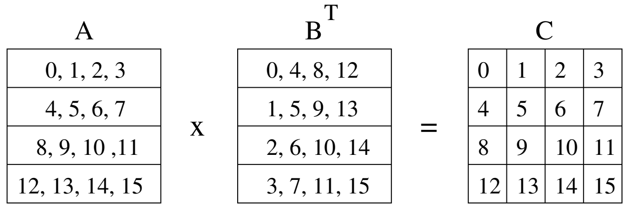

Figure 2: Data partitioning for matrix multiplication. The numbers represent the IDs of the processors that access a given region.

```
for (i = start_row[p]; i < end_row[p]; i++)
  for (j = start_col[p]; j < end_col[p]; j++)
    for (k = 0; k < n; k++)
       C[i][j] = C[i][j] + A[i][k] * B[j][k];
```

Figure 3: Code executed by processor p in matrix multiplication algorithm (matrix B is transposed)

accesses we first transpose matrix B, and then multiply matrices A and B^T. The transposition phase has a $O(n^2)$ time complexity while the multiplication phase has a $O(n^3)$ complexity. Hence, for large n, the execution time is dominated by the multiplication phase. To minimize the number of elements accessed by each processor, matrix C is logically partitioned into 16 square blocks (we assume 16 processors). Each block will be written by only one processor. Consequently, matrices A and B^T are logically partitioned into four bands. Each band is read by a *sharing group* of four processors as shown in Figure 2. Thus, each element from matrices A and B^T can be considered as widely shared. Furthermore, all lines belonging to a band have an identical sharing pattern. This feature is characteristic to many regular applications.

Another key observation is that the processors in a sharing group work more or less in lock step, which means that they all first access element one from a matrix row, then they all go to element two, etc. However, this *processor synchronicity* is not perfect. In practice the processors tend to slightly drift apart because of their different capabilities in processing requests. This discrepancy is mainly due to the distance of a processor to the memory module where data reside. The processors that are farther tend to lag behind others that are closer to the data. The first processor in a sharing group that touches a memory line is called the *leader of the sharing group* (with respect to that line). To maximize the efficiency of the forwarding mechanism we would like that, when the leader's request reaches memory, all other processors in its sharing group have already touched the stencil line. This ensures that a maximum number of forwards will be sent with just one memory access. Thus, we have to select a stencil distance that is large enough to meet this condition.

Nevertheless, a stencil distance that is too large assumes a very good spatial locality which might not always be the case. The locality is usually disrupted at the boundaries of data structures. For instance, at the beginning of an array, our forwarding method cannot be applied because the stencil line falls beyond the limits of that array. The larger the stencil distance, the larger the startup latency for the forwarding mechanism. Similar problems occur at page boundaries. We assume a cc-NUMA machine where the memory is distributed among the nodes based on a round robin policy at page level. Therefore, for some memory blocks the corresponding stencil lines reside in other nodes. In such cases the memory controller does not generate any forward messages.

In conclusion, we see that there is a tradeoff regarding the stencil distance. Empirically we determined that a value of 4 represents a good compromise. This value equals the degree of memory interleaving for our system. Thus, both the current requested line and its corresponding stencil line fall in the same memory bank. This fact could simplify the logic of the memory controller by allowing us to design a less sophisticated hardware in charge of just one bank, and replicating this circuitry for all four banks. More importantly, as we will explain in Section 2.3, this arrangement also helps reducing the average time needed to fetch the stencil line.

The memory data buffer

As discussed in the previous paragraph, it is very difficult to come up with the optimal stencil distance. Most often, when the leader's request reaches memory we are able to determine the majority of the processors in the sharing group, but not all of them. We also know that the rest of processors in the sharing group (the ones that were too slow) will issue their requests very soon (due to processor synchronicity). Based on this observation we propose to add a small FIFO buffer at the memory. When a line is forwarded, the data is also cached in this buffer. On a read request the buffer is searched associatively in parallel with the directory access. On a write, the matching entry (if any) is invalidated. Our simulations show that a buffer as small as 8 entries can be very effective in ameliorating the average memory latency seen by the processors.

2.2 Improving the Prediction Accuracy: Filtering and Feedback

To increase the performance of the Spatial Forwarding mechanism, we use simple filters that minimize the number of useless forward messages. As mentioned in the previous section, a simple bit mask filter using the directory bits prevents the memory controller from sending forward messages to processors that already have the line cached.

We employ a second filter to detect the situations where there is no spatial similarity in the application sharing pattern. This filter acts as follows: if processor p is the leader of a sharing group, then the forwarding mechanism is triggered only if p has its bit set in the directory entry corresponding to the stencil line. The reason is that, if processor p does not access the memory lines linearly, chances are there is not much spatial locality, and forwarding messages will be wasted. In our experiments this filtering proved to be quite effective in minimizing the negative effects of spatial forwarding when applied to irregular applications that have little spatial locality.

Moreover, to cope with pathological cases, where the vast majority of forward messages would be useless, we enhance our forwarding scheme with a mechanism that monitors its performance and takes corrective actions. Such a situation could arise when our underlying assumptions about spatial similarity are incorrect. For example, suppose we use the spatial forwarding scheme, but the application is irregular and has no spatial locality whatsoever. In this case the forward messages will just increase the network traffic and worsen the cache hit ratio by displacing useful lines.

To address this problem we implement a correction mechanism based on feedback information received from the caches. Specifically, we enhance the L2 cache lines with a *forwarding bit*. When a forwarded line is loaded into the cache, the cache controller sets the forwarding bit. This bit is reset when the processor hits on a forwarded line, which basically means that the forward was used. On the other hand, if a line with the forwarding bit set is invalidated or replaced, that forward was wasted. Such an event must be communicated to the memory controller. If the line was replaced, a replacement hint message is specifically sent to the memory. If the line was invalidated, we piggyback this information on the acknowledgment message generated as a consequence of the invalidation. Upon the reception of such feedback information, the memory controller will disable spatial forwarding for the respective line.

2.3 Hardware Implementation Issues

Concerning the hardware implementation, we are interested in two aspects: (1) the amount of storage overhead, and (2) how the memory access time is affected.

The implementation of spatial forwarding does not require additional storage (except for the bits needed to implement the feedback mechanism). So, from this perspective, spatial forwarding is cheap and hence attractive. On the other hand, the memory access time will suffer some degradation. This is due to the fact that for every read request our method needs to access two directory entries: the entry for the current request and the entry for the stencil line. Specifically, the actions performed by the memory controller upon a read request are the following:

1. Read directory information for the requested line and start reading the requested memory block
2. Read directory information for the stencil line
3. When the memory block is retrieved send the data to the requester
4. Compute the set of destinations for spatial forwarding and forward the data

From the above algorithm we see that spatial forwarding does not need additional memory accesses, but it needs an additional directory access (step 2 of the algorithm). We have to point out that this additional access does not hinder the current operation, but it might delay the next memory requests when such memory requests arrive back to back. If the directory were made of SRAM then the problem would be solved simply because the two directory accesses could be completely overlapped with the memory access (memory

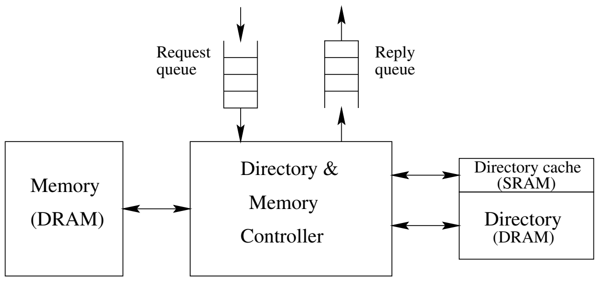

Figure 4: Memory subsystem organization

	Access time	Conditions	Probability
Case 1	0	Stencil line directory bits reside in the same dir. cache entry as the bits for the requested block	7/8
Case 2	1 SRAM access time	Directory bits for the stencil line are present in directory cache	1/8
Case 3	1 DRAM access time	DRAM directory must be read in order to get the stencil line directory bits	0

Table 1: Memory access overhead for spatial forwarding

which is made of slow DRAM). Unfortunately, the large size of directories imposes that they are made of DRAM also. A second possible solution to our problem would be to use a dual ported directory. However, even in this case the size of the directory might render this alternative impractical. The solution we advocate exploits the presence of a directory cache [19] which keeps the most recently used directory entries in a fast SRAM (see Figure 4). In our experiments, each entry in this cache holds directory data for 8 consecutive memory blocks. This value has been chosen according to a recent study by Michael and Nanda [19]. Under these conditions, step 2 of the above algorithm might take different amounts of time depending on the three cases presented in Table 1.

Since the stencil distance matches the degree of memory interleaving (i.e four), the directory data for the stencil line immediately precede the directory data for the current requested line. Thus, on average[1], case 1 will happen roughly with a probability of 7/8. Case 3 should occur very seldom. This is because in regular applications memory lines are usually accessed linearly, and hence there is a very good likelihood that the directory information for the stencil line is present in the previous directory cache entry (case 2). In conclusion, we expect the average delay introduced by spatial forwarding to be around 1/8-th of a directory cache access latency. The actual delay observed in our experiments averages 2.4 processor cycles. Therefore, the gains of spatial forwarding are expected to outweigh by far the small time penalty.

[1]For regular applications is normal to assume a uniform distribution of memory accesses

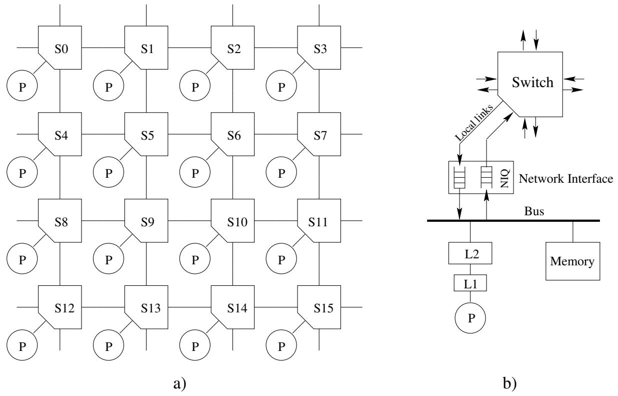

Figure 5: Multiprocessor architecture for simulation

3. EXPERIMENTAL METHODOLOGY

3.1 Hardware Setup and Other Architectural Assumptions

Our simulator is based on RSIM [24] which accurately measures the contention at all subsystems. The architecture we simulate is a shared memory cc-NUMA machine with 16 nodes connected by a 4×4 mesh like in Figure 5(a). Each node is composed of a processor, two levels of write-back cache, a memory module, a network interface and a switch (see Figure 5 (b)). We simulate a state-of-the-art, 600 MHz, 4-way superscalar processor with branch prediction, out-of-order speculative execution, non-blocking reads and release consistency shared memory model [10]. The only simplifying assumption we make is that instructions are considered to always hit in the I-cache of the processor. The instruction window of the processor has 64 entries and the functional units have latencies similar to the UltraSPARC processor.

Both L1 and L2 caches are write back, write allocate, with a line size of 32 bytes. L1 data cache is 16 KB large, two way set-associative and has an access time of one processor cycle. L2 cache is four way set-associative, is fully pipelined and has an access time of 8 processor cycles. Its size, 128 KB, is smaller than one can find in today's machines to account for the rather small problem sizes that we run.

The memory modules are four way interleaved and are distributed across the system in a round robin fashion at page granularity. The size of a page is 4 KB. The memory controller is clocked at 100 MHz and the memory chips are made of 100 MHz buffered SDRAM. The latency to read a whole memory line is 90 ns (54 processor cycles). This value includes the delay introduced by the controller.

Cache coherency is based on a write invalidate directory protocol with three states: Modified, Shared, Invalid (MSI). Each memory module has a full-map directory built out of SDRAM. To improve the directory access time we provide a two way set-associative directory cache [19] with 128 sets. Based on the results published by Michael and Nanda [19] each directory entry holds information for 8 memory blocks. The access time is one memory clock cycle (10 ns).

The bus connecting the various components is a split transaction bus with a width of 128 bits and clocked at 200 MHz. Arbitration is done in a round robin fashion.

The switch at each node is based on our design proposed in [23]. It has 4 input ports, 4 output ports and a pair of local links for the communication with the processing unit. It uses wormhole routing and has four virtual channels at each input port. The input buffer is dynamically allocated using a circular buffer [23] with a total size of 32 flits. A flit has 64 bits. The switch is fully connected which permits us to simultaneously read several flits out of an input port and pass them to the corresponding outputs. The physical links between switches have a width of 32 bits and operate at processor frequency. Link arbitration is based on a first-come-first-served policy implemented with age counters.

3.2 Benchmarks

As benchmarks we use six parallel applications. Four of them are regular dense matrix problems with spatial locality. The other two are irregular, but they are not dynamic, in the sense that the sharing pattern of memory lines remains the same throughout the entire execution of the program. A short description of the benchmarks follows.

FWA computes the shortest path between any two nodes in a directed graph and is based on the well known Floyd-Warshall algorithm [6]. The graph is represented as a $N \times N$ adjacency matrix, where element (i, j) stands for the distance between node i and j. The data are distributed blockwise to processors. The program iterates N times, gradually transforming the initial matrix. During iteration i all processors access their assigned rows, and also read row i. Thus, during each iteration one row is widely shared. The sharing degree is 16 (equal to the number of processors).

GS (Gramm-Schmidt) computes an orthonormal basis for a set of N-dimensional vectors. The vectors are divided between processors in a cyclic manner. In each iteration, the program contains two phases. In the first phase, a processor computes its own pivot vector followed by a barrier. Afterwards, every processor makes all vectors that follow the pivot vector orthogonal to the pivot vector. Hence, the pivot vector is widely shared and the sharing degree is 16.

GAUSS implements Gaussian Elimination with partial pivoting on a system of linear equations stored in a 2-D shared array. The computation on the rows is distributed across the processors in a round-robin fashion in order to maintain a good load balance as elimination proceeds. Similar to GS, the access pattern for the pivot row is single-writer/multiple-reader. However, in contrast to GS, the number of elements in the pivot row touched by processors decreases gradually due to the triangular nature of the algorithm.

TC computes the transitive closure of a directed graph using Warshall's algorithm. The connectivity is given by a $N \times N$ boolean matrix. A "1" for element (i, j) means that there is an arc from node i to node j. The program iterates N times and during every iteration one row is widely shared. The sharing degree is again equal to the number of processors (16).

EM3D [7] models the propagation of electromagnetic waves through objects in three dimensions. After a preprocessing step, the problem is transformed into a simple computation on an irregular bipartite graph containing nodes representing electric (E) and magnetic (H) field values. The computation consists of a series of iterations. During odd iterations changes in the electric field are calculated as a linear function of the neighboring magnetic field values. Specifically, the value of each E node is updated by a weighted sum of neighboring H nodes. Similarly, during even iterations the H nodes are updated using the E nodes. Each E or H node is shared by a certain number of processors. Due to the irregular nature of the problem the sharing degree varies from node to node. Our measurements indicate an average sharing degree of 5.1.

NBF (the Non-Bonded-Force kernel) is a molecular dynamics simulation taken from the GROMOS benchmark [12]. Its computational structure resembles the non-bonded force calculation in CHARMM [4], which is a well-known molecular dynamics code. Non-bonded forces are long-range interactions existing between each pair of molecules. NBF approximates the non-bonded calculation by ignoring all pairs of interacting partners which are beyond a certain cutoff radius. The cutoff approximation is achieved by maintaining a list of interacting partners for each molecule and iterating over this list at each time step. For each molecule, the program goes through the list of partners, and updates the forces on both a molecule and its partner based on the distance between them. In our experiments, the partner list is static. Each molecule has approximately the same number of partners (20), and the partners of each molecule spread evenly in about 2/3 of the total space. Reading the data from the partners results in widely shared data distribution. We measured an average sharing degree of 11.6. Updates to the forces are accumulated in private memory. After this computation, the processors update the shared forces in a pipelined fashion. The update is performed in *nprocs* steps. In each step, a processor updates $1/nprocs$ of the total data. The sharing pattern of this phase is migratory.

The input data size of the benchmarks and other relevant characteristics are shown in Table 2. To keep the simulation time within reasonable limits the problem sizes that we run are rather small. To compensate for this we shrink the L2 cache accordingly. Statistics have been collected only for the parallel part of the applications.

Application name	Input Size	Read/Write mem. req. (x 1000)	Overall miss rate(%)
FWA	128 node graph	41 / 6	0.68
GS	128x128 matrix	64 / 2.5	0.69
GAUSS	128x128 matrix	24 / 4	0.55
TC	128 node graph	26 / 5	0.54
EM3D	2048 nodes, 20% local	319 / 78	20.5
NBF	16386 molecules	320 / 181	7.04

Table 2: Characteristics of applications used in simulations. The overall cache miss rate is computed by dividing the number of requests that miss in L2 by the total number of processor requests.

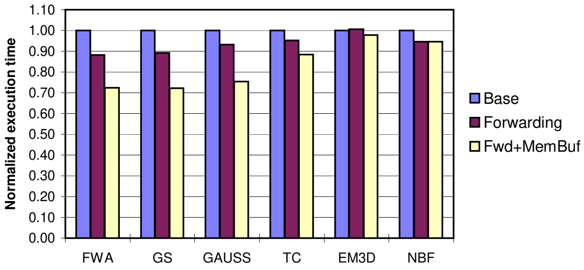

Figure 6: Normalized execution time for spatial forwarding. Feedback is turned on.

4. EXPERIMENTAL RESULTS

4.1 Performance Metrics

The most relevant metric in any performance evaluation is execution time and thus, this will be our main metric. The execution time of various schemes is always normalized to the base system where no enhancement is provided.

To evaluate the forwarding accuracy we will use a metric called *forwarding effectiveness*. In essence this shows how the forwarding messages were used. We identify three categories: wasted, used-on-time and used-late. A forwarding message is *wasted* if the corresponding forwarded memory line is either replaced or invalidated from the cache before it has had a chance to be used. When a processor hits on a forwarded cache line we say that the forward was *used* and *on-time*. However, if at the moment the forwarded line arrives at the cache, the processor has already issued a request, the forward is still used, but is *late*. In this case the memory latency is only partially hidden. For forwarding effectiveness, the number of forward messages is always normalized to the number of read requests of the base architecture. We do this to reveal how intense the forwarding activity is.

4.2 Performance of Spatial Forwarding

In Figure 6 we present the normalized execution time of spatial forwarding for our six applications. Note that the feedback mechanism described in Section 2.2 is enabled. From the graph we see a good improvement for regular applications that have widely shared data. FWA and GS exhibit reductions in execution time of 12% and 11% respectively. GAUSS and TC have more modest improvements. For GAUSS this happens because the length of the rows that are widely shared decreases as the application progresses. Consequently, towards the end of the run, there is not sufficient locality for the forwarding mechanism to work properly. For TC the main culprit is the forwarding effectiveness as explained next.

To gain more insight on how accurate the forwarding mechanism is, let us look at the forwarding effectiveness plotted in Figure 7. First, we observe that regular applications issue numerous forward requests. Also, we can see that the number of wasted forwards is very small (only 7% for FWA, 3% for TC and negligible for GS and GAUSS). About one third of the forwarded messages arrive on-time (only 25% for TC). However, a large fraction of the messages being used are late

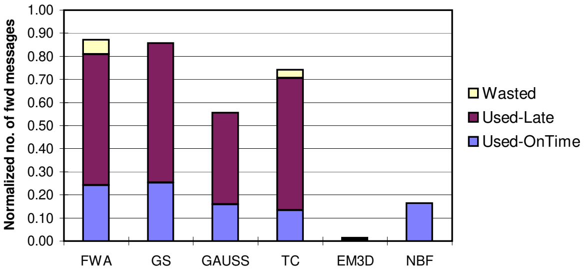

Figure 7: Forwarding effectiveness. Normalization is done with respect to the number of read messages of the base architecture.

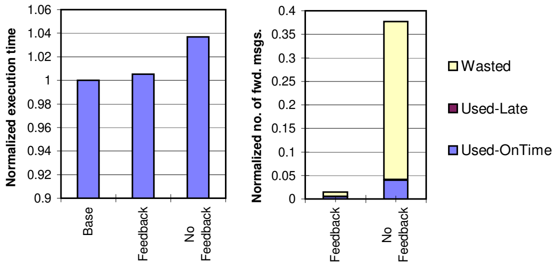

Figure 8: The effect of the feedback mechanism on spatial forwarding - EM3D

due to processor synchronicity. In the next section we will show the performance of a method that accelerates the triggering moment for forwards in an attempt to minimize the number of late forwards.

The eight entry data buffer included in the memory controller boosts the performance of the four regular applications (see Figure 6). The reason is that the buffer serves the "slow" read sharers that couldn't be captured by the forwarding mechanism. Another reason stems from the large number of late forward messages. If a forward is late, it means that the processor has already issued a read request for the respective line. This read request will only waste memory bandwidth because the cache miss will be satisfied by the forward message anyway. The small buffer that we added at the memory alleviates this problem by servicing the superfluous read requests directly and improving the average memory latency.

We also wanted to see the behavior of our forwarding scheme when applied to irregular applications. Thus, we included in our benchmark set two irregular applications. From the start, such applications are not good candidates for our spatial forwarding mechanism: Even if they have widely shared data, the sharing pattern of adjacent memory lines is expected not to be identical. Nevertheless, as seen in Figure 6, spatial forwarding manages to improve the execution time for NBF. This somewhat surprising behavior is due to the large sharing degree for the memory lines (11.6 on average). With a large degree of sharing, all memory lines look, more or less, the same in terms of sharing pattern. This fact is confirmed by Figure 7 which shows that the number of wasted forwards is almost null.

A different situation occurs for EM3D. Here, the average sharing degree is relatively low (5.1) and the true behavior of an irregular application is revealed. In the beginning, our spatial mechanism has a high miss-prediction rate and lots of forward messages are wasted. However, the feedback mechanism senses this anomalous behavior, and turns off forwarding for the rest of the iterations. Thus, only a small performance degradation is observed for EM3D. To better illustrate the effect of the feedback we plot in Figure 8 the execution time and the forwarding effectiveness of EM3D with and without the feedback mechanism. We can see that the feedback mechanism is definitely helpful since it limits the possible slowdown for irregular applications.

Application	Average sharing degree	Observed average sharing degree
FWA	16	12.2
GS	16	13.9
GAUSS	16	8.0
TC	16	11.8
EM3D	5.1	2.1
NBF	11.6	3.6

Table 3: Comparison between the average sharing degree exhibited by applications and the sharing degree observed by the forwarding mechanism.

Another factor that can influence the performance of spatial forwarding is the observed sharing degree. Basically, this represents the number of forward messages that can be sent with just one memory read access. Of course, we would like this number to be as large as possible. Table 3 shows that, for regular applications, the observed sharing degree is close to the real sharing degree. In contrast, for irregular applications there is a serious discrepancy between the two numbers. This fact might be in our advantage because it would limit the number of forward messages that would be inadvertently sent to wrong destinations.

In Section 2.3 we explained why the memory access overhead introduced by spatial forwarding is expected to be low. We measured this overhead and the results are depicted in Figure 9. As one can see, in the majority of the situations there is no overhead for accessing the directory for the stencil line (case 1). Also, the number of accesses to the DRAM directory is negligible (case 3). Case 2, when the stencil data is retrieved from the directory cache, happens with a higher

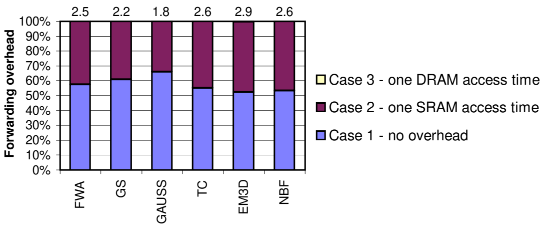

Figure 9: Distribution of the memory access overhead sources for spatial forwarding. The numbers above the columns represent the average forwarding overhead (in processor cycles) measured in our experiments.

frequency than what we theoretically estimated. Nevertheless, the observed average overhead (2.4 processor cycles) can be regarded as small when compared to the memory latency.

4.3 Triggering the Forwarding Mechanism Earlier: Combining Spatial Forwarding with Software Prefetching

Spatial forwarding is triggered by the first read request that arrives at the memory for a widely shared line. There are two disadvantages with this approach. First, the leader of a sharing group still incurs a full miss penalty. Second, and most importantly, due to the fact that processors work quasi-synchronous, all processors in a sharing group issue their requests for a line at about the same time, leading to a large number of late forwards. To reduce these drawbacks we propose to use prefetch requests as triggers for the forwarding mechanism. A prefetch request is issued by the processor before the request for the desired data, and therefore it creates the premises for hiding the latency of all read requests. In this study we chose to use software prefetching due to its increased accuracy. We employed this technique solely for regular applications for two reasons:

1. While software prefetching has been proven to be successful when applied to regular scientific code, for irregular applications there is no established technique to insert prefetch instructions.
2. For irregular applications, the processors' accesses to a memory block are temporally scattered rather than clustered together. This will lead to a very small number of forwards that are late. In contrast, for regular applications a large number of forwards do not arrive on time (due to processor synchronicity). Triggering the forwards on a prefetch instead of the first request, could alleviate this shortcoming.

In conclusion, software prefetching for regular applications is at the same time simpler and has the potential of greater returns. We insert prefetch instructions by hand using loop peeling, unrolling and software pipelining according to the methodology developed by Mowry [21] A prefetch request for a memory line triggers the forwarding mechanism at the memory controller.

The results presented in Figure 10 demonstrate that the hardware/software combination is successful. While prefetching alone brings an improvement slightly less than forwarding, their combination reduces the execution time by 6 to 32%, and if we add the memory buffer, by 10 to 37%. This suggests that the two methods complement each other very well and they should be used together wherever possible. Also, the graph indicates that the small memory buffer helps much more the forwarding scheme than prefetching.

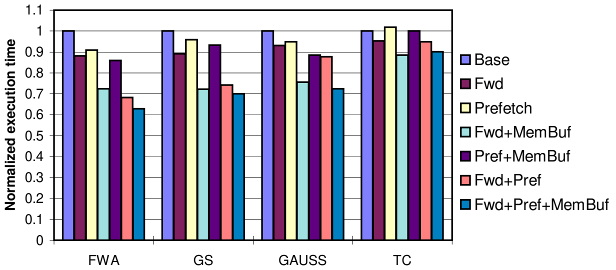

Figure 10: Performance of spatial forwarding plus software prefetching

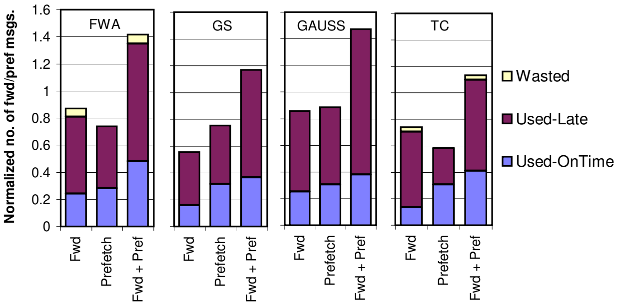

Figure 11: Forwarding effectiveness of spatial forwarding plus software prefetching

If we look at the forwarding effectiveness depicted in Figure 11 we notice that the addition of prefetching indeed increases the number of forwards that arrive on-time. Note that the figure does not make a distinction between forwards and prefetches. This explains why the normalized number of forward messages exceeds 100% for the third bar.

Interestingly, even though prefetching has a better efficiency, spatial forwarding has been shown to obtain a better performance. As explained in the introduction of the paper, three factors can contribute to this behavior:

- Prefetching consumes tag-cache bandwidth while forwarding does not.
- Spatial forwarding reduces the average memory latency because only one memory access is needed to send a group of forwards.
- Spatial forwarding usually reduces the network traffic while prefetching doesn't.

The last assertion is directly confirmed by our results presented in Figure 12. As usual, the normalization has been done with respect to the base architecture.

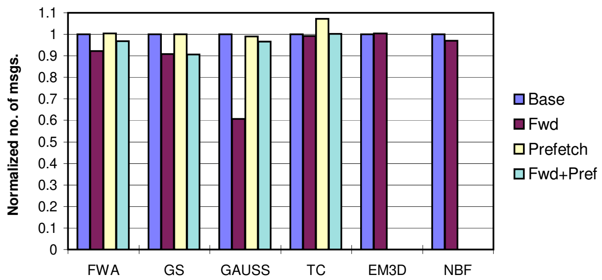

Figure 12: Normalized number of messages in the network

5. CONCLUSIONS

In this paper we presented a forwarding mechanism placed at the memory controller, that is able to improve the performance of applications with widely shared data. This mechanism anticipates the read requests coming from processors and pushes the desired data ahead of time into the processor caches.

Our method exploits the sharing pattern similarities of adjacent memory blocks. Thus, it is applicable to regular applications where data are accessed in a linear fashion. Our results show that significant reductions in execution time are possible. A small 8 entry buffer placed at the memory is capable of boosting the applications' performance even more (up to 29% over the base) due to improved average memory access time.

The prediction accuracy of spatial forwarding is kept high by preventing forwarding errors and at the same time taking corrective actions in case of errors. The prevention part is done with simple filters that detect some of the situations where the spatial locality is disrupted. The forwarding miss-predictions are corrected by employing a feedback mechanism from the caches. We showed that, to preclude serious slowdowns in certain circumstances, feedback is a required add-on. The number of forward messages that were wasted is very low. However, many messages are late which is probably the main limitation of this method. To overcome this drawback we proposed to trigger the forward mechanism earlier by using prefetch requests. This combined software-prefetching/hardware-forwarding approach reduced the execution time even more (up to 37%).

Also, another notable feature of our design is that it incurs a minimal overhead in terms of both memory access time and storage (storage is needed only for the feedback mechanism).

6. REFERENCES

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Third International Symposium on High-Performance Computer Architecture*, pages 204–215, Feb. 1997.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91*, pages 176–186, 1991.

[3] D. Bhandarkar and J. Ding. Performance characterization of the Pentium Pro processor. In *Third International Symposium on High-Performance Computer Architecture*, pages 288–297, Feb. 1997.

[4] B. Brooks, R. Bruccoleri, B. Olafson, D. S. es, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[5] W. Chen, S. Mahlke, P. Chang, and W.-M. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th International Symposium on microarchitecture*, 1991.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[7] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parrallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Nov. 1993.

[8] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the 25th MICRO*, pages 102–110, 1992.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, Apr. 8–11, 1991.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Henessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th ISCA*, pages 15–26, June 1990.

[11] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing a MIMD, shared-memory parallel machine. *IEEE Trans. on Computers*, 32(2):175, Feb. 1983.

[12] W. Gunsteren and H. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, 1988.

[13] R. Iyer and L. Bhuyan. Switch cache: A framework for improving the remote memory access latency of CC-NUMA multiprocessors. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 288–297, Jan. 1999.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[15] S. Kaxiras and J. R. Goodman. The GLOW cache coherence protocol extensions for widely shared data. In *Conference Proceedings, 1996 International Conference on Supercomputing*, pages 35–43, May 25–28, 1996.

[16] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1250–1264, Dec. 1996.

[17] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–105, May 1992.

[18] C.-K. Luk and T. C. Mowry. Compile-based prefetching for recursive data structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[19] M. M. Michael and A. K. Nanda. Design and performance of directory caches for scalable shared memory multiprocessors. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 142–151, Jan. 1999.

[20] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[21] T. C. Mowry. Tolerating latency through software-controlled data prefetching. Thesis Technical Report STAN//CSL-TR-94-628, Stanford University, Computer Systems Laboratory, June 1994. [Adminitrivia V1/Prg/19950109].

[22] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[23] N. Ni, M. Pirvu, and L. Bhuyan. Circular buffered switch design with wormhole routing and virtual channels. In *Proceedings of the 1998 IEEE International Conference on Computer Design*, pages 466–473, October 1998.

[24] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[25] S. S. Pinter and A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *Proceedings of the 29th MICRO*, pages 214–225, 1996.

[26] D. K. Poulsen and P.-C. Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *Proceedings of the 23rd International Conference on Parallel Processing. Volume 2: Software*, pages 276–280, Aug. 1994.

[27] A. J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, Dec. 1978.

[28] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Apr. 8–11, 1991.

[29] D. M. Tullsen and S. J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 278–288, May 1993.

[30] K. M. Wilson and K. Olukotun. Designing high bandwidth on-chip caches. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 121–132, June 2–4 1997.

[31] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing cache port efficiency for dynamic superscalar microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 147–157, May 22–24, 1996.