A RISC Approach to Weak Cache Coherence

J. Risau*, A. Mikschl, W. Damm

Carl von Ossietzky Universität Oldenburg, FB 10 Abteilung Rechnerarchitektur, Ammerländer Heerstrasse 114-118, D-26129 Oldenburg, Germany

Abstract. Data used by parallel programs can be divided into classes, based on how threads access it. For different classes of data different coherence mechanisms might be optimal.
This paper presents four primitives designed for use in a shared memory multiprocessor system, where each processor has its private cache. Using these primitives, programmers can implement those coherence models that are best suited to their applications. The paper gives a description of the primitives and some implementation details.

1 Introduction

A simple and intuitive memory model for shared memory multiprocessors is that of sequential consistency ([Lam79]). However, this model severely restricts the use of many optimization techniques in multiprocessor systems ([AH90]). Weak cache coherence models (see e.g. [Mos93] for an overview) often provide a restricted form of sequential consistency, namely only at certain points in time (synchronization points, e.g. in [BH90]), only for a subset of memory addresses (e.g. synchronization variables, [DSB88]) or only if tasks adhere to certain conditions ([AH90]), thereby allowing some optimization to be used. However, sequential consistency is not the only feasible memory model.

Many programs (tasks) are written to produce input determined results[2]. To do so, parallel threads of such a task compute some values, possibly using local variables, input data and values computed previously by other threads. They then assign some of these values to shared variables and finally signal that they have done so (e.g. they synchronize, using flags, semaphores, barriers and the like). Synchronizing allows other threads to use the values just computed for their own computation.

Data can therefore be (at least) divided into shared, private (local variables) and data used to synchronize threads. For different classes different coherence mechanisms might be optimal ([CBZ91]). However, hardware can usually not distinguish between these classes.

* email: Juergen.Risau@informatik.uni-oldenburg.de

[2] We restrict our discussion to this class of tasks. Other classes, like nondetermistic algorithms (e.g. chaotic algorithms) or tasks that don't relay on accurate, up to date data (e.g. some load balancing algorithms), are omitted for space reasons, but they, too, can be implemented using our primitives.

To adhere to these observations, threads should be given means to synchronize and choose which data they want to keep coherent. Following the RISC approach to provide primitives rather than solutions, we propose four instructions that were designed for a shared memory multiprocessor with private caches per processor. Using these instructions, threads can lock memory addresses and control some aspects of caching. The primitives can be combined to implement high level language synchronization primitives and coherence schemes.

The paper is organized as follows. In section 2 we present the new primitives. Section 3 gives implementation hints and section 4 shows further work.

2 The Primitives

We assume a shared memory multiprocessor where each processor has a private cache. Caches implement a write back strategy. *STORE* instructions do not cause any kind of coherence action, but merely write the value in the local cache. We propose the following new instructions:

- *LOCK address*
  This instructions locks the specified address if it was not already locked. If it was, then the instruction is only completed if *address* becomes unlocked.
- *UNLOCK address*
  The specified *address* becomes unlocked.
- *COPY_BACK node, address, nr_of_addresses*
  This instruction affects all lines in the cache of processor *node* containing data with addresses between (and including) *address* and *address* + *nr_of_addresses* -1. All dirty words, that is words that were written by the processor, from these lines are written back to main memory. Special values for *node* are **own** and **all**. A special value for *nr_of_addresses* is **all**, which affects all lines from the address space of the task issuing this instruction.
- *REPLACE node, address, nr_of_addresses*
  This instruction behaves exactly like *COPY_BACK* with the exception that lines are additionally invalidated (removed from the cache).

Essentially *LOCK* and *UNLOCK* serve as synchronization primitives. Implementing a binary semaphore is straightforward, since at most one thread can lock a given memory address at any time. General semaphores and other forms of synchronization can be implemented in a obvious way.

REPLACE and *COPY_BACK* are used to make data written by a processor visible to other processors. Threads can use these instructions to copy their results back to main memory and invalidate the corresponding lines in the caches of the processors running threads that are to read the results.

Since *STORE* actions do not cause any coherence actions, multiple processors can modify a copy of the same line in their caches. This usually happens because either threads don't synchronize or false sharing occurs. The problem of false sharing is solved by specifying *COPY_BACK* and *REPLACE* as is done above, vis. that they write only dirty words to main memory (if a line is written back due to the replacement policy of the cache, only it's dirty words must be

actually written, too). If threads write to the same word of the same line without synchronizing, it is undefined which word will be in main memory after the lines have been written. This usually is a result of a programming error[3] that would occur in sequentially consistent systems, too.

3 Implementation

The primitives presented in the last section have been implemented in a VHDL model of the WAMCOT-Architecture. WAMCOT is a shared memory NUMA architecture, with an optical bus ([AG92]), multithreaded processors ([AD96]) and a weak cache coherence protocol based on these primitives. The VHDL model showed the feasibility of the protocol but did not lend itself very well to a simulation with real benchmarks. Toy benchmarks yielded a processor utilization of up to 83%, which encourages us to further pursue our approach. Due to lack of performance results with real applications, we won't describe the implementation in detail here, but merely discuss some important aspects and costs.

To keep book about which addresses are locked, the memory controller of an architecture implementing our primitives must be equipped with a lock table. This data structure contains one bit for each lockable data item, that is, one bit per word in main memory. Implementations can of cause decide to make only a small amount of main memory lockable and leave it to the compiler or run time system to map a portion of this into the address space of each task. We choose a different approach and implemented one lock bit per so many words as are in a cache line. This implies that a thread that locks an address in fact locks the entire line containing that address. Since threads do usually not know which addresses are in the same line, a thread trying to lock two addresses from the same line will produce a deadlock. In our implementation, threads are therefore only allowed to lock at most one address at any time. Since threads can agree to exclusive access to any number of addresses by locking one of them, this is not a severe restriction. Assuming 32 bits per word and 8 words per line, our implementation requires 0.4% additionally memory to accommodate the lock table.

We have already mentioned, that an implementation of the primitives requires one dirty bit per word in a cache line. Apart from these, only 3 bits are needed for the status of each line. One is a valid bit which shows whether the words in the line contain valid data. The second is the request bit which is required due to the multithreading ability of the processor. The third is called lock bit and is an optimization to reduce bus traffic. Whenever a processor issues a *LOCK* instruction on an address that is already locked, the lock bit in the cache is set. If the processor tries to lock the address again, no access to the lock table is needed, because the information that the line is locked is already in the cache. An *UNLOCK* causes the memory controller to send invalidation messages to all caches, thereby clearing the lock bits, too. If the lock bit is not set, the state of

[3] in input determined tasks. Other classes of tasks might make use of this behavior.

the corresponding bit in the lock table is not known and so the processor must access main memory.

Order restrictions between instructions are kept to a minimum in this protocol: *STOREs* must have been performed (the value must be written in the cache) before either *COPY_BACK* or *REPLACE* can perform. *COPY_BACK* and *REPLACE* must be performed before an *UNLOCK* can perform (Values must be made visible before synchronization). Intra processor dependencies must be preserved, too.

4 Conclusion

We have presented four primitives that can be used by threads to implement those synchronization mechanisms and coherence models best suited to their needs. Since order restrictions are very few, most optimizations can be implemented in architectures employing these primitives. Simulations using toy benchmarks showed very good results. Further work will consist of several activities. The protocol will be compared to other coherence schemes and a more abstract model of our architecture will be implemented. This model will be used to gain performance results using real benchmarks from the SPLASH suite and or-parallel prolog programs. Properties of the protocol will be verified using a framework like that given in [Col92].

References

[AD96] A.Mikschl and W. Damm. MSPARC: A multithreaded Sparc. In *EuroPar 96*. Springer LNCS (this volume), 1996.

[AG92] Siemens AG. ESPRIT III - Project / Hierarchical Optical Interconnects for Computer Systems (HOLICS). Internal paper, Siemens AG, München, 1992.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. IEEE, May 1990.

[BH90] Lothar Borrmann and Martin Herdieckerhoff. A coherency model for virtualy shared memory. In *Proceedings of the International Conference on Parallel Processing*, August 1990.

[CBZ91] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of munin. *Symposium on Operating System Principles*, 1991.

[Col92] William W. Collier. *Reasoning about parallel architectures*. Prentice Hall International, Inc., 1992.

[DSB88] Michael Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronisation, coherence and event ordering in multiprocessors. *IEEE Computer*, 21(2), February 1988.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programms. *IEEE Transactions on Computers*, C-28(9), September 1979.

[Mos93] David Mosberger. Memory consistency models. *ACM SIGOP Operating Systems Review*, 27(1), 1993.