Cache coherency and memory consistency in NoC based shared memory multiprocessor SoC architectures

Frédéric Pétrot and Alain Greiner
Département ASIM du LIP6
Université Pierre et Marie Curie
France

***Abstract*— This paper briefly reviews cache coherency and memory consistency problems in general purpose, non bus based, shared memory multiprocessors. These problems have been extensively studied in the past, and a huge amount of litterature is available on the subject. Although the SoC context is not (yet) the general purpose computing context, the increasing acceptance of NoC technology is such that the shared memory multiprocessor SoCs are already encountering these problems.**

SoC benefit on the one hand from better knowlegde of the applications, no address translation, and lower interconnect latencies, but on the other hand suffer from more tight design constraints. Taking these specificities into account, we try to define simple and yet efficient solutions to both problems.

I. Introduction

The increasing integration density will allow the building of Systems on Chip (SoCs) with from several dozens to hundreds of virtual components (or IP cores) within a four billion transistor chip before the end of this decade [1]. Interconnecting those components becomes the main architectural issue. Some recent works [2] have proposed the use of integrated switching networks as an alternative approach to interconnect IP cores in SoCs. IP core reuse is mandatory for such systems, and this reuse relies either on a clear separation between virtual component design and interconnect design or a widely accepted *de facto* standard. Open solutions, such as the Virtual Component Interface (VCI) standard [3] of the VSIA consortium or the OCP initiative of Sonics, advocate the use of wrappers that perform protocol conversion. Other well known proprietary solutions, for which separation of communication and conputation is not an issue, are also *de facto* standards, such as AMBA in several versions, including its announced AXI specification.

All these solutions conceptually involve point-to-point connections, implemented either as Network on Chip or as a bus hierarchy. In both cases, the interconnect is not a single, centralized, set of wires.

The paper first explains why caches are necessary even in the SoC domain, then briefly remind the reader with the cache coherency and memory consistency problems, and presents several actual solutions. From these solutions, we extract what we think is a good tradeoff beween complexity and performance for NoC based SoC, and present a few simulations to confirm this.

II. Use of caches

Many SoC designs are cost constrained and caches take area. Many SoC designs have real-time constraints and caches add undeterminism on execution delays. However, SoC design are also power constrained, and caches are saving power, and latency and thoughput may simply not be sustainable on demanding non-hard real-time applications like audio or video decoding. Also, a general trend in system on chip is the growing part of software components in embedded applications[4], and the knowledge of general purpose computing is that cache is a simple and efficient mean to increase software performance.

III. The problems

We make the following general assumptions[1]:

- The architecture contains two types of components plugged on the interconnect: initiators and targets. Initiators take the initiative of a read or write transfer. Targets are waiting for initiator requests to fullfill the read or write transfer. The initiators may cache data in order to firstly benefit from burst transfers to access data they will likely need and second to work on their local copy when several access to a data are required,
- The target architecture is build around a shared address space. All system components are connected to an interconnect that, if is latency is ignored for now, behaves as a cross-bar, allowing every initiator to request each target. The addresses are alike for all initiators. This means that an address uniquely identifies a datum in memory or a resource in a peripheral.

The figure 1 presents the reference shared memory architecture of the litterature. In this architecture, the request and response networks are independent, to avoid deadlocks, and many requests can concurently be pending or served.

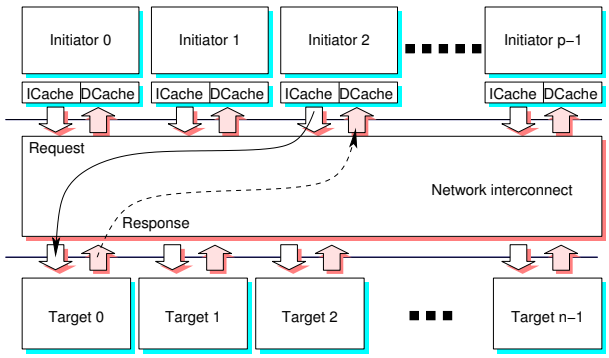

Fig. 1. Generic system architecture

[1]We use the VCI terminology here.

A. Cache coherency

The problem of cache coherency is illustrated on figure 2. This problem appears when data area are shared between threads running on different processors. If the processor P_1 writes a new value at address X, and the X address is cached by processor P_0, the data cached by P_0 should be invalidated or updated to reflect the fact that the value of the data at this address has changed. The classical approach to cache/memory coherency is to snoop the system bus[5]: As all cache controllers are directly connected to the system bus, each cache controller can spy the bus transfers and be informed of all write transactions between any processor and the memory banks. This allows the cache controllers to take actions if they cache a data at the address of the write operation. The action depends on the delayed (copy-back) or immediate (write-through) nature of the write. This approach unfortunately does not scale well due to the limited bus bandwidth.

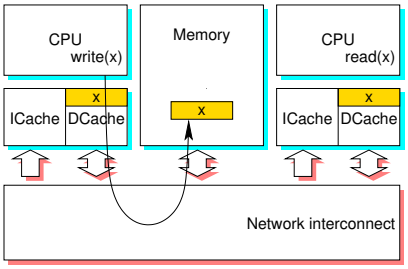

Fig. 2. The cache/memory coherency issue.

B. Memory consistency

All initiators can concurrently access targets, and a given target can receive requests from several initiators in sequence. In packet switched networks on which several routes are possible, there is no garanty that requests send in the network by a given initiator are recieved in the order of emission by the target. Furthermore, requests coming from different processors may have identical addresses. In that situation, the question is in which order should the requests be served ? This is what memory consistency is all about. For a complete dissertation on memory consistency issues, we refer the reader to [6].

Memory consistency is different from cache coherency in the sense that it is related to the programming model in use. We classically assume here that a multiprocessor program is a set of parallel threads, each thread being a sequence of machine instructions. We further assume that shared data are protected by locks. An *acquire* operation on a lock delays all the memory accesses occurring after the operation until the lock is indeed gained. A *release* operation frees the lock, but it can be seen that it is not necessary to wait until the completion of the *release* to perform the subsequent memory operations since they are by nature not protected by any lock. The compiler is allowed to reorder the memory access outside of the *acquire/release* blocks (as long as it does not violated data dependencies), but not inside such blocks.

This model is called *release consistency*, and it assumes that, for a given processor, the hardware will deliver the request to the same address in the order of emmission, and when several processors are involved, this will also be garanteed if a lock is aquired before the memory operations and released after them.

Having these constraints in mind, the problem is to enforce these orderings on packet switched networks.

IV. The solutions

A. Cache coherency

There are four main solutions to solve this problem:

- The first solution is to suppress all data caches in the system. Some applications such as network processing can run without data caches, but this cannot be a general solution,
- The second, purely hardware, approach is called *directory based* and was introduced back in 1978 by Censier and Feautrier[7]. It has been experimented in massively parallel shared memory multiprocessor architectures, and a recent and well known user of this approach is the DASH project[8]. This approach has a great advantage: it does not require any modification of the software. However, it is quite costly, as it implies to implement on the memory side a logically centralised, yet physically distributed directory. In this approach, the memory must be able to initiate a transfer, although it usually is a target, and it also requires the caches to be the target of control transfer, whereas caches are by definition initiators. The memory overhead in cache memory (2 bits per block) is negligible (and identical to what's required for copy-back coherency on busses) but the memory devoted to the directory grows like $O(\frac{m}{b}\,p^2)$, where p is both the number of processors and memory banks, m is the total memory size and b the cache block size. According to [8], this easilly reaches 20% of the total memory. If such a cost is acceptable for high-end parallel machines, this is clearly a dead-end for consumer electronic SoC designs,
- The third, purely software, approach is called *software cache coherence*. The idea is to invalidate, by software, a cache line when accessing a data that is known to be shared and dirty. If the shared nature is known by the program, the dirtyness of the data is dynamic and a conservative approach leading to systematic invalidation is required. This approach is costly in CPU time, leads to many unnecessary invalidations, and requires the use of write-through write buffers since otherwise the copy-back coherency would have to be handled by hardware some way,
- The fouth approach is also software oriented, but instead of trying to benefit from the cache behavior, it simply makes the shared data uncached and the local, non-shared data, cached[9].

B. Memory consistency

To ensure correct behavior regarding release consistency, the hardware, *initiator, targets and network* must garanty the following points:

- Packets issued by a given initiator to the same target are delivered in-order. This ensures the sequentiality of memory access for a processor,

- If a given initiator needs to send requests to two different targets, then it awaits the response from the first target prior to send the request to the second target. This avoid the situation where the release of a lock travels faster than the data it protects on a network, which clearly leads to inconsistency if an other initiator access the lock and the data,
- the interconnect is not allowed to arbitrary drop packets when it cannot handle them.

V. The proposed method

In the SoC context, having the shared data uncached and the local data cached seems promising because:

- the system integrator that maps the application on the target SoC plate-form knows well the data that are shared and the data that are not,
- the multiprocessors micro-kernels used in SoC are simple enough to be adapted to these programming constraints,
- the number of shared and local memory banks can be adapted to the application to minimise contention,
- the NoC latency is one order of magnitude smaller than parallel machine networks,
- the CPU directly use physical addresses, avoiding lengthy TLB flushes,
- it incurs no hardware cost at all.

For the illustration of the method, we assume that the parallel application is written as a set of t POSIX threads. We remind the reader that by default all the threads share the same address space, which is adequate for our programming model.

A. Tasks and data allocation model

The application programmer statically binds the t threads onto the p processors: Each processor is identified by an index, and this index is an explicit parameter of the thread creation primitive.

The memory is statically partitionned into two types of segments: local segments and shared segments.

A local memory segment L_i is defined for each processor P_i, therefore there are p local segments. This local memory segment L_i contains all private data used by the T_{ij} threads bound to the P_i processor. This includes (a) the execution stacks (automatic variables + caller and callee saved registers), (b) global per thread variables (POSIX key), (c) locally dynamically allocated memory (using a dedicated function) and (d) contexts of the T_{ij} threads.

As many shared segments S_i as necessary can be defined. We call the number of shared segments n. There must exist at least one such segment because the global variables are shared by definition. The number of shared segments is an output of the analysis of contention on the shared variables. Shared variables must be explicitly declared as such by the system integrator. All shared data (such as interthreads communication buffers) must be allocated in a shared segment.

B. Synchronisation

The synchronisation is necessary to gain exclusive access to data to ensure the release consistency model.

Having atomic test and set (or equivalent atomic operation) around a network is a difficult issue. A *read modify write (RMW)* opcode may exist. In that case, if the thread T_{ij} running on processor P_i makes a *RMW* access to a memory location X in shared memory, the thread T_{ij} gets an exclusive ownership of X until the next write access to X by the same T_{ij} thread. This semantic must be enforced by the target, and thus necessitate that all adresses in the target keep the indentifier of the initiator that performed a *RMW* transaction on it. In practice, this is not possible because it costs $\log_2 p$ (if the processors are the only initiators of the system) control bits per word. A typical implementation will simply prevent any other initiator to access the whole target, possibly leading to artificial deadlocks.

To overcome this practical limitation, we suppose that the architecture contains a *semaphore engine* that has a special behaviour: A *read* access is interpreted by the semaphore engine as an atomic *read then set to* 1. The thread that reads the value 0 at address X gets exclusive ownership of the X lock. A thread that reads a 1 is required to wait until it reads a 0. There is a strong software assumption: all threads should consider a 1 as a non crossing barrier and only the thread that owns the lock is authorised to reset it by issuing a *write* with value 0.

All locks in the system must be allocated in specific shared memory segments, that will be mapped on dedicated targets. Several targets may be necessary if for some reason a single semaphore engine becomes the system bootleneck.

C. Memory consistency

The required hardware behavior can be enforced as follows:

- in-order delivery to a given target can be done by tagging each request by an *(initiator-id, packet-id)* couple as suggested in VCI. The target memory is in charge of consuming the packets in increasing order of *packet-id* for a given *initiator-id*,
- avoiding that a memory operation overtakes the lock operation that protects it can be done by the initiator. When switching targets (the locks are in a specific semaphore engine), the initiator waits for the acknowledges of all requests to the first target before doing a request to the second target. (It requires that the initiator knows the targets mapping in memory).
  This illustrates the necessity of a response even in case of write operations. It is worth to note that the first version of OCB did not acknowledge the write, making it impossible to maintain memory consistency.

VI. Theoretical performance evaluation

The proposed approach allows the design to benefit from the cache efficiency with no hardware cost and at moderate software cost for the system integrator. In order to estimate the performance improvement, we compute the average number of cycles per instruction (CPI) obtained for a multithreaded application running on a multiprocessor architecture using the proposed software approach for cache coherence. Then me make the same CPI computation when all the data are considered uncached.

We make the following assumptions:

- Each processor executes one RISC instruction per cycle. We neglect the performance degradation introduced by the miss on the instruction caches,
- With a word width of 32 bits, and a typical network on chip, the average memory read latency is $l = 40$ cycles,
- All data caches implement a write-through policy, and the posted write buffers make negligible the cost of write,
- For the considered application the percentage of read instructions is 20%, we note this percentage $r = 0.2$. Among all read instructions, 10% address shared (uncached) data, we note this $ucr = 0.1$.
- The average data cache miss rate, noted mr, is 15%.

The CPI is by definition computed as the time spent in each instruction times the percentage of occurence of the instruction. With our hypothesis, the general CPI formula is: $\text{CPI} = (1-r)+r\cdot(ucr\cdot(l+1)+(1-ucr)\cdot((1-mr)+mr\cdot(l+1)))$

On a system with all data in uncached memory, we have 80% of the instructions that are not read and take 1 cycle to execute and 20% that are uncached reads and take 41 cycle to complete, thus: $\text{CPI}_{\text{uncached}} = (.8 + .2 \cdot 41) = 9$

With the proposed approach, we have 80% of the instructions taking 1 cycle. On the remaning 20%, 10% are uncached and 90% are cached and hit the cache 85% of the time, thus: $\text{CPI}_{\text{proposal}} = .8 + .2 \cdot (.1 \cdot 41 + .9 \cdot (.85 + .15 \cdot 41)) = 2.88$

As expected, exploiting the fact that only a small fraction of the data are actually shared improves the global performance by a factor of 3.

VII. Practical performance evaluation

These theoretical results must be confirmed by the experimentation. We use a Motion-JPEG decoder made of 6 tasks of different granularity that allows to decode p image flows concurently, each processor executing one decoding flow. The architecture includes a single semaphore engine and as many shared memory segment as processors. The processors are MIPS R3000, and each processor has independant 1K word instructions and data caches sharing the same NoC interface. The simulation is done at the cycle accurate level using the CASS simulator [10], available within the Disydent environment [11].

Since our interest is to measure the system performances either using a fully uncached approach or our proposed approach used in different context, the experiments are as follows: (1) Execute the application without data caches, (2) Execute the application with thread resources in local segments and communication resources in shared segments, (3) Execute the application with thread resources and communication buffers are in local segments,

The results, for 48×48 pixels movies of 24 images, are given in the figure 3. They confirm, on this example, the theoretical analysis.

VIII. Conclusion

The issues of cache coherence and memory consistency are more and more important in multithreaded applications running on SoC multiprocessor architectures.

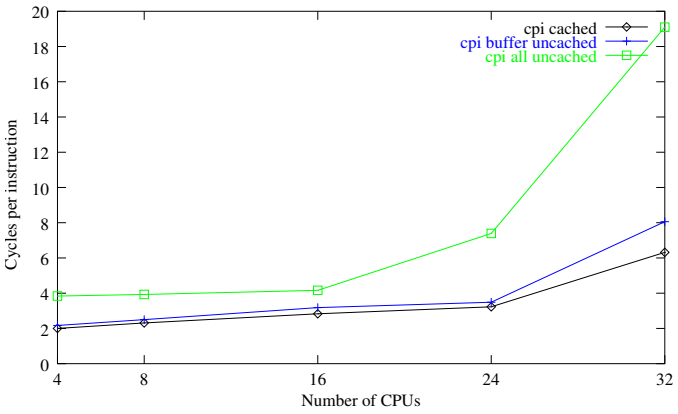

Fig. 3. CPI for the three experiments

We propose a simple, software oriented solution, taking advantage of the specific features of most systems on chip: No virtual memory, a single multithreaded application, and a large number of memory banks, thanks to the modularity permitted by integration on a single chip.

The performance improvement is expected to be at least a factor 3, versus the fully uncached solution. Cycle accurate simulations have confirmed those theoretical performance evaluations.

These experiments do not give an upper bound on performance, as it would be achieved (at a much higher hardware cost) by a directory based approach. How far is the current proposal from this has still to be appreciated.

References

[1] L. Benini and G. D. Micheli, "Networks on chips: A new soc paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[2] P. Guerrier and A. Greiner, "Architecture for on-chip packet-switched interconnections," in *Proc. of Design Automation and Test in Europe*, Paris, France, Mar. 2000, pp. 250–256.

[3] *Virtual Component Interface Standard (OCB 2 1.0)*, VSI Alliance, On-Chip Bus Development Working Group, Mar. 2000. [Online]. Available: http://www.vsi.org/library/specs/summary.htm#ocb

[4] "Medea+ eda roamap 4th release," 2003. [Online]. Available: http://www.medea.org

[5] M. Hill, S. Eggers, J. Larus, G. Taylor, G. Adams, B. Bose, G. Gibson, P. Hansen, J. Keller, S. Kong, C. Lee, D. Lee, J. Pendleton, S. Ritchie, D. Wood, B. Zorn, P. Hilfinger, D. Hodges, R. Katz, J. Ousterhout, and D. Patterson, "Design decisions in spur: A vlsi multiprocessor workstation," *IEEE Computer*, vol. 19, no. 11, Nov. 1986.

[6] K. Gharachorloo, "Memory consistency models for shared-memory multiprocessors," Ph.D. dissertation, Stanford University, Dec. 1995.

[7] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. Comput.*, vol. c-20, no. 12, pp. 1112–1118, Nov. 1978.

[8] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The stanford dash multiprocessor," *IEEE Trans. Comput.*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[9] F. Zandvelt, "On caches for a (rt) multiprocessor environment," Unpublished, Philips Research, Apr. 1998, prepared for the VSI/OCB working group.

[10] F. Pétrot, D. Hommais, and A. Greiner, "A simulation environment for core based embedded systems," in *Proc. of the 30th Int. Simulation Symp.*, Atlanta, Georgia, Apr. 1997, pp. 86–91.

[11] I. Augé, F. Pétrot, and D. Hommais, "A pragmatic approach to the design of embedded systems," in *Proc. of Design Automation and Test in Europe*. Munich, Germany: IEEE, Mar. 2001, pp. 170–174.