Implementing Snoop-coherence Protocol for Future SMP Architectures

Wissam Hlayhel[1], Jacques Collet[2], and Laurent Fesquet[2]

[1] IRIT-Université de Paul Sabatier, 118 route de Narbonne
31062 Toulouse-France. email: hlayhel@irit.fr
[2] LAAS-CNRS, 7 avenue du colonel Roche,31077 Toulouse-France.
email: {collet,fesquet}@laas.fr

Abstract. Maintaining the coherence is becoming one of the most serious problems faced when designing today's machines. Initially, this problem was relatively simple when the interconnection network of Symmetric MultiProcessors (SMP) was an atomic bus, which simplified the implementation of invalidation coherence protocols. However, due to the increasing bandwidth demand, atomic busses have been progressively replaced by split busses that uncoupled the request and response phases of a transaction. Split busses enable initiating new requests before receiving the response to those already in progress but make more complicated the preservation of the coherence. Indeed, a new request induces a conflict when it concerns a block address involved by another current request and when one of the requests is a WRITE miss. Several solutions exist to solve this problem. That one used in the SGI machine is based on a shared data bus which traces the completion of transactions. Unfortunately, it becomes impracticable in the recent machines which replace data busses by more efficient networks (again for bandwidth constrains), ultimately by a crossbar. This work describes and quantitatively evaluates two possible solutions to the coherence problem for the new architectures where all the data responses cannot be traced by each processor.

1 Introduction

The presence of caches in multiprocessors sets the problem of the data coherency. Each processor may load and subsequently modify a data block in its cache that makes incoherent the copies in the different caches. The coherence is preserved either by broadcasting each modification to immediately update all the other caches (write update protocol) or by invalidating the copies in the other caches (write invalidate protocol). This second solution is mostly used [1]. The way copies are invalidated depends on the network architecture. In a shared-bus multiprocessor, each memory transaction is snooped by all the processors. This simplifies the implementation of coherence protocols by global invalidation. The MESI (Illinois) protocol [2,3] is often used in this kind of architecture. In the first implementations, the bus and the memory were completely locked during a transaction from the address transmission phase to the reception of the data.

The bus was atomic. This operation mode simplified the preservation of the data coherence. Unfortunately, the atomic bus progressively became unable to satisfy the bandwidth demand because of the slow evolution of the bus-memory subsystem with respect to increase of the processor power. It was replaced by a 64-bit multiplexed and split bus in the XDBus from Sun [4], by two separate busses (one for the addresses, one for the data which was enlarged up to 256 bits) in the SGI Challenge [5] or in the AlphaServer 8x00 [6]. In the recent designs, data are transmitted through a crossbar network and addresses through a shared bus for preserving the cache coherence (ESCALA-Bull [7], STARFIRE-Sun [8]).

Uncoupling the address and data phases of transactions makes more complex the implementation of the MESI invalidation protocol as conflicting situations may occur when several progressing requests affect the same memory block. Consider for instance the insertion of a WRITE miss on the bus before completion of a preceding WRITE miss on the same block. A conflict occurs because the latest request must not invalidate the first one before completion of its WRITE operation and in the same time it cannot ignore the first request which is anterior in the order of insertion on the bus. A coherence conflict occurs when two (or more) current requests affect the same block and when at least one request is a WRITE miss.

Existing architectures surely solve this problem, but to our knowledge, the protocol has been only disclosed for the SGI machine. Based on the implementation of a shared data bus (see the next section), it does not work in the new data-crossbar SMPs. In the following we study the coherence conflicts and introduce two solutions free from restrictive data-network hypothesis. The work is organized as follows: We review in section 2 the SGI machine solution . Our two new proposals are described in section 3.1 and 3.2 . They are evaluated in section 4. We especially study the influence of the memory latency on the conflict number. The results are summarized in the conclusion.

2 The solution of the SGI Challenge machine

The SGI machine [5] is a Symmetric Multiprocessor (SMP) using a split transaction bus. Coherence conflicts are avoided by delaying the emission of the requests which can generate a conflict. For this purpose, each cache controller includes an associative table which lists and traces the current requests of all the caches. Sharing the address bus enables to detect the initiation of any new request that is added to the table. Sharing the data bus enables snooping the response and removing the request from the table. Thus, the cache controller delays the initiation of a new request when it is conflicting with one already in its table.

We stress that this solution is appropriate when the size of tables is small (8 entries are sufficient in the SGI machine where a maximum of 8 requests coming from all caches may be in progress) because all entries of tables must be searched at snooping cycle. The snooping cycle ought to be extended in nowadays non-blocking caches[9] (which accept up to 8 simultaneous misses) to allow searching in larger tables (as 256=8x32 entries are required for 32 processors).

3 New solutions

The solution of the SGI machine is simple, cheap but inappropriate when the data network is not a shared-bus. In this case, a processor cannot snoop the data responses (i.e., the request completion) which are intented to the other processors. Therefore, it cannot foresee a conflict when emitting a new request. However, as the processors permanently snoop the address bus, they can detect that any new request is conflicting with a local one already in progress. Two actions are possible: The processor which detects the conflict may ask the other processor to postpone the request (so-called retry solution) or it may register the request and forward the block as soon as possible (notification solution). Nowadays SMPs probably use one of these two logical solutions to preserve the coherence. To our knowledge, no quantitative results have been so far published. We detail the operation of the two techniques now.

3.1 Retry technique

In this technique, a controller detecting a new conflicting request on the address bus asks the emitter to postpone the request. The emitter will retry later with the hope that the conflict has disappeared. Sometimes, several retries are necessary. When probating this technique with our simulator (see section 4) we discovered that conflicting requests sometimes face a service equity issue, which may induce an execution deadlock. This situation occurs in the following conditions: With the "Round Robbin" arbitration mechanism implemented in the classical busses, a controller cannot resubmit a request before the scan of the bus by all the other controllers. Suppose initially that the request 'i' is conflicting with the progressing request 'a'. If no new conflicting request on the same bloc is meanwhile generated, an attempt to resubmit 'i' will be successfully done after the completion of 'a'. But contrarily, if a new request 'j' on the same bloc attempts the bus before 'i', it will be submitted successfully. In consequence, next attempt for request 'i' will fail, but because of request 'j' this time. If the race between the request 'i' and a new request 'j' iterate permanently, request 'i' will be never submitted successfully. We observed this phenomena during the execution of a LOCK by several processors. The processor which executes the UNLOCK never succeeds to release the LOCK as the other processors permanently attempt to get the LOCK.

This problem can be avoided by imposing a constraint on the arbitration mechanism : a controller which must resubmit a request keeps the bus ownership regarding the emission of miss request until the conflict disappears. During that period, only invalidation or block flush requests can be sent by the other controllers.

The retry technique simply solves the conflicts and requires a minimal hardware extension which reduces to adding one supplementary line to the bus for dispatching the retry order. However, it wastes a fraction of the bus bandwidth as a failed request gets the ownership of bus and thus prevents submitting other requests, even if they are independents.

3.2 Notification technique

A second solution enables initiating new conflicting requests (not to waste the bus bandwidth) provided that a coherent response is warranted to each one. A response is said coherent if the block READ returns the latest copy of this block (latest in the appearance order of the WRITE requests on the bus). Thus, it suffices to register the apparition order of the requests on the bus and to propagate the responses in the same order. For instance, in the case of several WRITE misses on the same block, the first controller registers the request of the second controller, the second controller that from the third one... and so on. When the first controller gets the block, it modifies and forwards it to the second controller. As this block is only kept during a WRITE period (as it will be written by a second controller), this operation can be executed on the fly without having to write the block in the cache (this operation is carried out in a buffer). Then, the second controller similarly modifies and propagates the block to the third controller... The requests notification mechanism is executed in the cache controller tables. Each local request is stored in the table with two additional fields to specify whether the request must be forwarded and to which processor. These tables only store the local requests contrarily to the solution of the SGI architecture which keeps track of all the processor requests.

This technique not only enables optimizing the bus bandwidth, it also reduces the service latency of a conflicting request. When calculating the request latency of the second processor in our previous example, an evident reduction of the latency is observed (see the definitions below).
Access latency in the retry technique = latency to get the response for request 1 + load and write latency of the block in cache 1 + resubmission latency for request 2 + latency to read and transfer the block form cache1 to cache 2.
Access latency in the notification technique = latency to get the response for request 1 + block update time in the buffer + latency to transfer the block form cache1 to cache 2.

The notification mechanism also enables to propagate the responses of non-conflicting requests. In the case of consecutive READ miss on the same bloc, the block is fetched from the memory for the first READ, then forwarded from one controller to another one for the other requests. This optimization is not studied in the following.

4 Experiment

The two techniques were validated and evaluated with an instruction-driven simulator that we developed to study the network-memory subsystem in the next generation of processors. The development cycle was complicated and required numerous efforts to construct a simulator both accurate, fast and able to execute the SPLASH2-suite programs [10].

4.1 Validation

The occurrence of unexpected states represents the major difficulty for preserving the coherence. These states are hardly predictable considering the multiple possible executions of a parallel program and the complexity to analyze the execution. Thus, we resorted to the logical definition of an execution which preserves the coherence of a shared-bus architecture: Each block READ must return the latest copy of this block (latest in the appearance order of the WRITE requests on the bus). As a result, we systematically stored the latest copy of each block and the processor cycle number of the corresponding WRITE operation. For any READ, we tested whether the returned and the stored blocks were identical. If they were not, the number of the WRITE cycle was accessible to start a step by step code debugging. A lot of time was saved by this method when testing and implementing the two coherence solutions.

4.2 Evaluation

The simulated architecture is a symmetric multiprocessor (SMP). The MESI algorithm, extended by the Retry or Notification technique implementation, preserves the coherence of caches. The architecture is composed of a set of processor-cache units (PCU) connected by a network to a shared memory. The number and the latency of PCUs, the data network model and the memory are fully configurable. Contrarily, the address transactions are always transmitted through a shared bus. We consider 8-way-issue superscalar processors with a speculative and out-of-order execution. The functional units and the issue/retire buffers were configured following Ref. [11]. Each processor is connected to two levels of lock-up-free caches. The 32-Kbyte first-level cache (L1) is direct-mapped with write-through policy and a one-cycle access time. The second-level cache (L2) is 4-way associative with write-back policy. Its size is 512Kbyte. The size of the cache block is 64 bytes for the L1 and L2. We describe in the following a standard and an aggressive models for the network and the memory.
Standard Model: We consider a classical split bus, i.e., two separate busses for transmitting address and data. The latency yields 3 bus cycles both for address and data transactions. This corresponds to 9 processor cycles (busses and processors operating at 100 MHz and 300 MHz respectively). The 8-bank interleaved memory enables 8 simultaneous accesses with an access time of 20 processor cycles ($\sim$70ns). The request service latency (counted from the cycle when the address request leaves the cache through the cycle when data come back) can be much larger than the memory plus the network access time due to contentions.
Aggressive model: As for the *standard* model, addresses are transmitted through a shared bus to maintain the coherence of caches. The bus latency as previously yields 9 processor cycles. The "novelty" in the aggressive model is that we consider a fully-connected data architecture. We assume that the rest of the memory-network is perfect, that is to say, that there is no contention effects. The data service latency is constant. It yields 100 processor cycles when data come from the memory, and 30 processor cycles when they come from another

cache (direct cache to cache transfer). We had to consider longer access times in this model as more hardware layers must be bypassed in strongly connected architectures (crossbar, complex memory controller...).

The simulator executed the following benchmarks: BARRIER: This simple program was written to estimat the topmost possible gain. Each processor executes a loop of synchronization barrier. LU, OCEAN: These applications from the SPLASH2 suite [10] have been parameterized with a reduced number of input data to increase the number of conflicts. Executing LU requires 6.7 millions of instructions including 17 synchronization BARRIERS. Executing OCEAN necessitates 27 millions of instructions, 491 BARRIERS and 114 LOCKS.

Benchmark	Execution time		Bus waste		Average latency	
BARRIER	25 %	27 %	28.8 %	33.4 %	23 %	23 %
LU	0.4 %	1.5 %	0.9 %	3.1 %	22 %	52 %
OCEAN	5 %	13 %	8 %	20.6 %	13 %	25 %
	16 Proc.	32 Proc.	16 Proc.	32 Proc.	16 Proc.	32 Proc.

Table 1 : Simulation results for the *Standard* Model

Benchmark	Execution time		Bus waste		Average latency	
BARRIER	31 %	42 %	43.4 %	51.3 %	27 %	32 %
LU	0.4 %	3.3 %	2 %	5.4 %	26 %	29 %
OCEAN	13 %	30 %	19.4 %	42.5 %	21 %	36 %
	16 Proc.	32 Proc.	16 Proc.	32 Proc.	16 Proc.	32 Proc.

Table 2 : Simulation results for the *Aggressive* Model

Tables 1 and 2 above display the simulation results for the two models considering 16 or 32 processors. Column 2 (Execution time) shows the execution time gain, namely $(N_R - N_N)/N_R$ where N_R and N_N are the numbers of cycles necessary for executing the application with the Retry or the Notification technique respectively. Column 3 displays the waste of bus bandwidth in the retry technique defined by (Number of conflicts x 9 cycles)/ N_R, where 9 cycles is the address transaction latency. Column 4 shows the relative reduction of the request service latency, namely $(T_R - T_N)/T_R$ where T_R and T_N are the average request latencies for the Retry and the Notification technique respectively.

It turns out that the performance gain varies from 0.4% to 27% in the standard model and from 0.4% to 42% in the aggressive model. The gain (column 2) depends on the conflict frequency during the execution which can be represented by the bus waste (column 3). The BARRIER application shows the topmost enhancement with the second technique. The OCEAN and LU applications exhibit a smaller (though not negligible) enhancement. When comparing LU and OCEAN, it turns out that the gain for LU is clearly much smaller than that for OCEAN (for instance: 1.5% for LU and 13% for OCEAN, lines 3 and 4 in table 1 for 32 processors). This is because with the same network configuration

and number of processors, the frequency of conflicts depends on the communication behavior of the benchmark (OCEAN exhibits more synchronization and communication phases than LU).

Now if we assume different network configurations for the same Benchmark, the frequency of conflicts mostly depends on three parameters. It linearly increases versus the number of processors (see for instance the Bus waste columns for 16 and 32 processors in the tables), the Memory service latency and the address bus bandwidth. The Bus waste is smaller in the standard than in the aggressive model (see tables 1 and 2) as the request service latency is longer in the aggressive case.

It must be stressed that the results displayed in the tables have been generated by "extreme" benchmarks which exhibit numerous conflicts. The performance enhancement is not so high for usual applications as the conflicts only occur in the "hot" phases of the execution characterized by the synchronization and the communications of data. In this case, improvement of the notification technique is applicable to 20-50% of the applications execution time.

4.3 Influence of the latency on the number of conflicts

Evaluating the influence of the memory latency versus the conflict frequency is especially interesting as the memory latency has continuously increased in the last years versus the processor cycle. This paragraph describes this evaluation in the framework of the aggressive network model to consider service latencies independently of the contention effect. Figures (a) and (b) show the increase of the conflict frequency versus the latency for 16 and 32 processors. Dots in the two figures were calculated using the previously defined "Bus waste" formula. The latency notation (x-y) for the X-axis means that the block transfer and the memory READ latency last x and y processor cycles respectively. We used the retry technique to count the number of conflicts. A request which needs n retries is counted as n conflicts.

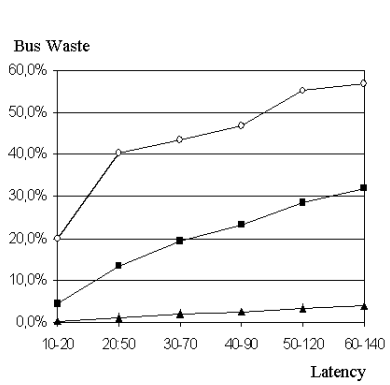

Figure (a) : Curves for 16 processors

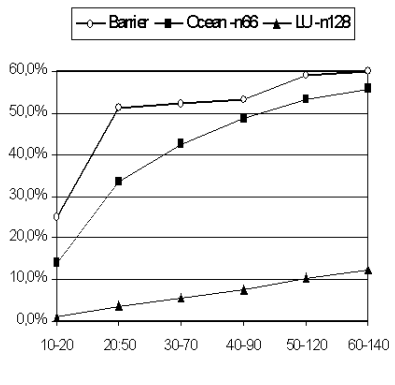

Figure (b) : Curves for 32 processors

As shown in these curves, the number of conflicts will increase significantly versus the latency, thus future architecture must pay more attention to this problem.

5 Conclusion

The technological advances permanently widens the performance gap existing in the parallel machines between the processors, the interconnection network and the memories. New solutions are explored to reduce this gap, but every time, the implementation of caches sets the data coherence problem.

The introduction of split transactions enables using new non-shared data networks, but to the cost of an increased complexity for managing and preserving the coherence. We described and evaluated two new solutions to this problem : Retry and Notification techniques. Retry technique is less expensive than the second technique since no additional buffers are required to memorize notification data. We think that it is practical for small classical SMPs where coherence conflicts are not so important (limited number of processors and reduced memory latency). However, notification technique performs better on large scale SMPs, and provides a better implementation flexibility for future address medium because it only snoops the medium and does not require a direct intervention on it (to abord a current transaction).

References

[1] D. Culler, J.P. Singh, and A. Gupta : Parallel Computer Architecture, A Hardware/Software Approach. Morgan Kaufmann Publishers, section 5.5.4

[2] M. Papamarcos and J. Patel : A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. The 11th Annual International Symposium on Computer Architecture, p. 348-354, June 1984

[3] James Archibald and Jean-Loup Baer : Cache Coherence Protocols : Evaluation Using a Multiprocessor Simulation Model. ACM Transactions on Computer Systems, 4(4) p. 273-298, November 1986

[4] Pradeep Sindhu, et al. XDBus : A High-Performance, Consistent, Packet Switched VLSI Bus. Proceedings of COMPCON, p. 338-344, Spring 1993

[5] Mike Galles and Eric Williams : Performance Optimisations, Implementation, and verification of the SGI Challenge Multiprocessor. The 27th Annual Hawaii International Conference on Systems Sciences, January 1993

[6] Technical information on the WEB : www.digital.com/alphaserver

[7] Technical information on the WEB : http://www.bull.fr

[8] Alan Charlesworth STARFIRE : Extending the SMP Envelope. IEEE MICRO, p.39-49, JAN/FEB 1998

[9] D. Kroft : Lockup-free Instruction Fetch/Prefetch Cache Organization. The 8th International Symposium on Computer Architecture, p. 81-87, May 81

[10] S.C. Woo et al. : The SPLASH-2 Programs : Characterization and Methodological Consideration. The 25th International Symposium on Computer Architecture, p. 24-36, june 1995

[11] S. Jourdan, P. Sainrat et D. Litaize : Exploring Configuration of Functional Units in an Out-Of-Order Superscalar Processor. The 25th International Symposium on Computer Architecture, p. 117-125, june 1995