Adaptive Cache Coherence Protocol Using Migratory Shared Data

Divya Jhalani (divyajhalani@gmail.com)

Diana Palsetia (palsetia@gmail.com)

Department of Electrical and Computer Engineering

University of Wisconsin, Madison

Abstract

A traditional approach to connect several processors having private caches is achieved by interconnection network. To maintain consistent view of the shared memory for each processor, coherence protocols are employed. As the architecture scales, the interconnection network becomes a bottleneck due to coherence-induced traffic. One way to reduce this traffic is to study and employ data access patterns in the coherence protocols. To capture such access patterns during program execution, the coherence schemes must be adaptive in nature. The goal of this paper is to investigate one of the adaptive schemes that detects access pattern of shared data called migratory data. Migratory data sharing is a distinct sharing pattern exhibited by parallel programs where the data moves between different processors, and at each processor the data is in exclusive state.

1. Introduction

In multiprocessor systems where separate processors logically share a common memory, it is necessary to keep their caches in a state of coherence by ensuring that any shared data that is changed in one cache is changed throughout the system. The two main protocol categories for maintaining updated copies upon a write operation are Write Update (WU) and Write Invalidate (WI). However, as the architecture scales, the bus becomes a bottleneck due to coherence-induced traffic. Studies have shown that coherence protocol performance is dependent upon the access patterns exhibited during execution of the program [4]. These access patterns can be utilized in the protocols to minimize the bus transactions.

To analyze these data access patterns, various statistics can be maintained during program execution. Two of the most common metrics are: the number of consecutive write operations to a memory location by a processor (called write-run), and the number of consecutive read operations from a memory location [4]. However, WU and WI protocols do not make use of these access patterns on a per application basis. WU protocol works best when there are many "readers" in the system resulting in read-runs. On the other hand, WI protocol works best when there are many "writers" in the system (or write-runs). Based on the collected statistics, we can select between these two strategies off-line for each application. This scheme is known as Hybrid (HY) coherence protocol [5].

An even stronger approach would be to modify the basic protocol behavior dynamically to manage coherence operations based on data access patterns. This approach is called Adaptive Hybrid (AHY) protocol [5]. In AHY protocols, protocols adapt to the change in access pattern of data during program execution. There have been many adaptive schemes that monitor the read-write pattern of a block of data and perform coherence actions accordingly. In this project, we study and evaluate an AHY protocol based on Migratory Sharing.

The AHY protocol with Migratory Sharing access pattern is proposed and implemented using directory coherence protocol in [3]. For convenience, we refer to this protocol as Migratory Adaptive Protocol throughout this report. Migratory data sharing is a distinct sharing pattern exhibited by parallel programs where the data moves between different processors, and at each processor the data is in exclusive state. In other words, the data is read and then written to immediately afterwards during a sequence of time intervals. Such a pattern arises due to common data-sharing programming techniques such as locks and monitors.

The following section explains how to detect migratory data. In Section 3, we explain the selection strategy of the base protocol on which Migratory Adaptive Protocol is built upon. Section 4 provides information about the

architecture model and the implementation of this adaptive protocol. In Section 5 we present results and compare adaptive protocol with conventional protocols like MSI and MESI. This is followed by an analysis of benefits of the adaptive protocol in Section 6. Finally, Section 7 presents the conclusion of our findings.

2. Detection of Migratory Shared Data

Data is detected to be migratory if one of these two conditions holds [3]:

a) At the time of a write-hit, a block appears in a Shared state with exactly two cached copies and the processor currently requesting the invalidation operation (current invalidator) is not the same as processor that invalidated it previously (previous invalidator).

b) At the time of write-miss on a block for which there is a single cached copy.

Migratory Adaptive Protocol uses the above two tests to determine migratory data pattern during an application execution. If the data block is migratory then the protocol expects that the block will be modified at every processor that it visits. Therefore, if a block is not modified before it moves to another processor, then the protocol sets the migratory bit of the block to false. Table 1 provides some example situations when the migratory bit is set or cleared.

The authors in [3] state that this protocol can quickly adapt to the changes in access pattern based on a set of events as evidence or several successive occurrences of these events (hysteresis). Although hysteresis is more accurate, its implementation further expands the size of the directory entry. Therefore in the implementation of the adaptive protocol, for every memory block they use only one status bit called migratory which is set or cleared based on set of events.

Examples when migratory bit is set		Examples when migratory bit is cleared	
P1 Read P1 Write P2 Read P2 Write M = 1	P1 Read P1 Write P2 Write M = 1	if (M == 1) P1 Read P2 Write M = 0	if (M == 1) P1 Read P2 Read M = 0

Table 1: Example situations of migratory bit modifications

3. Selection of a Base Protocol for Migratory Sharing Pattern

In write-update protocol, when a processor writes to a shared data, it sends the new data to all the sharers. In response all the sharers update their copies of the data. On the other hand, in the write-invalidate protocol, write to a shared data entails the processor sending an invalidation message to all the processors and becomes an exclusive owner of the data. Since migratory data entails writes by different processors, write invalidate protocol is a better candidate for this data sharing pattern.

A write invalidate protocol is optimized by building the migratory adaptive cache coherence protocol on top of it. This optimized version is used whenever migratory sharing is detected. On other patterns, a regular version of invalidate protocol is used. The advantage of using the optimized version is that for a migratory data pattern, it uses equal or less coherence messages over the regular version. For example, assume that a data held by processor P_i is migratory. In regular write-invalidate protocol, a read miss by another processor P_j will create a new copy of the block in P_j's cache (*replicate-on-read-miss*) and a shared coherence message will be sent to P_i. Suppose P_j now wants to write to the same data block, then an invalidate coherence message will be sent to P_i. On the other hand, for optimized version, a read miss will use a policy called *migrate-on-read-miss*, that is, it will copy the block to P_j's cache and at the same time invalidate the P_i's copy. The *migrate-on-read-miss* policy thereby halves the number of inter-cache operations used to move the migratory block.

If there is no migratory data present in an application, the optimized version will not be used. In this case, the regular version would preside throughout the application execution. Therefore, the number of coherence messages will not improve. This performance dependence on applications is visible in our results section.

4. Implementation of Migratory Adaptive Protocol

4.1 Architectural Model

To implement the Migratory Adaptive Protocol, we used the RSIM simulator [1] to perform execution-driven simulation. RSIM simulates shared-memory multiprocessor system using CC-NUMA system with directory-based coherence (Figure 1). Each processor in the system consists of a two level cache hierarchy, a portion of the system's distributed physical memory and its associated directory, and a network interface (a two-dimensional mesh network). A pipelined split-transaction bus is used to connect the secondary cache, the memory and directory modules, and the network interface. Local communication within the node takes place on the bus. The network

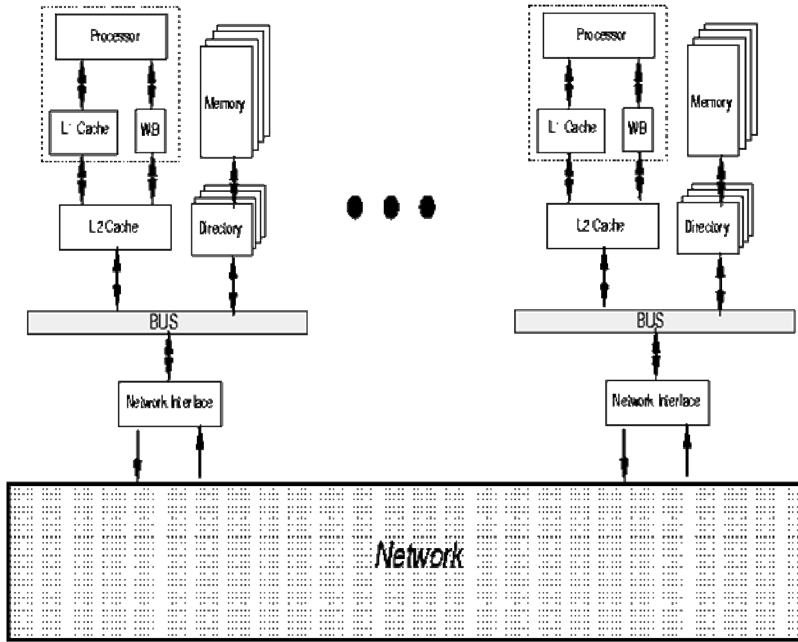
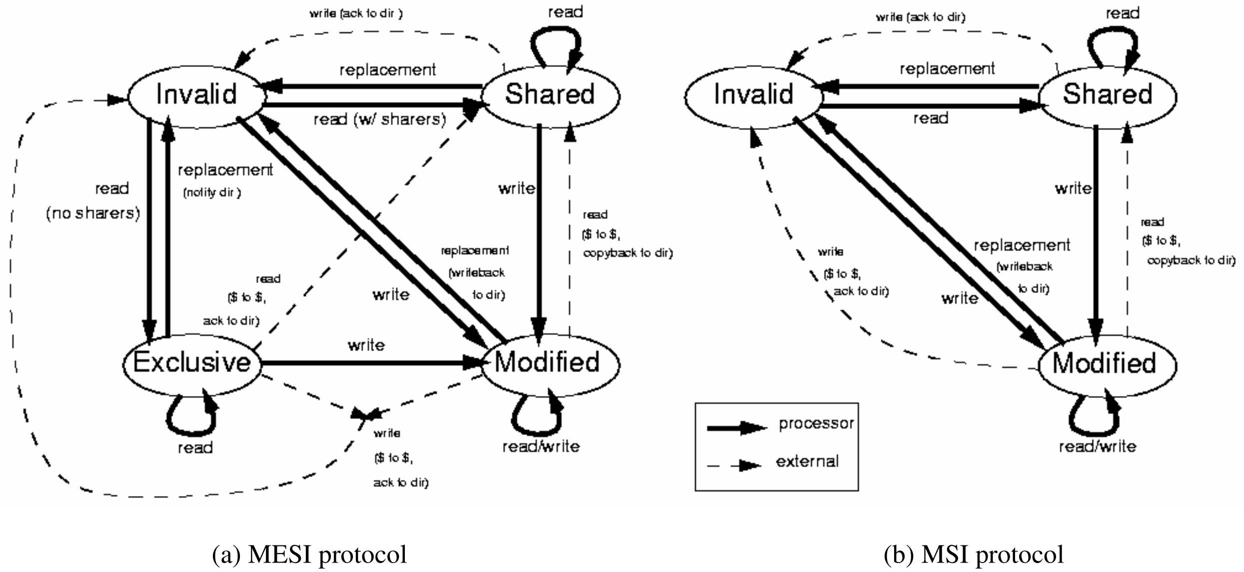


Figure 1: RSIM Memory System [RSIM]

interface connects the node to a multiprocessor interconnection network for remote communication. The memory system supports both MSI and MESI cache coherence protocols along with support for sequential consistency.

The L1 and L2 cache sizes used for all the simulations in this paper are: 32KB (2-way associative) and 1MB (4-way associative) respectively. Furthermore, L1 cache is write through with no allocate policy. L2 cache is write back with write allocate policy. Maximum number of processors supported by RSIM is 128. These processors are superscalar with configurable pipeline width. For our simulations we used a pipeline width of four. Memory is accessed in parallel with an interleaved directory. The directory implements a full map cache coherence protocol, i.e. sparse-grained, with one bit assigned to each processor.

Each directory line can be in three possible states: Uncached, Private or Shared. The Uncached state corresponds to the *invalid* state in MESI and MSI protocols. For MESI protocol, both *exclusive* and *modified* states correspond to Private in RSIM implementation. However, this does not affect the MESI protocol state transitions. As shown in Figure 2 (a), both *exclusive* and *modified* states have the same transitions, and hence can be fused together. Since there is no distinction between clean and dirty data blocks, on a transition from Private to Shared,



(a) MESI protocol (b) MSI protocol

Figure 2: State diagrams for MESI and MSI protocols (Source: [1])

RSIM just does a cache-to-cache transfer of the data instead of getting the potentially stale data from the memory. Figure 2 (b) shows the state diagram for MSI protocol. For MSI, Private state in RSIM implementation corresponds to *modified* state.

4.2 Directory-based Implementation

We have implemented the Migratory Adaptive Protocol using MESI as the base protocol. To the existing directory line states we add three new states to distinguish between migratory states and regular states. These are un-cached migratory (Uncached_M), private migratory (Private_M), and 2-processor sharing (Shared_Two). Furthermore, each directory entry also has two status bits added to it, i.e., *migratory* and *dirty* bits. The *migratory* bit is used to determine if a block of data is migratory. It can be set or cleared based on the migratory data pattern. The *dirty* bit is to identify whether a block in Private state has been modified or not. In the original directory implementation of RSIM, for a cache line in exclusive state there is no distinction between it being dirty or clean (it is always assumed to be dirty). However, for the adaptive protocol it is important to know if the data held exclusively was modified or not. Therefore *dirty* bit is important so that unmodified data does not get categorized as migratory data. We also add another status field for every directory line called *last invalidator*. This remembers the last processor, which invalidated other processor(s) on a write request for that block of data.

In Figure 3 we present the pseudo-code of the Migratory Adaptive Protocol [3]. This code shows transitions between different directory line states based on the request type for the data block.

```
Read  miss:
     switch(state)
          case Uncached:
               state = Private;
          case Uncached_M:
               state = Private_M;
          case Private:
               state = Shared_Two;
          case Private_M:
               if(dirty == false) then
                    state = Shared_Two;
                    migration = false;
          case Shared_Two:
               state = Shared;
          case Shared:
               do nothing;

     if state = Private_M  then
          migrate the block;
     else
          migratory = false;
          replicate the block;

write miss invalidating one or more copies of a block:
     if state = Private_M then
          if (dirty == false) then
               state = Private;
               migration = false;
     else if (state = Private && last invalidator != current processor){
          if(migratory == true)
               state = Private_M;
          else migratory = true;
     }
     else state = Private;
     last invalidator = current processor;
     migrate the block;

write hit invalidating one or more copies of a block:
     if (state = Shared_Two && last invalidator != current processor){
          if(migratory == true)
               state = Private_M;
          else migratory = true;
     }
     else state = Private;
          migration = false;
     last invalidator = current processor;
     perform invalidations;

write hit on a clean, exclusively-held block:
     if (state = Private && last invalidator != current processor){
          if(migratory == true)
               state = Private_M;
          else migratory = true;
     }
     last invalidator = current processor;
```

Figure 3: Pseudo-code for Migratory Adaptive Sharing

Benchmark	Description	Input Size
LU	Parallel dense blocked LU factorization	256×256 matrix
MP3D	Particle flow in simulated wind tunnel	50,000 particles
Radix	Integer radix sort program	524,288 keys
FFT	1-D fast Fourier transform	2^{16} complex data points
Water	Molecular dynamics simulator	512 molecules

Table 2: Benchmark characterizations [Source: [6], [7]]

5. Evaluation

5.1 Simulation

We simulated five SPLASH benchmarks: LU, MP3D, Radix, FFT and Water. Their descriptions are given in Table 2, along with the input sizes we used in our simulations. We simulated these benchmarks over 4, 8 and 16 processor system. Each benchmark was run using MSI, MESI and Migratory Adaptive Sharing protocols. The number of bus transactions was computed for every case. A bus transaction is defined as the following operations:

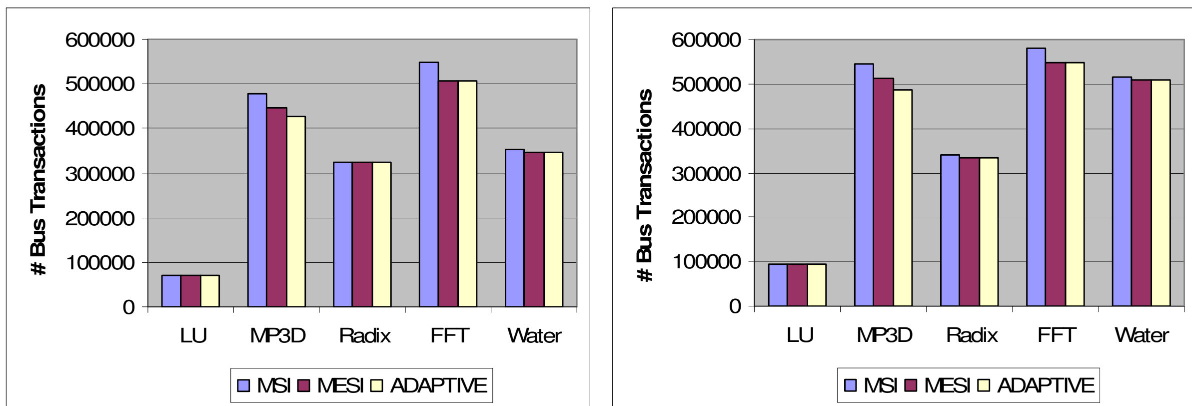
1) Data movement from memory to cache in case the line is un-cached.
2) Cache-to-cache transfer (data replication) where one processor copies the data to another processor's cache.
3) Data migration where data is transferred from one processor to another and the directory invalidates the supplier.
4) Invalidation due to cache evictions and bus writes.

According to [3], the relative effectiveness of an adaptive protocol increases with larger caches. This is because larger caches reduce the number of capacity and conflict misses, which, in turn, reduces the amount of writebacks. Moreover, migratory access pattern is better exploited in larger caches due to higher probability of data migrations between caches. Therefore, for this project, we chose reasonable large caches: L1 was 32KB (2-way associative) and L2 was 1MB (4-way associative). The block sizes were 32 bytes for both the caches. This cache configuration enabled us to extract relatively more migratory behavior in the applications. However, as we will see in the following sub-sections, Migratory Adaptive Protocol is still not optimal for all the applications. It fails to

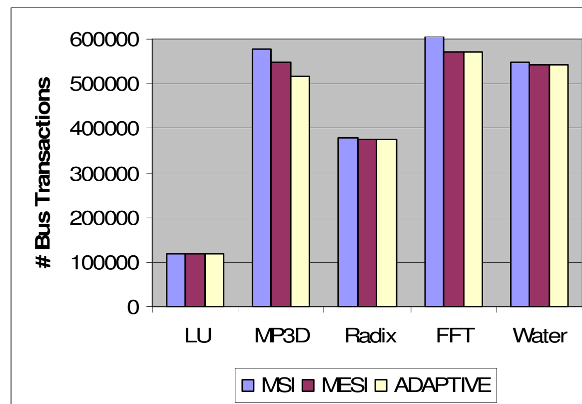
work for those which do not exhibit any migratory access patterns, and sometimes even produces negative results in terms of performance increase.

5.2 Comparison of the Cache Coherence Protocols

Figure 4 shows the number of bus transactions for the five benchmarks implemented with MSI, MESI and Migratory Adaptive schemes over 4, 8 and 16 processor system. As the number of processors is increased, the number of bus transactions also increases. This is because with increasing number of processors, the number of coherence messages also increases. However, the amount of increase is application dependent. For instance, Water performs approximate 1.5 times worse when scaled from 4 processors to 8. On the other hand, Radix performs only 1.02 times worse for the same scaling.



(a) 4 Processors (b) 8 Processors

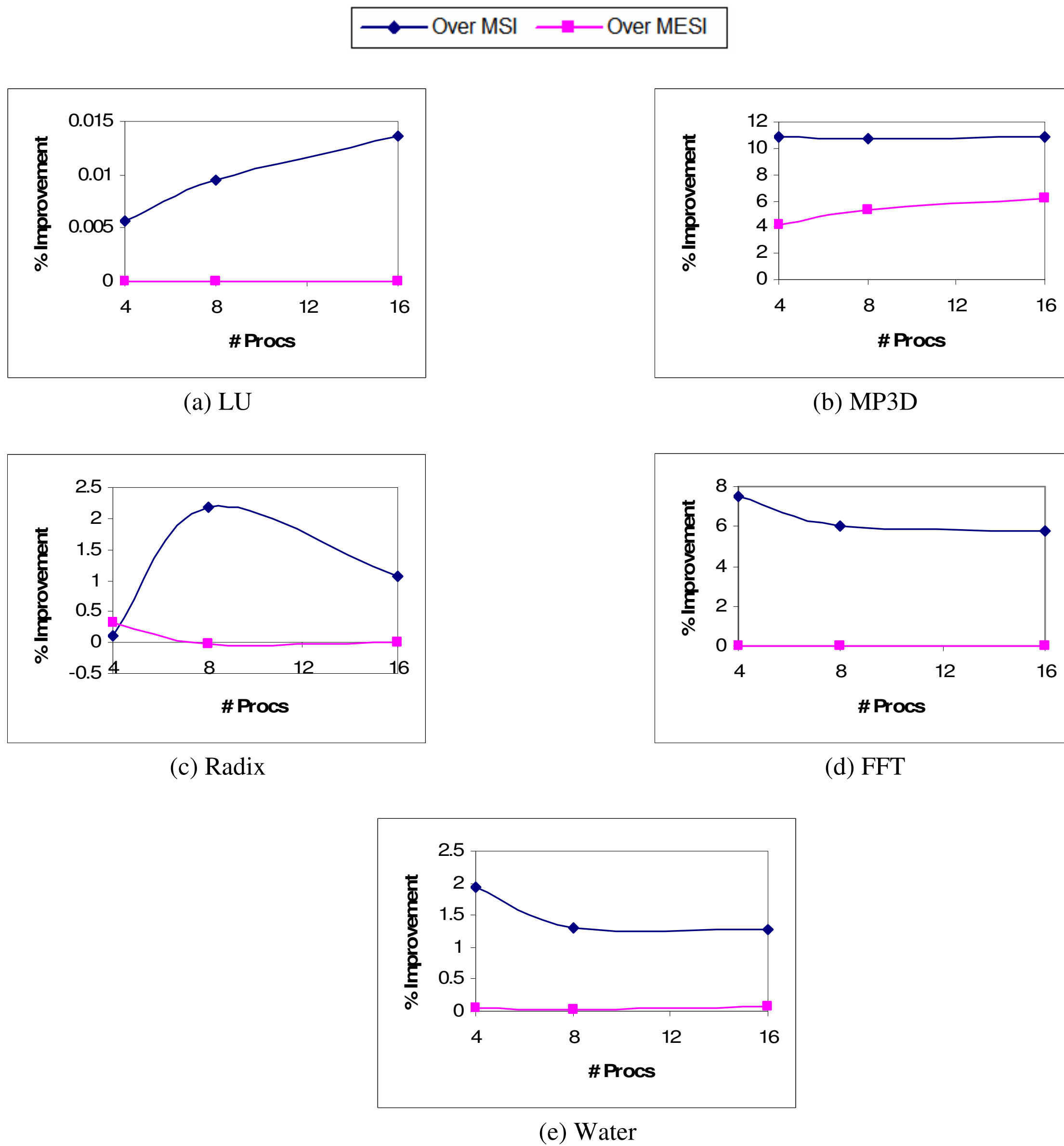


(c) 16 Processors

Figure 4: MSI, MESI and Migratory Adaptive Protocols for 4, 8 and 16 processor system

Furthermore, the Migratory Adaptive Protocol performs almost equal or better than the other protocols. Since this graph has bus transactions in units of hundreds of thousands, it is hard to notice any small deviations of the adaptive protocol over MSI or MESI. Radix actually performs worse with adaptive protocol when compared to MESI. This behavior is further explored in the next section.

5.3 Improvement of Migratory Adaptive Protocol



(a) LU

(b) MP3D

(c) Radix

(d) FFT

(e) Water

Figure 5: Improvement of Migratory Adaptive Protocol over MESI and MSI for various benchmarks

In this section we compare the performance of the adaptive protocol over MSI and MESI. This is done by calculating the percent improvement of the adaptive protocol using the following equation:

$$\text{Percent improvement of N over B} = \left(1 - \frac{T_N}{T_B}\right) \times 100\,\%$$

where T_N is the number of bus transactions for algorithm N and T_B is the number of bus transactions of algorithm B.

In Figure 5 we show the percent improvement graph for each benchmark. For all the benchmarks the adaptive protocol is better than MSI. However, the degree of improvement varies greatly from benchmark to benchmark. For instance, MP3D has over 10 % improvement for all the processor systems, whereas LU has only 0.006 % improvement for four processor system. In the case of MESI protocol, LU, Water and FFT have close to 0 % improvement for adaptive protocol. For MP3D, the improvement is as high as 6 % for 16 processor system. However, in the case of Radix the adaptive protocol performance deteriorates to a negative value.

We speculate that this is because radix is a sort program, and it needs to exchange the values between the processors, thereby resulting in some writes and many read operations. Adaptive protocol tries to categorize data as migratory whenever possible. However, this strategy may become too aggressive for some applications like Radix, and cause unnecessary invalidations. In those cases, MESI would outperform migratory adaptive protocol. This behavior is further explained in the following example.

Suppose processor P_1 has exclusive access to some data and the data has been classified as migratory from its past access pattern. Then processor P_2 reads the same data followed by P_1.

Operation	Directory line Current State	Directory line Next State	Bus Action
P1	Private	-	-
P2 read	Private	Shared	data replication
P1 read	Shared	Shared	-
P2 read	Shared	Shared	-

Table 3: MESI example

Operation	Directory line Current State	Directory line Next State	Bus Action
P1 (data is migratory)	Private_M	-	-
P2 read	Private_M	Private_M	data migration
P1 read (migratory is clear)	Private_M	Shared	data replication
P2 read	Shared	Shared	-

Table 4: Migratory Adaptive example

In the case of MESI, Table 3 shows that there is only one bus action needed. However, in migratory adaptive protocol (Table 4), two bus actions are required: the first one for data migration where P_2 becomes the exclusive owner of the data; the second one for data replication where the data is shared by both P_1 and P_2. The extra bus transaction happens because the data was wrongly classified as migratory and was migrated to P_2 even though P_2 was not going to write to it.

The adaptive protocol proposed in [3] uses only 1-bit history vector (*migratory* bit). From the example, we can speculate that if we were to use a longer history vector we would be able to better capture the migratory data access patterns. In other words, hysteresis would be better since we would not wrongly classify the data as migratory based on just the last access pattern. We would instead look at multiple past access patterns of the data.

5.4 Ratio of Migrations to Total Bus Transactions

Figure 6 below graphs the percentage of data migrations in the overall bus transactions for every benchmark. LU and FFT both have 0 % migrations. This result reinstates the results from previous sub-section. Both of the benchmarks had 0 % improvement over MESI (Figure 5). Furthermore, MP3D and Water show a comparatively

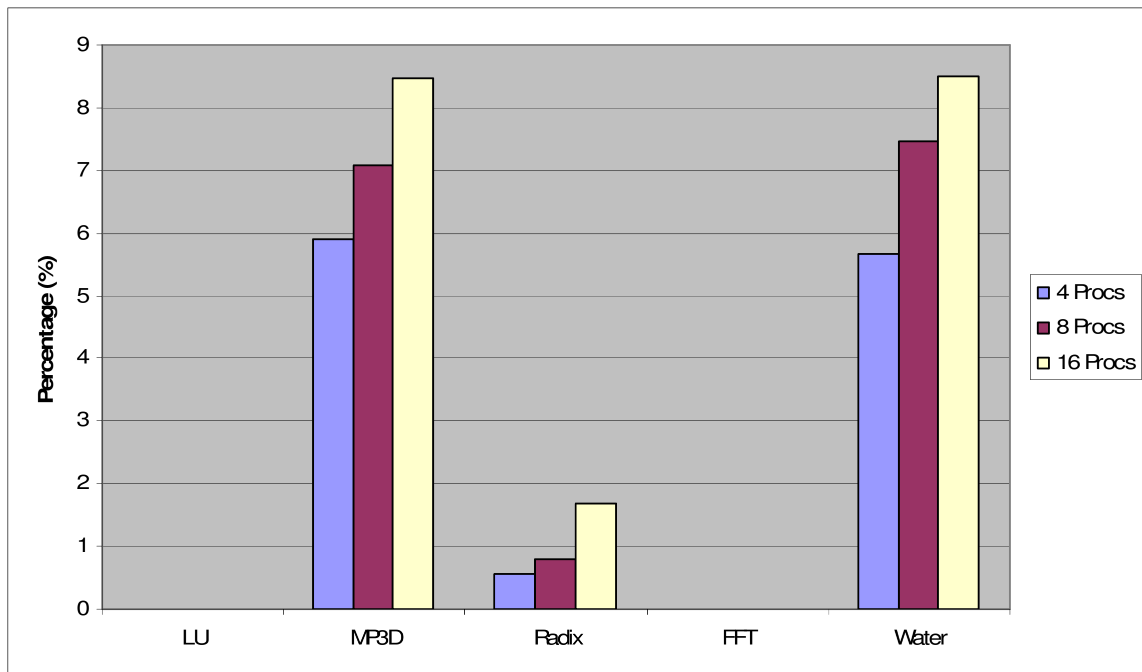


Figure 6: Percentage of Migrations out of total Bus Transactions

large ratio of migrations in their bus transactions. Finally, Radix has a small percentage of migrations. As stated previously, these migrations actually cause the adaptive protocol to perform worse than MESI. Even though some of these migrations may be beneficial in correctly identifying migratory data, enough of these migrations are still erroneous creating an undesirable end result.

6. Analysis

As seen from the Results section, Migratory Adaptive Protocol does perform better than MESI for some benchmarks, but is not a clear winner as there are still cases when MESI outperforms it. One possible solution to this is to use hysteresis for every directory line. This would give us more history of the data access patterns and enable us to better predict whether the data is migratory or not. Therefore, instead of having just one migratory bit, we could have a migratory field of 2 bits. This slight increase in the history vector would enhance the performance by a lot, as we would be able monitor up to four patterns that occurred in the past.

Another approach would be to explore more adaptive protocols that employ these data access patterns in different ways. For example, schemes like Random Walk Protocol and Last-Three-Samples Protocol adjust a value called *invalidate threshold* for each block over time [2]. To determine this threshold, the protocols use write-run statistics as discussed in Section 1. Based on the invalidation threshold, a protocol chooses whether to update or invalidate a block. Because of this threshold value, these protocols may be able to perform better in benchmarks like Radix.

7. Conclusion

In this project, we implemented Migratory Adaptive Protocol on top of an invalidation protocol scheme in RSIM simulator on a directory based coherence mechanism. This protocol is a kind of Adaptive Hybrid Protocol that uses Migratory data access patterns to decrease the number of bus transactions during a program execution. We compared this adaptive protocol with MSI and MESI protocols.

One of the main results was that the benefits of adaptive protocol were not evenly observed throughout the benchmarks. For two out of five benchmarks simulated, the adaptive protocol did better than MESI. For the other two benchmarks, both adaptive and MESI protocols performed exactly the same. For the last benchmark, the

performance actually degraded for adaptive protocol. On the other hand, Migratory Adaptive Protocol always did better than MSI protocol.

Due to the variability in the performance of Migratory Adaptive Protocol, it does not overcome the initial goal of reducing the number of bus transactions for all the benchmarks. Because of some failures, it may not be suitable as a coherence mechanism scheme. However, adding another bit to the migratory data access pattern field may enhance the results considerably. Future work may be done in this area by simulating the directory with variable size migratory data access field to see which size works the best.

8. References

[1] S. Adve, et. al., RSIM – Rice Simulator for ILP Multiprocessors, *University of Illinois at Urbana-Champaign*, http://rsim.cs.uiuc.edu/rsim/dist.html, Apr. 2006.

[2] C. Anderson and A. R. Karlin, "Two Adaptive Hybrid Cache Coherency Protocols," *Proc. Second Int'l Symp. on High-Performance Computer Architecture*, pp. 303-313, Feb. 1996.

[3] A. L. Cox and R. J. Flower, "Adaptive Cache Coherency for Detecting Migratory Shared Data", *Proc. 20th Int'l Symp. on Computer Architecture*, San Diego, California, pp. 98-108, May 1993.

[4] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation," *Proc. 15th Int'l Symp. on Computer Architecture*, Honolulu, Hawaii, pp. 373-382, May 1988.

[5] R. Giorgi, *Evaluation of a Coherence Protocol for Eliminating Passive Sharing in Shared-Bus Multithreaded Multiprocessors*, PhD thesis, University of Pisa, Jan. 1999.

[6] A. Goel, A. Roychoudhury and T. Mitra, "Compactly Representing Parallel Program Executions," *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Diego, California, pp 191-202, 2003.

[7] W.-K. Hong, N.-H. Kim and S.-D. Kim, "Design and Performance Evaluation of an Adaptive Cache Coherence Protocol," *Proc. Int'l Conference on Parallel and Distributed Systems*, Tainan, Taiwan, pp. 33-40, Dec. 1998.