

Hardware Prediction for Data Coherency of Scientific Codes on DSM

JT. Acquaviva¹ and W. Jalby²

¹CEA/DAM

²PRiSM Lab.

French Atomic Energy Commission
acquaviv@bruyeres.cea.fr

Versailles University
jalby@prism.uvsq.fr

July 28, 2000

Abstract

This paper proposes a hardware mechanism for reducing coherency overhead occurring in scientific computations within DSM systems. A first phase aims at detecting, in the address space regular patterns (called streams) of coherency events (such as requests for exclusive, shared or invalidation).

Once a stream is detected at a loop level, regularity of data access can be exploited at the loop level (spatial locality) but also between loops (temporal locality). We present a hardware mechanism capable of detecting and exploiting efficiently these regular patterns.

Expectable benefits as well as hardware complexity are discussed and the limited drawbacks and potential overheads are exposed.

For a benchmarks suite of typical scientific applications results are very promising both in terms of coherency streams and the effectiveness of our optimizations.

1 Introduction

Current architectures offering the shared memory abstraction over a physically distributed memory (DSM) constitute a very attractive solution for large scalable shared memory systems. Many of them rely on a hardware based coherency scheme which is costly in terms of hardware complexity but also in terms of performance. Latency of memory operations involving complex coherency transactions may become very large. Additionally, the coherency traffic itself can become substantial, consuming a non negligible portion of the network bandwidth.

Previous studies have showed that coherency traffic exhibit some regular patterns [CBZ90]. Corresponding optimizations have been proposed to address some of these

specific patterns [Per93], [Kax98]. More general schemes have also been proposed, but they remain costly in hardware, they require on-chip modification, or large extension of directory structure (memory overhead) [MH98].

In the scientific computing world, it is well known that data access are highly structured. This property is induced by the specific nature of scientific algorithms. Success of vector architectures, and a large amount of research in latency optimization technique rely on this fact. Stream buffers in T3E, processor prefetching in Power3, or with the coming Merced, and even compiler directed prefetching of SGI aim at exploiting these regular patterns.

Our investigations show, that for scientific codes running over DSM there is a strong presence of vectors in coherency traffic. A way to optimize traffic is to take into account data streams and regularity in coherency patterns. This is illustrated by figure 1.

Starting from this observed regularity of the coherency phenomena we propose a mechanism able to capture and exploit efficiently such regular patterns not only within loops but also between parallel loops. Our objectives are twofold: first, reduce read and write latencies and second reduce data traffic (invalidation/downgrade).

Our approach will be essentially based on anticipating the memory transactions (Read, Write, Invalidation, Downgrade) relying on a simple hardware mechanism located at the L2 cache level (off chip). Our scheme will first work by exploiting spatial locality properties within a loop, to aggregate sequence of homogeneous references to contiguous cache lines (such sequences will be called streams). For each of these sequences, a compact descriptor is dynamically built up. Then, our proposed mechanism is capable of recognizing the occurrences of similar streams in the following loops, exploiting temporal locality between streams.

One of the difficulty with prediction mechanism is the

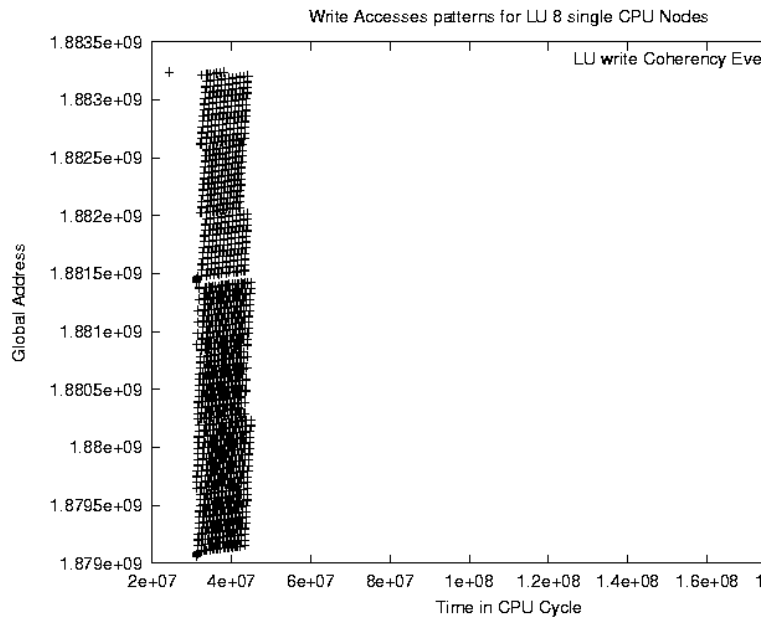
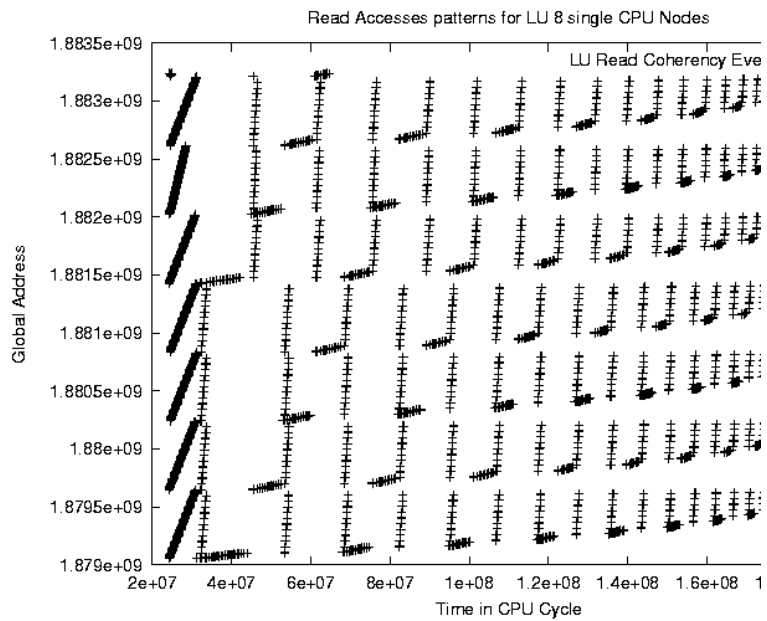


Figure 1: Data requests in shared or exclusive mode occurring for LU. Horizontal x-axis represents the time in CPU cycles, while the vertical y-axis represents the address space by cache line number. Accesses present a high degree of stream and coherent nature is also structured. Streams corresponding to the different nodes are recognizable. Such a figure is strongly related to the code structure and the parallelization scheme. Our preliminary profiling investigations clearly exhibits that few loop nests produced the main part of this coherency traffic. Notice the strong behavioral distinction between shared and exclusive accesses. No write events appear after the first quarter of the execution, all nodes have already requested all needed data in Exclusive mode and further writes are only local operations not logged on the graph.

compromise to be made between depth of anticipation (How far ahead an action can be anticipated), and the accuracy (wrong predictions and too many of them in particular) can severely degrade performance due to useless traffic. We will show that our mechanism is capable of performing well with respect to these two problems.

The remainder of the paper is organized as follow. In section 2 the framework of our study is defined. Section 3 exposes the proposed architecture for the coherency optimizer. Methodology and simulation environment are described in section 4. Results are analyzed in section 5. Several enhancements/extension are presented in section 6. Section 7 presents and reviews other anticipation mechanisms which have been proposed in the literature. Finally, a short conclusion is given in section 8.

2 Background/Framework

2.1 Target Codes

Our proposal is strongly aimed at improving performance of the so called scientific codes. A large fraction of these codes can be characterized by very regular data structures (typically multidimensional arrays which in turn are accessed in a very regular (predictable) manner. This regularity has led to the concept of “vector” (a sequence of memory locations regularly spaced) which has been very successfully exploited by vector architectures.

2.2 Programming Model

Due to the underlying architecture, we will naturally use a shared address space and we will assume that these scientific codes have been parallelized using OpenMP parallel constructs. Due to the structure of these codes, most of the parallelism will be exploited at the loop level. According to our terminology, an epoch is a fully parallel construct, in which no synchronization other than barriers¹ (located at the end and the beginning) are necessary to ensure correct execution. In particular, we will assume that there is no critical section within an epoch. This assumption is essentially made for simplifying the presentation.

The scheduling of iterations within an epoch is static, i.e., the iteration space is divided equally among the nodes, at compile time.

Interestingly enough, the regularity of access patterns which is naturally presents within epochs, is also present between epochs i.e. the same vectors are used from one epoch to the other one (cf figure 1).

¹Barrier code include an extra write to a memory mapped region in order to inform our mechanism than a barrier is occurring.

2.3 Target Architecture

Our target architecture is a Distributed Shared Memory system, whose overall architecture is given in figure 2. The address space is distributed across nodes, on a page basis, in a round robin fashion. With each page a *home node* is associated and this home node will remain unchanged during the whole code execution: no page migration mechanism are used. For sake of simplicity, we will assume that the node is a uniprocessor. Coherency is maintained via a full-map directory structure at cache line granularity. For every cache line, the corresponding directory structure and information are maintained by the corresponding home node.

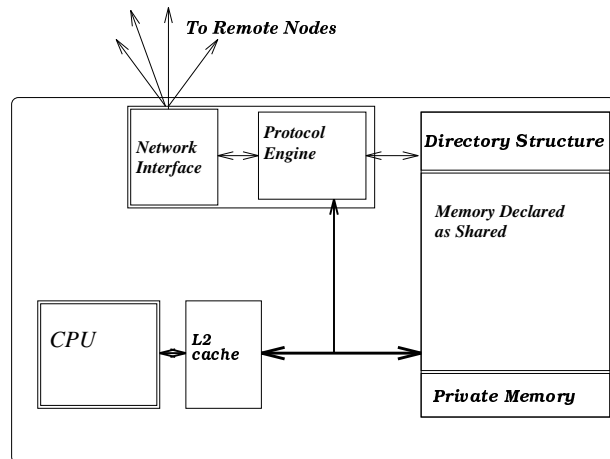


Figure 2: **Basic Architecture of a node in a DSM system.**

The consistency model supported is relaxed consistency. Data races may occur only within epochs. Memory coherency is enforced only at synchronization barrier. It is the compiler’s or programmer’s responsibility to place barriers to guarantee the correctness of the execution. A cache line can be in one of the three classic states: Invalid, Shared or Exclusive and the coherency protocol works by invalidation. The transition from the Exclusive or Shared state down to Invalid state, will be called Invalidation, and the transition from Exclusive to Shared will be called Downgrade.

Coherency protocol is executed by a protocol engine located at the Network Interface level of each node. This engine is involved to maintain coherency among nodes. Since a node may be composed of multiple CPU, inside a node, the MESI protocol is used to guarantee the coherency of each cache with the node memory. This last layer is transparent for the protocol engine.

This system is strongly related to the Prism architecture described by Ekanadham and *al.* [ELPS98]. Prism supports an interesting page migration algorithm, but interaction of page migration and coherency are not investigated in this paper.

3 Coherency Optimizer

3.1 Overview

In order to reduce coherency overhead, we propose to add a coherency optimizer within the protocol engine of each node. The regularity of data access will be characterized by means of streams. A Stream is constituted of a segment of cache lines, consecutive in term of addresses, having the same memory and coherency behavior.

More precisely, a stream will be characterized by a head (start address), a tail (end address), type of memory access (Read of Write), type of transition in the state diagram. The set of characteristics of a stream will be called a Stream Descriptor (SD).

The following subsections give an outline of the coherency optimizer design, some details might have to be changed depending upon implementation choices.

The coherency optimizer will work in two phases:

- *Streams detection/recognition*, identical coherency requests to consecutive data are detected. At this stage, a key decision has to be made: either we are observing a new stream and a Stream Descriptor has to be generated or we “recognized” a Streams already observed in the past.
- *Streams exploitation*, again two cases need to be distinguished. In the case of a new stream, anticipation of the actions will be launched usually in a careful manner (one or two blocks ahead of the current block). In the case of an already observed stream, the anticipation can be much more aggressive, i.e. anticipating the behavior of multiple consecutive blocks, even of the whole stream.

Although, our mechanism share a lot in common with standard hardware prefetchers, three major differences have to be noted. First, we monitor not only Read/Write operations but also the corresponding coherency actions. Second, we store information on data access behavior through the Stream Descriptor, this is essential for the exploitation phase. Third, we observe not only the outgoing requests issued by the processor but also the incoming requests (Invalidation/ Downgrade).

To summarize, we advocate an off-chip coherency engine, performing prediction at address level and monitoring data flowing out of the node as well as requests flowing in. We need also a dedicated buffer to store prefetched data. We should try to limit as much hardware complexity but still be able to catch the main part of coherency traffic.

3.2 Targeted Traffic

The streams detected will be classified into four major categories:

- **Read Streams**, this corresponds to a standard prefetch request.
- **Write Streams**, act like a Read stream, except that data are requested in Exclusive mode. Hence, Invalidation are anticipated and overlap is improved.
- **Invalidate Streams**, a node may receive a large sequence of invalidation for data it has loaded. On such a sequence, the coherency optimizer generates a stream of self invalidation requests, invalidation messages are also sent to the corresponding home node to decrease the degree of sharing.
- **Downgrade Streams**, similarly to the Invalidate Streams, a node may receive large sequence of remote read requests for data it has produced. These requests downgrade the coherency state from exclusive to shared, send a copy to the requesting node and to the home node. The coherency optimizer anticipates such behavior, downgrading in advance the mode from exclusive to shared, which leads to an update message to the home node. A further optimization would be brought by anticipating also the requester and sending him the data.

3.3 Hardware Organization

The coherency optimizer needs to be tightly coupled to the coherency protocol engine. It has to monitor requests produced by each node to detect the regular data patterns (vector) of identical requests. Due to page allocation which tries to balance accesses on all home nodes, these vectors will be less obvious to detect if the Descriptor is placed at the home node level.

Hence, we chose to place the optimizer at the Network Interface level, which is also the location of the coherency protocol engine.

Network Interface and coherency engine allow the optimizer to access global addresses, alleviating the address translation problem found in several DSM (like

Prism [ELPS98] or more generally S-COMA systems [SWCL95],[SN95]).

A simplified view of the overall hardware organization is given in figure 3. Four sets of basic components are distinguished:

- **Address Buffers:** these buffers are in turn subdivided into four buffers (Read Miss Buffer, Write Miss Buffer, Invalidation Buffer, Downgrade buffer) depending of the type of memory request. All four address buffers are indexed by the address of the missing cache line and hold the most recent transactions observed, i.e. potential candidates for stream detection.
- **Stream Buffers:** these buffers are used to store the data corresponding to a detected stream. These buffers are subdivided into two classes, Read Stream Buffers and Write Stream Buffers. Each Stream Buffer will be allocated to a given detected stream, according to a LRU policy. The buffers will operate according to a FIFO policy. For example, a Read Buffer will be filled by requests issued by the anticipation mechanism and emptied by requests issued by the processor. The two key parameters for these buffers are first their depth which is strongly related to the degree of anticipation and second their number which corresponds to the number of simultaneously detected streams within an epoch. In our experiments, we found out that 32 buffers were enough.
- **Stream Descriptor Table:** this table will record all of the descriptors corresponding to streams detected. This table is indexed by the head address of the stream. At each time, a new stream is detected, an entry is allocated in the SD Table. This table is split into two sections: the first section called New Section is dedicated to “new” Streams detected within the current epoch while the second section, called old section is dedicated to Streams detected in previous epochs. With each of these entries an activity flag is set depending whether the Stream is active for the current epoch. For the New Section, all of the activity flags should be set to active. In case of an overflow of the table, Streams Descriptors are discarded according to a LRU policy. ² In our experiments we observed that a 256 entries table was large enough to store all of the streams detected.
- **Address Comparators:** these comparators will be used first for checking that a miss request can be ser-

²A better policy would be to discard entries in the table depending upon the length of the stream. Short streams should be evicted first.

viced by a stream buffer and second for detecting that a stream has been already detected.

Upon a synchronization (typically a barrier), all of the address buffers and the Stream Buffers will be flushed and the content of the New Section of the Stream Descriptor Table is shifted into the Old Section to include the streams detected within the finishing epoch. All of the activity bits are reset to 0.

3.4 Stream Detection/Recognition

This phase is an essential one, since all further optimizations/ anticipation will take place on detected/recognized streams.

Let us distinguish two cases depending whether requests are issued by the local CPU or by a remote CPU.

Local request

Upon a read miss (the write miss case will be handled in a similar manner), the address will go through potentially up to three stages (corresponding to checks with the various buffers)

- **STAGE 1:** Check against the Stream Buffers.

If the data is found in one of the Stream Buffer (hit), it is directly sent to the processor and the corresponding Stream Descriptor is updated; if it is a new Stream being currently detected, the tail address is incremented by one, otherwise no operation is performed on the SD table.

If no Stream Buffers contains the requested cache line, the requested address is forwarded directly to the network and in parallel, the requested address is sent to Stage 2.

- **STAGE 2:** Check against the entries of the Old Section of the Stream Descriptor Table.

If the requested address matches one of the head address, the activity bit is set and a prefetch is launched. This case corresponds to the situation where a stream has been recognized. Otherwise, the requested address is sent to stage 3.

- **STAGE 3:** Check against the miss buffer.

First the address is decremented by one before looking for a match with an entry in the miss buffer. If a match is found, it means that two consecutive cache lines have been requested and a new stream has been detected. In such a case, an entry is created into the new section of the Stream Descriptor Table and the matched address is retrieved from the miss buffer.

If no match is found, the requested address is simply inserted into the miss buffer.

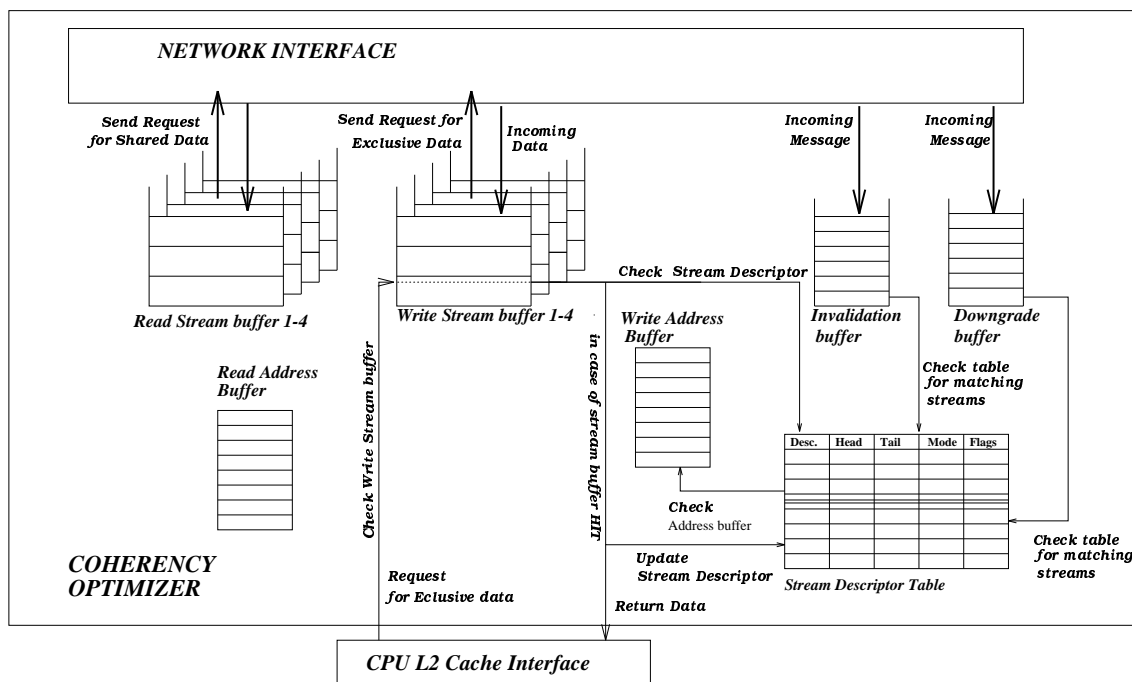


Figure 3: **Coherency Optimizer overview:**

A request for data in exclusive mode triggering a L2 cache miss reaches the Coherency Optimizer. Several cases are possible:

Write Stream Buffers are checked for the requested data. In case of hit, the data is returned back to the processor, a Write prefetch is sent and the Stream Descriptor Table is updated.

In case of Write Stream Buffer miss: a look-up is performed in the *Old section* of the Stream Descriptor Table for a previously detected stream matching with the requested address.

In case of Stream Descriptor Table hit, the corresponding activity bit is set. Depending the previous state of the Stream Entry, the Stream may be re-activated.

If the Look-up is a miss, the address is sent to the Write Address Buffer

For simplicity reasons the set of comparators is not depicted on the figure.

Remote requests

Within the request flow coming from remote processors, only Invalidation and Downgrade requests are tracked down. It means that we do not attempt to create New Invalidation Streams. The remote requests follow a similar procedure as the one described above except that stage 2 and stage 3 are inverted, the miss buffers are checked before checking the SD table.

- STAGE 1: Check against the Stream Buffers.

If the data is found in one of the Stream Buffer (hit), the corresponding entry is invalidated in the Stream Buffer and the corresponding request is forwarded to the L2 controller.

If no Stream Buffers contains the requested cache line, the requested address is forwarded in parallel to the L2 controller and to Stage 2.

- STAGE 2: Check against the miss buffer.

First the address is decremented by one before looking for a match with an entry in the miss buffer. If a match is found, it means that two consecutive cache lines have been subject to the same coherency action. Therefore, we consider that a stream has been detected and we go directly to stage 3.

If a match is not found, the requested address is simply inserted into the miss buffer.

- STAGE 3: Check against the entries of the Old Section of the Stream Descriptor Table.

If the requested address matches one of the head address, the activity bit is set and an anticipated action (Invalidate or Downgrade) is launched. This case corresponds to the situation where a stream has been recognized.

3.5 Stream exploitation

The two major cases to be distinguished are whether we are processing a New Stream or an Old Stream.

New Stream

The anticipation will be strictly limited to one block ahead of the last block requested by the processor. This corresponds to a conservative approach because the tail of the stream is not known a priori. The operation mode is relatively straightforward: upon a hit in a one of the Stream Buffer, a prefetch is launched on the consecutive address.

Old Stream

There we are relying and betting on the fact that the requested stream will have the same length as the recorded

stream. In such a case, anticipation can be more aggressive, by allowing a higher degree of anticipation.

3.6 Correctness issue

A crucial aspect in anticipating coherency protocol actions, is to ensure that misprediction will never lead to data corruption.

First of all, all buffers are flushed between epochs and all anticipated actions should be completed/cancelled before finishing an epoch.

Second, all anticipated actions will be flagged tentative and in case of two transactions, one of them being a regular one and the other a tentative one, the tentative will be systematically discarded.

Now if we look in more details at the anticipated actions, they fall into two categories.

The Read/Write anticipations are strictly equivalent to standard prefetch and should be handled exactly the same techniques.

The Invalidation/Downgrade operations are a bit trickier. First, for Invalidation, in a normal system, a cache line can be thrown away from the L2 cache due to a cache conflict for example. Therefore, our strategy will be to follow the exact same protocol steps as if the cache line had been evicted from the L2. For the Downgrade, the situation is similar.

3.7 Overhead Investigation

Cost for the critical path is fairly limited, all of the mechanisms are located off chip. The longest sequential path implies: SD-table look-up, check of stream buffers and miss buffer comparison. All these three steps can be individually more or less done in parallel depending upon the number of comparators used.

This scheme relies on exploitation of both spatial and temporal locality. For applications which don't exhibit such properties (for example fully irregular data access), the detection stage of our mechanism will not identify streams, not generating any speculative actions and hence the overhead in term of misprediction will be very limited.

The real worst case occurs for code presenting just enough regularity to trigger streams allocation, but not enough to allow stream exploitation. For this much more limited class of applications, the coherency optimizer may generate up to 1/3 of useless requests.

A way to avoid such waste of resource is to add a hang up mechanism relying on a misprediction counter. This counter is incremented for every data fetched in stream buffer but which are never accessed (i.e. flushed at the

end of an epoch). When this counter overflows, the coherency optimizer can switch off by itself or even pass in monitor only state, where prediction are made but no requests emitted.

4 Simulation Environment and Methodology

4.1 Simulation Environment

Our investigations were done with the following software environment:

Benchmarks were run with the Prism simulator provided by IBM [ELPS98]. This software is based on a port of Augmint [NMST96] for PowerPC, hence simulation model is execution driven. Prism simulator allows to simulate a cluster of SMP nodes, coherency protocol at the cache line level, and complete memory hierarchy. In particular, contention at network interface and protocol engine level are fully modeled.

This simulator has been augmented with our proposed mechanism.

Due to the execution-driven mechanism, out-of-order execution is not currently supported (in particular, coherent caches are blocking).

Although out of order execution may change coherency cost by allowing a greater degree of overlap, globally the number of coherency messages (except for false sharing/ping pong phenomena) should not be affected much by the execution order.

Two memory management schemes are possible: S-COMA or CC-NUMA. In order to limit noise from cache effects, simulations were run in S-COMA mode.

4.2 Methodology

Precise earnings with respect to execution time have been evaluated.

However, although execution time is the key metric to be measured, it depends upon many system parameters and in a complete system it tends to make the analysis of the proposed mechanism extremely dependent upon the system characteristics in which it is integrated.

Therefore, in addition to execution times, we have been analyzing in detail the ability of our proposal to detect access patterns and to capture coherency effects. Currently, one of our goal is to prove quantitatively (with respect to application characteristics) the validity of this concept.

4.3 Parameters of the Target Simulated System

Targeted architecture is a S-COMA DSM system, composed of 8 single CPU nodes. Coherency is enforced with a full-map directory at the cache line level. Cache line size is 64B. Each node includes a PowerPC 604e, with 4 MB of L2 cache, a network interface and a hardware protocol engine. Programming model is SPMD, with relaxed consistency, coherency is applied at synchronization points.

The following parameters were used for the coherency optimizer:

- The number of Read and Write stream buffers was set to 32, each of these buffers being of depth 1.
- The address buffer's depth was set to 32
- The size of the SD table was set to 256.

4.4 Benchmarks

All of the benchmarks used, are extracted coming from the SPLASH, SPLASH-2 [CSG98] suite except the CG code which is a NAS benchmarks. This last code was tested using two different preconditionners, a diagonal one (CG-DIA) and a polynomial one (CG-POLY).

Benchmarks can be sorted into four categories, depending upon two parameters: memory access regularity (spatial locality), and iterative nature (temporal locality).

The details of the data sets used and benchmarks type are given in table 1

5 Experimental Results

5.1 Metrics Used

In the tables, an event refers to complete memory/coherency transactions: a R/W event refers to a Read or a Write request having generated a coherency action (local Reads or hits in the L2 are not counted), an Invalidation (resp. Downgrade) event corresponds to all of the actions related to an Invalidation (resp. Downgrade) request. For example, a cache line present in 5 different caches when another processor attempts to get Exclusive ownership will generate 5 Invalidation events.

The term **Optimized Event** refers to an event which was predicted correctly by our mechanism.

The term **Recognized Event** corresponds to Optimized Events which were captured by stream recognition: i.e. the same stream occurred at least twice across different epochs. This metric is interesting because it describes the temporal locality of codes.

Benchmarks	Problem size.	Type
LU	768x768 matrix, block of size 16.	regular and highly iterative
CG-DIA	50x50 matrix.	regular and highly iterative
CG-Poly	50x50 matrix.	regular and highly iterative
FFT	64 Kpoints.	regular and weakly iterative
OCEAN-cont	Grid of size 130x130	regular and weakly iterative
OCEAN-non-cont	Grid of size 130x130	regular and weakly iterative
Radix	524288 elements, radix 1024	regular and not iterative
MP3D	20000 particles for 5 time steps	irregular and moderately iterative
Water-spatial	2197 molecules for 5 steps.	irregular and moderately iterative

Table 1: **Benchmarks set and associated problem size.**

All of the statistics were gathered for the whole execution of the code.

The Invalidation (resp. Downgrade) prediction were made using the most aggressive scheme: as soon as two requests were hitting consecutive lines at the beginning of a Stream, the whole Stream was invalidated (resp. Downgraded)

Our objectives were twofold: first, evaluate how well our mechanism was able to detect and recognize streams (coverage ratio), and second to evaluate the accuracy of the mechanism (misprediction ratio).

This last point is fundamental because, incorrectly predicted actions consume bandwidth directly related to the useless action. Furthermore, it can generate additional traffic to recover from the misprediction: invalidating a cache line which was not required will cost the extra invalidation and on the top of that if unfortunately, the cache line is subsequently needed by the processor, it will have to be reloaded!

5.2 Comments on the results

First global improvement on execution time brought by our scheme is exposed in table 2. Table 3 presents the results obtained for the R/W optimization, a more detailed analysis of the stream recognition across barriers being given in Table 4.

Tables 5 (resp. 6) presents optimization performed for Invalidation (resp. Downgrade).

- LU: within an epoch, the coverage factor is pretty good, however the mispredictions are relatively high due to the shortening of vectors (and therefore streams) during the code execution. Even if R/W streams are well recognized, the variation in length leads to mispredicted stream tail, i.e. additional traffic. Hence, LU doesn't benefit from information col-

lected during previous epochs. Nevertheless, SD table is useful to handle downgrade requests, which leads to traffic reduction and performance gains.

- CG-DIA and CG-POLY; these codes which share common characteristics are analyzed together. In both cases, due to the very regular underlying data structures and the highly iterative nature of the code, our mechanism performs extremely well, both within an epoch and between epochs.

After an initial learning phase, where streams are detected (i.e. the SD table is built), the recognition ratio is excellent. Despite this, CG-Poly exhibits a non negligible percentage of misprediction for Downgrade traffic and CG-Dia for Invalidations. Misprediction is coming from stream overlap. For instance, each node performs an initial read access on the same large stream, then each node only writes a slice of this stream. This pattern yields to a indiscriminate invalidation of the whole stream on all nodes, even on the portion that each processor needs to write. It should be noted, that this will not lead to data reload, since the local slice has to be written anyway (old values are scratched) from shared state to exclusive state.

- FFT: within an epoch the coverage factor is very good. Between epochs, the results are much more disappointing, this is due to the very limited number of epochs (8) throughout the whole code. Additional experiment shows that caches lines which are accessed more than 4 times concentrate less than 0.4 % of the total number of R/W events. Clearly, with to few epochs the scheme suffers from its learning curve.
- Oceans Contiguous/Non Contiguous: our mechanism performs well on both codes, although in both

Application	LU	CG-Dia	CG-Poly
Basic	160,735,466	87,088,792	17,424,393
Optimized	153,250,797	77,586,113	15,803,924
Ratio	95.3	89.1	91.8
Application	FFT	Ocean-Cont.	Ocean-non-cont.
Basic	18,061,764	58,683,199	31,061,465
Optimized	16,748,673	55,438,018	29,905,357
Ratio	92.7	94.5	96.3
Application	Radix	MP3D	Water-Spatial
Basic	38,623,341	73,971,006	407,468,331
Optimized	36,096,679	73,133,654	431,560,466
Ratio	93.0	98.1	105.9

Table 2: Execution Time with the coherency optimizer

Basic, corresponds to the execution time in CPU cycle on the plain system, without any coherency optimizer. **Optimized**, corresponds to the execution time in CPU cycle on the augmented system, which includes the full implementation of our proposal. **Ratio**, Execution time on the Optimized system over execution time on the Basic system.

cases, our mechanism is suffering from being limited to stride 1 stream detection. Not surprisingly, contiguous version performs better, taking advantage of longer streams, up to 300 cache lines long for ocean-contiguous as detected streams never exceeds 8 cache lines for the non-contiguous version. Temporal locality is also more present in ocean-contiguous, with a coverage factor across epoch of 34.3 % with 934 epochs as ocean-non-contiguous is limited to 13.3 % with 902 epochs.

- Radix: surprisingly, our mechanism performs well on radix, due to the highly regular data access. In this application 99.9 % of the data set is touched at most two times. Access pattern is completely dominated by a producer-consumer behavior. Hence, for R/W events, despite the fact that they are highly regular (strong presence of streams) and optimized, the recognition stages is never involved. Performance improvement is dominated by local optimization, and SD table is used to predict downgrade traffic. Notice that this case cannot be optimized by mechanisms relying on a per cache line predictor.
- MP3D: although the code is accessing sparse structures, our mechanism is performing quite well. The iterative nature of the code, helps substantially for cross epoch optimization. The lack of regularity leads to a high misprediction ratio, nearly one predicted request out of three is useless. Despite this, we observe a performance improvement, due to the fact that bandwidth limitation was not the key bottleneck. Like for Water-Spatial, some refinements like

a less sensitive stream allocation policy, can be used to avoid to be misled by all these too short streams. Further more all our experiments were conducted with a 64 B long cache line which is quite small.

- Water-Spatial: results are disastrous. First, the irregularity of the data structure cannot be detected by our mechanism. Worse, many very short streams (2 cache lines long) mislead the stream detector, resulting in a very high misprediction ratio. In this application spatial and temporal locality are not coupled, 5 % of the shared address space summarizes 63 % of R/W coherency events, and this small fraction is not accessed with enough regularity to be caught by the detection stage.

If we analyze and compare the two types of optimizations (within an epoch and between epochs), the first one performs much better in terms of coverage factor. The essential reason of such a behavior is that within an epoch, anticipation based on spatial locality and (recent references) performs better than cross epoch optimization which tends to correlate phenomena far apart in time.

As a summary, our proposed mechanism will perform at best for codes which are both highly structured (vectorizable) and highly iterative: for such codes, performance improvement is over 50%, reaching over 70% in the most favorable cases. If only one of the two conditions is met, our scheme still provides a substantial improvement (between 25 and 40 %). If none of the conditions are met, clearly our scheme brings limited improvement (10 to 20 %). In all cases, accuracy of the predictions (and the corresponding overhead) are very good except for highly un-

Benchmarks	Nb. R/W Events	Optimized R/W Events	Coverage Factor	Misprediction Ratio
LU	276433	146062	52.8 %	18.9 %
CG-DIA	187906	171355	91.2 %	2 %
CG-Poly	56087	41681	74.3 %	3.3 %
FFT	129875	114945	88.5 %	3.9 %
OCEAN-cont	278727	87573	31.4 %	1.8 %
OCEAN-non-cont	53493	13978	26.1 %	3.6 %
Radix	127210	56064	44.1 %	0.9 %
MP3D	422613	108503	25.7 %	3.5 %
Water-spatial	211998	30234	14.3 %	28.7 %

Table 3: **Summary of R/W Optimizations**

Nb. R/W Events: total number of read/write coherency requests transiting in the system during the execution time.

Optimized R/W Events: number of R/W events that are correctly coalesced in streams and correctly anticipated by the optimizer.

Coverage Factor: ratio of the two previous metrics.

Misprediction Ratio: useless traffic (in number of messages) generated and incorrectly predicted by the optimizer over **Nb. R/W events**.

Benchmarks	Optimized R/W events	Recognized R/W Events	Recognition Factor
LU	146062	54600	37.4 %
CG-DIA	171355	165597	96.6 %
CG-Poly	41681	18975	45.2 %
FFT	114945	28688	24.9 %
OCEAN-cont	87573	30093	34.3 %
OCEAN-non-cont	13978	1865	13.3 %
Radix	56064	0	0 %
MP3D	108503	25458	23.4 %
Water-spatial	30234	4360	14.4 %

Table 4: **Streams Recognition across barriers:**

Optimized R/W events: number of R/W events correctly predicted by the optimizer

Recognized R/W Events: number of optimized R/W events that appear at least in two epochs and are captured and optimized by our mechanism, successfully exploiting information gathered during previous epochs.

Recognition Factor: number of **Recognized R/W Events** over number of **Optimized R/W Events**.

Benchmarks	Invalidation	Streams I.	Coverage Factor	Misprediction Ratio
LU	0	0	na.	na.
CG-DIA	89747	80135	89.3 %	1 %
CG-Poly	19828	14208	71.6 %	7.9 %
FFT	25078	2874	11.4 %	0.5 %
OCEAN-cont	115439	15556	13.4 %	1.1 %
OCEAN-non-cont	10045	1443	14.3 %	1.2 %
Radix	19	0	0	na.
MP3D	208835	5013	2.4 %	1.3 %
Water-spatial	128804	598	4.6 %	0 %

Table 5: **Summary of Recognition phase for Invalidation Events:**

Invalidation: total number of invalidation events emitted in the system.

Streams I.: number of useful invalidation events generated by our mechanism.

Coverage factor: number of useful invalidation events performed over total number of invalidation.

Misprediction Ratio: number of wrong invalidation generated by the coherency optimizer over total number of invalidation events.

Benchmarks	Downgrade	Streams D.	Coverage Factor	Misprediction Ratio
LU	41274	10135	24.5 %	2.4 %
CG-DIA	68173	66123	96.9 %	19.8 %
CG-Poly	17942	17512	97.6 %	0 %
FFT	21606	3450	15.9 %	11.9 %
OCEAN-cont	92561	26443	28.5 %	4.8 %
OCEAN-non-cont	15113	1100	7.3 %	0 %
Radix	57432	25442	44.3 %	6.1 %
MP3D	190117	3544	1.8 %	0.5 %
Water-spatial	28620	80	2.7 %	0 %

Table 6: **Summary of Recognition phase for Downgrade Events:**

Downgrade: total number of downgrade events emitted in the system.

Streams D.: number of useful downgrade events generated by our mechanism.

Coverage factor: number of useful downgrade events performed over total number of downgrades.

Misprediction Ratio: number of wrong downgrades generated by the coherency optimizer over the total number of downgrade events.

structured code such as Water Spatial. This last case could be easily corrected by triggering stream detection on three contiguous cache lines instead of two.

This technique would increase the learning phase, a more ambitious mechanism will be able to switch off by itself when a misprediction counter overflow (number of data fetched in stream buffer but never accessed) or even to pass in monitor only state, where prediction are made but no requests emitted.

6 Future Works

6.1 Mechanism Improvement

- **Stride Detection:** in the proposed mechanism, our stream detector is triggered only for two requests hitting consecutive cache lines. This scheme works perfectly for detecting stride 1 and even up to stride 8 access, since the cache line is 8 words long. However, for larger strides, our stream detector is not triggered. This lack of detection could be corrected by integrating stride detector similar as the ones proposed for standard prefetching [CB95] [Hag92]. Such an addition would clearly enhance the coverage factor capabilities of our mechanism.
- **Degree of Anticipation** The anticipation scheme used is very simple: one block ahead within an epoch and the whole stream between epochs. First the degree of anticipation could be made adaptive within epoch and second for the anticipation between epochs, a fixed degree of anticipation would reduce the potential over-invalidation (such as the one observed in LU) and for a moderate cost in terms of coverage factor.
- **Request combining/Stream merging,** request combining is a classical optimization in coherency protocol [GGK⁺83], [LL97]. This allows to save both bandwidth and latency. A simple statement such as $A[i]=A[i]+1$, first triggers a remote read miss and then a remote write miss. Two distinct coherency requests have to be issued the first one for the read and the second one for the write. Combining both requests into a single Exclusive request would halve the number of coherency actions. This idea can be easily integrated in our scheme: on a write stream buffer checking, read streams buffer are also checked first. If a hit occurs in both buffers, the two streams can be combined into a single request stream.

6.2 Mechanism Extensions

- **SMP nodes.**

Our scheme has been presented in the context of uniprocessor nodes. It could be extended to deal with SMP nodes. Some extension are straightforward: instead of interfacing with the L2 controller, our coherency engine would interface with the bus coherency mechanism (cf PRISM). However, more subtle questions have to be investigated: are stream buffers allocated on a processor basis or are the requests of all processors lumped together to constitute streams. More experimentation is needed to get a clear view on both options.

- **Combining Hardware and Software.** Our scheme is essentially hardware based, and it could benefit from hints (such as Stream Descriptors) provided by a static analysis done by the compiler. Careful analysis have to be performed to determine how hardware and software should cooperate to exploit at best all of the opportunities by spatial and temporal locality.

7 Related works

Anticipation is a very old technique which has been successfully exploited in various hardware mechanisms: branch prediction, prefetching, coherency anticipation...

Following Kaxiras [Kax98] suggestion, anticipation mechanisms can be classified according first how anticipation actions are inserted (either statically by the compiler or dynamically at run time by a specialized hardware) and second on the type of information they are correlated to (instructions or address).

Address oriented methods for prediction are expensive [MH98] and often outperformed by instructions driven schemes [Kax98]. However instruction oriented method need to be tightly integrated with the CPU which could be rather difficult. Furthermore, for scientific code, the high degree of regularity clearly advocated for the choice of address based method (as we propose).

To support this point, many prefetch mechanism have been proposed using address based methods [ABH97], [PK94], [Hag92]. Following the instruction driven path, Baer and Chen [CB95] have proposed a very elegant mechanism using the program counter. In general, all of these schemes do not address coherency anticipation. Furthermore, they only exploit spatial locality within a loop, missing opportunities between loops.

With respect to coherency anticipations, very powerful schemes have been proposed: Cosmos [MH98], is a

complete coherency prediction mechanism. This scheme is implemented in hardware and is address oriented. Every cache line is tracked, and a history of the coherency events is stored in an enhanced directory structure. Then the future behavior of each cache line is anticipated with an Yeh and Patt's adaptive predictor [YP92]. This scheme is very attractive, but the exhaustiveness leads to a large memory overhead to store lines history. Furthermore, it is targeting only temporal locality and not at all spatial locality which is detrimental for scientific codes. This scheme was further improved by Falsafi and *al.* [LF99].

Various coherency prediction mechanisms were investigated by Kaxiras in his thesis [Kax98], [KG99]. His research was done in a SCI environment and he successively evaluates static and dynamic method. Conclusion were in favor of a hardware prediction engine, instruction oriented and integrated on-chip. Despite its promising results, this solution is mainly focused on coherency prediction, it doesn't address spatial locality exhibited by data patterns, and it's correlated to SCI and network topology issues.

Skeppstedt [Ske97] proposes an interesting coherency aware prefetcher, combining both hardware and software. The hardware prefetch engine sits on the L2 cache and is able to launch requests for data in different coherency states. Stream are defined at compile time, and on stream allocation the compiler forwards some information to the prefetcher, basically stream head, tail and coherency mode. Hence, the programmable prefetch engine can perform shared/exclusive loads merging, avoid complex hardware stride detection and limit over-prefetch.

Even if a hardware mechanism is involved, the main task, i.e. stream detection and coherency analysis, is realized statically during the compilation phase. Modifications are needed both in the compiler to perform the analysis and in L2 cache to support compiler inserted instructions. As usual, with all compiler based techniques, the hard limitations are the cost and the complexity of analyzing data patterns. In particular, memory aliasing can make any compiler analysis almost infeasible, while dynamic methods at run time can still operate very profitably in such cases. Further more the analysis proposed by Skeppstedt is essentially local to a loop and more important Invalidation and Downgraded streams are not optimized.

8 Conclusion

In this paper, a new coherency anticipation mechanism for DSM architectures has been described. This mechanism, purely hardware based, could be easily integrated

at the L2 level. Its main strengths are its capabilities to exploit locality properties both within a loop and between loops. These capabilities are exploited to enhance the potential degree of anticipation, without degrading performance with mispredictions.

First evaluations of the proposed mechanisms on Splash 2 codes and on a NAS one have clearly shown the interest and the potential performance gains which can be very high for very well structured and highly iterative codes. Such codes represent an important fraction of scientific codes.

Several potential improvements of the basic mechanism have been proposed. Furthermore, our coherency optimizer could be enhanced for dealing with SMP nodes and also for exploiting infos provided by a static analysis performed by the compiler.

Acknowledgements

Ekanadham Kattamuri from IBM for his helpful advises and comments on the IBM PRISM simulator.

References

- [ABH97] Ed Anderson, Jeff Brooks, and Tom Hewitt. The Benchmarkers' Guide to single-Processor Optimization for CRAY T3E systems. Technical report, Benchmarking group, CRAY research Inc., June 1997.
- [CB95] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, 44(5):609–623, May 1995.
- [CBZ90] John Carter, John Bennett, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the Conference on the Principles and Practices of Parallel Programming*, 1990.
- [CSG98] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture a Hardware/ Software approach*. Morgan Kaufman, 1998.
- [ELPS98] Kattamuri Ekanadham, Beng-Hong Lim, Pratap Pattnaik, and Mark Snir. PRISM: An integrated Architecture for Scalable Shared Memory. In *Proceedings of the 4th International Conference on High Performance Computing Architecture*, February 1998.

- [GGK⁺83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. the NYU Ultracomputer Designing a MIMD Shared-Memory Parallel Computer. *IEEE Transactions on Computers*, C-32(2), February 1983.
- [Hag92] Erik Hagersten. *Toward scalable Cache Only Memory Architectures*. PhD thesis, Swedish Institute of Computer Science, October 1992. Also available as SICS Dissertation series 08.
- [Kax98] Stefanos Kaxiras. *Identification and Optimization of Sharing Patterns for Scalable Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1998.
- [KG99] Stefanos Kaxiras and James R. Goodman. Improving ccnuma performance using instruction-based prediction. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 161–170, February 1999.
- [LF99] An-Chow Lai and Babak Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.
- [LL97] James Laudon and Daniel Lenoski. The SGI Origin: A CC-NUMA highly scalable server. In ACM press, editor, *Proceedings of the 24th International Symposium on Computer Architecture*, volume 25, June 1997.
- [MH98] Shubhendu S. Mukherjee and Mark D. Hill. Using Prediction to Accelerate Coherence Protocol. In *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.
- [NMST96] A-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation Toolkit for Intelx86 Architecture. In *Proceedings of the 1996 International Conference on Computer Design*, October 1996.
- [Per93] Per Stenström, Mats Brorsson and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.
- [PK94] Subbarao Palacharla and R.E. Kessler. Evaluating stream buffers as secondary cache replacement. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 24–33, May 1994.
- [Ske97] Jonas Skeppstedt. *Compiler-based approaches to reduce memory access penalties in cache coherent multiprocessors*. PhD thesis, Chalmers University of Technology, April 1997.
- [SN95] Ashley Saulsbury and Andreas Nowatzky. Simple COMA on S3MP. In *Proceedings of the 1995 International Symposium on Computer Architecture Shared Memory Workshop*, June 1995.
- [SWCL95] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An Argument for Simple COMA. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, January 1995.
- [YP92] T-Y Yeh and Yale Patt. Alternative Implementation of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.