Efficient Synchronization Techniques in a Decentralized Memory Management System Enabling Shared Memory

Oliver Mattes, Martin Schindewolf, Roland Sedler, Rainer Buchty, Wolfgang Karl
Institute of Computer Science & Engineering (ITEC)
Karlsruhe Institute of Technology (KIT)

Abstract: The rising integration level enables combining more logic on a single chip. This is exploited in multiprocessor systems-on-chip (MPSoCs) or manycore research prototypes such as the Intel SCC. These platforms offer access to shared memory over a limited number of controllers which may lead to congestion. In order to scale the memory with the core count, the memory management must become more flexible and distributed in nature. In the near future decentralized systems with multiple self-managing memory components will arise.

The problem tackled in this paper is how to realize synchronization mechanisms for coincident access to shared memory in such a decentralized memory management system. Furthermore, improvements of the distributed synchronization mechanism are integrated and evaluated. To speed up the synchronization, additional logic in the form of a locks queue, is added. In order to reduce the network traffic this is combined by extending the synchronization protocol with exponential backoff. In the evaluation, side effects of combining both techniques are discussed and explained.

1 Introduction

The rising integration level in chip manufacturing enables combining more logic on a single chip every year. In the majority of cases this additional logic is not used to enhance the complexity of a single core, but rather to duplicate existing cores to build a multicore chip. The integrated cores may be less complex, but in total a higher performance can be achieved. By now multicore processors with 4 to 8 cores are state-of-the-art in the desktop market. Through combining multiple copies of these cores, higher numbers of cores are possible. In research, processors like the Intel Single Chip Cloud Computer (SCC) exist, containing 48 small Pentium-based x86-cores connected with a 2D-mesh on a single die [Int10]. If this trend continues, in the near future the core count in processors will likely scale up to more than 100. Similarly, also in embedded systems the core count rises: multiprocessor systems-on-chip (MPSoCs) are becoming increasingly widespread. In MPSoCs, multiple heterogeneous processing elements like processor cores or application specific accelerators are connected using a network-on-chip (NoC) and combined on a single chip with integrated memory components. An example for such a MPSoC architecture is GRAPES [MPSV06] with 4 to 16 cores.

Scaling the number of cores in a processor can be achieved rather easily. However, with the increasing core count, the memory has to scale as well. By now centralized mecha-

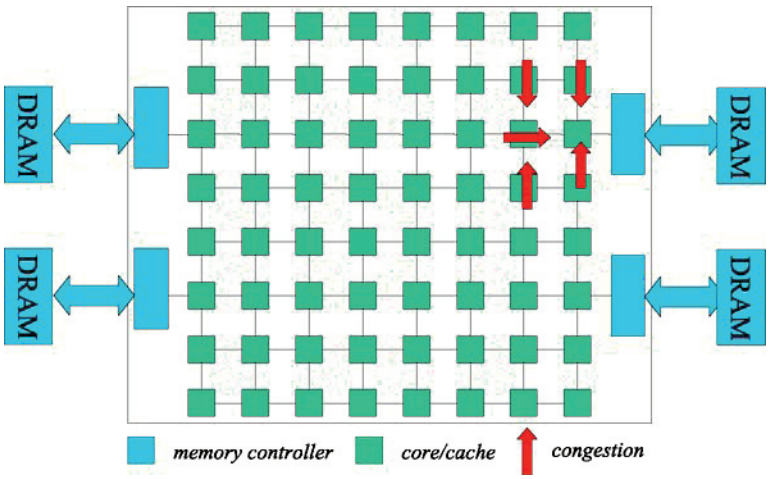

Figure 1: Memory Congestion [Dua09]

nisms for memory management are de facto standard. In both examples – Intel SCC and the mentioned MPSoC architecture GRAPES – the memory connection becomes a possible bottleneck if several cores perform memory accesses at the same time. In the GRAPES system architecture, all processor cores and coprocessors are connected via NoC to a single shared memory and a single main memory controller for non-shared data. All 48 cores of the Intel SCC are connected to memory using only 4 controllers. These controllers can be statically assigned to the running applications. Already when having few concurrent memory accesses, this kind of memory organization tends to memory controller congestion. An example for this situation is shown in Figure 1, where only a few concurrent accesses to a single memory controller, in here the one on the upper right side, lead to a congestion. Every additional memory access intensifies this problem. The problem of congestions will become even bigger with an increasing number of cores and a more intensive usage of shared memory.

This scenario could be prevented by scaling the number of memory controllers and splitting up the memory in several independent parts. Therefore memory management has to become more flexible and distributed in nature. Prospectively, decentralized systems containing multiple self-managing memory components will arise and be conceivable, possibly in combination with concepts like 3D-stacking of memory [Loh08]. As an example for such a system, Self-aware Memory (SaM) is given [BMK08]. With SaM, memory accesses are processed by self-managing system components. The memory is no longer tightly and statically coupled to a special compute resource, but can be dynamically assigned and exchanged through brokering mechanisms. These additional system components also provision shared memory. A more detailed introduction to Self-aware Memory is given in Section 2.

Accessing shared memory by multiple threads of a parallel application has to be coordinated to keep the data consistent. Otherwise parts of the applications could do calculations

on invalid and outdated data resulting in corrupt results. This coordination is done by synchronizing the parallel access to the memory using locking techniques based on shared lock variables like semaphores, mutual exclusions or monitors. Locks require hardware support by the use of atomic instructions such as test-and-set or atomic exchange.

The problem tackled in this paper is how to synchronize accesses to shared memory in a decentralized memory management system. We show how the decentralized memory management could be extended with synchronization primitives for supporting efficient access to shared memory. Furthermore, in order to reduce network traffic, we propose to combine a protocol extension (exponential backoff) and an extension of the hardware components (locks queue). The exponential backoff mechanism increases delay time between two successive attempts of acquiring a lock to prevent the flooding of the network by locking attempts. The locks queue is added to the memory component and enables specific signaling of the next waiting core. Both the exponential backoff and a locks queue have disadvantages for oneself. We examine the interactions of both mechanisms to get well-matched parameters so both methods complement each other.

Section 2 introduces the SaM architecture and briefly summarizes related work. Section 3 systematically investigates the adaptation of the synchronization primitives for the distributed memory system. A preliminary evaluation, including a discussion of experimental results, is presented in Section 4. Finally, Section 5 draws the conclusions and gives some directions for future work.

2 Background and Related Work

Self-aware Memory (SaM) [BMK08] represents a memory architecture, enabling self-management of system components for building up a loosely coupled decentral system architecture without central management instance. Traditionally, memory management tasks are handled software-based such as operating system and program libraries assisted by dedicated hardware components, e.g., the memory management unit or a cache controller. The main goal of SaM is developing an autonomous memory subsystem for increasing the overall system reliability and flexibility. This is crucial in upcoming computer architectures.

Firstly, SaM controls memory allocation, access rights, and ownership. The individual memory modules act as independent units and are no longer directly assigned to a specific processor. Figure 2 depicts the structure of SaM.

Due to this concept, SaM acts as a client-server architecture in which the memory modules offer their services (i.e., store and retrieve data) to client processors. The memory is divided into several autonomous self-managing memory modules, each consisting of a component called SaM-Memory and a part of the physical memory. The SaM-Memory is responsible for handling access to its attached physical memory, administration of free and reserved space, as well as mapping to the physical address space. As a counterpart of SaM-Memory, the so called SaM-Requester augments the processor with self-management functionality. The SaM-Requester is responsible for handling memory requests, perform-

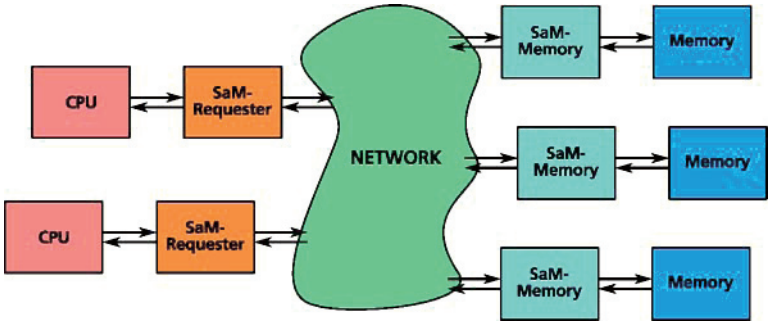

Figure 2: SaM Structure

ing access rights checks, and mapping of virtual address space of the connected processor into the distributed SaM memory space. With SaM, shared memory is realized by an interaction of the SaM Requester and SaM-Memory components by granting access rights of shared memory regions.

The distributed self-managing components are connected via a network. The basic SaM mechanism is independent of a distinct network structure or hierarchy. However the mechanism could be adjusted and fine-tuned depending on the actual network. Several structures as grids, stars or buses, in different hierarchies are possible. All interconnected self-managing network components can interact using the SaM mechanism. In the following a network on chip with a grid structure is taken as basis.

Prior related work established mutual exclusion locks as the de facto mechanism for concurrency control on shared memory data structures. A thorough introduction to the topic is given in [HP03], and some of the spin-lock alternatives for shared memory multiprocessors are presented in [And90]. Busy-wait techniques, building the base of most common routines, like mutual exclusion or barrier synchronization, generate large amounts of memory and interconnect traffic. Typically, the optimization of busy-wait synchronization exploits locality of cache memories. Such algorithms rely on cache-coherency protocols like MESI or MOESI [HP03, SS86].

A different approach is described in [MPSV06] in which a MPSoC with private off-chip memory and a small shared on-chip memory is presented. A dedicated hardware block, the synchronization-operation buffer (SB), augments the memory controller and stores requests issued by the processing elements. The SB manages the polling on shared locations locally, thus avoiding network traffic and memory accesses. In this particular setup, only a small performance improvement can be achieved. This may also be due to the fixed small (512 kB) on-chip shared memory, which limits the size of the working set. Further connecting the shared memory to the NoC through a single network controller leads to contention. In contrast to that, our approach features flexible assignment of memory resources to PEs, and no single network interface becomes a bottleneck. Also, allocation of shared memory up to the available size of the main memory is supported.

If contention for shared resources is high, exponential backoff [And90] can be employed

for throttling requests, leading to an effective reduction of network traffic. Hence, when a collision is detected, the processor exponentially increases (e.g., doubles) its mean delay time. Queuing locks can also significantly improve performance [HP03]. The idea is constructing a queue of waiting processors, denoted locks queue, and explicitly hand over the lock from one processor to another.

3 Integration of Synchronization Primitives

Software synchronization approaches rely on hardware primitives, which atomically read and modify a memory location. The return value is used for signaling whether the operation was successful or not. These hardware primitives represent the basic building blocks for a wide range of user-level synchronization mechanisms, including locks, barriers, semaphores, as well as more sophisticated abstractions like monitors. In the following, the integration of different basic synchronization primitives is presented and the evaluation methodology for their usage in such decentralized self-managing systems is explained.

We improve these mechanisms regarding bandwidth requirements by using exponential backoff on CPU and queuing on memory side, and study the effects and performance improvements. Moreover, the effects of combining both techniques are evaluated to analyze whether synergistic effects are observed.

Synchronization Primitives

Atomic Exchange The atomic exchange instruction uninterruptible interchanges a value in a register for a value in memory. The instruction allows to implement locks in a straightforward manner. Thus, a return value of 0 indicates that the lock is free, whereas a 1 signals that the lock is unavailable. In a distributed system, as motivated in the introduction, memory accesses are performed through messages over a network. Consequently, both a memory read and write can be implemented in a single, uninterruptible operation. If an atomic exchange instruction is executed, the memory management component checks access rights to the memory region, packs instruction and register values into a message, and sends that over the network to the corresponding memory component. On memory side, data is exchanged and the value is sent back to the CPU-side.

Test-And-Set A somewhat related instruction is test-and-set, which tests and sets a value if the value passes the test. However, implementing test-and-set is more cumbersome in a distributed system. Basically, two possibilities exist. Either, the memory can be enabled to perform the test locally. This approach facilitates a simple implementation very similar to the aforementioned atomic exchange. Or, the test can be carried out by the CPU. However, this alternative requires several additional message transfers between CPU and memory for locking the memory region, setting the value, and unlocking it. Locking and unlocking has to be done even if the test fails.

Therefore, enabling memory, more specifically the SaM-Memory component, to execute the test locally offers better performance with minimal hardware costs (i.e., comparator logic).

Load Linked/Store Conditional Using the exchange operation for implementing spin locks in a multiprocessor system bears additional costs associated with cache coherence protocols [HP03]. Consequently, a non-blocking abstraction based on special load/store instructions is the better choice. The load-linked (LL) and store-conditional (SC) instructions are used in sequence. Several threads can concurrently execute LL to the same memory address, but only one will successfully complete the SC. Typically, the instruction pair is implemented by storing the memory address specified by the LL instruction in a link register, and relying on a cache-coherency protocol to track stores to this location. When the cache line is invalidated, the corresponding content of the link register is cleared. Then a subsequent SC will fail.

The SaM system has to be extended with a so called link table on the memory side, holding the contents of the link register of each processor having access to a shared memory region. In the worst case, the link table has just as many entries as there are processors in the system. An LL instruction creates or updates an entry in the link table, whereas a successive SC instruction invalidates the corresponding entry. SC completes successfully if, and only if, the associated link table entry is valid and the store address matches the one in the table. The memory then signals the CPU side whether the SC succeeded. The design needs to address the synchronization of the link register and link table contents.

4 Evaluation

In this section, we present performance numbers of synchronization mechanisms and study the impact of system parameters on performance. For this evaluation we choose atomic exchange as a synchronization primitive. We study the effects of combining the exponential backoff mechanism with queueing locks. In particular, we vary the size of the locks queue and the initial delay time of the exponential backoff. The experiments comprise multiprocessor systems with 10 and 20 processors.

To achieve experimental results a SystemC-based simulation prototype was realized implementing and combining a set of modules for the UNISIM simulation environment [UNI]. As our prototype, a simulation model was extended by adding self-management functionality to the memory components. On CPU-side, the corresponding units for handling the communication protocol between the CPUs and the memory are integrated. The number of integrated components is parameterizable as well as the topology of the underlying network (NoC). Changes on connection parameters are dynamically adjustable. The network traffic can be monitored and analyzed. This enables evaluating the proposed synchronization mechanisms regarding network traffic and run time. Designated application scenarios are provided using memory traces on CPU-side to run through different memory access sequences.

As already mentioned in the previous section, exponential backoff and queuing locks can

be combined, complementing each other by decreasing the disadvantages of the individual method, to speed up lock-based synchronization. The system performance heavily depends on the number of processors in the system, the type of the interconnect, the memory subsystem, and the characteristics of the executed applications. For benchmarking, a simple multiprocessor system was considered: the system consists of n identical processors, and one memory module. The memory architecture does not include any caches. The network provides a connection between each SaM-Requester and SaM-Memory. Since SaM-Memory can buffer only one memory request at a time, pending messages are queued by the interconnect subsystem until SaM-Memory is ready to receive a new request. The memory traces represent different processors executing very similar applications, accessing lock-protected shared data; hence, each processor tries to acquire the lock before entering the critical section. The critical section comprises load/store instruction pairs, which first read and then overwrite shared memory locations. The critical section is executed within a loop. The size of the critical section (i.e., the number of load/store instruction pairs) and the number of loop iterations are the same for all the processors in the system. If the critical section is executed only once, the processor, which manages to acquire the lock first, does not spin on the lock at all. In order to avoid this situation, for benchmarking, the loop was iterated twice.

The value of the initial delay impacts the efficiency of the exponential-backoff mechanism. Fig. 3 shows the benchmarking results for a 20-processor system and different initial delay values. In all cases, the locks queue size is one.

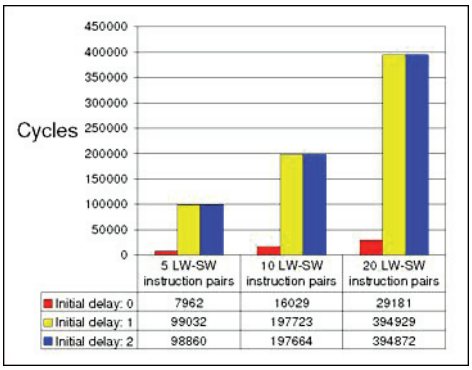

Figure 3: Exponential backoff: Impact of the initial delay on the system performance - the size of the locks queue is one in all the cases.

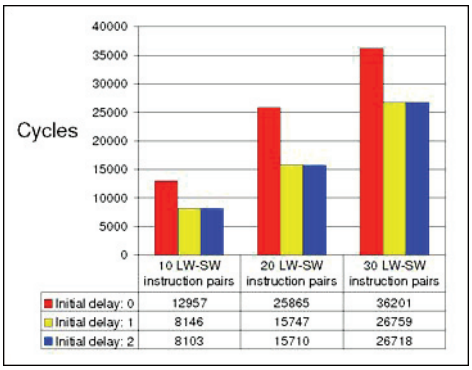

Figure 4: Exponential backoff: Impact of the initial delay on the system performance - the size of the locks queue is five in all the cases.

If the locks queue can accommodate only one lock request, all but one of the spinning processors produce network traffic, thus slowing down the thread in the critical section. With exponential backoff, the SaM-Requester doubles the delay on each failed attempt to acquire the lock. Consequently, when the lock is released, the mean delay of the spinning processors is already large. Furthermore, since the lock will be handed over to the processor in the locks queue and only one lock request can be buffered, the delay of the spinning processors will increase even further. Subsequently, depending on the value of the initial delay, it is possible that, although the lock is free and there are threads polling

the lock, the system makes no progress. This is because the SaM-Requesters do not forward the lock acquire requests to SaM-Memory. As shown in Fig. 3, long critical sections favor this behavior. On the other hand, if the initial delay is zero, the SaM-Requester forwards each acquire lock request to the corresponding SaM-Memory; in this case, the exponential-backoff mechanism is inactive. The results indicate that for the benchmarked system with a locks queue of one significant speedups can be achieved by disabling the exponential backoff mechanism.

One may wonder whether exponential backoff is not actually slowing down the system. As illustrated in Fig. 4, if SaM-Memory can queue more than just one lock request, exponential backoff can really help improve performance: the results refer to a system identical with the 20-processor system from above, except that SaM-Memory can now buffer up to five lock requests. Considering the range of the values on the y-axis in both figures, the results in Fig. 4 are better than those presented in Fig. 3 because of a the well-matched parameters at the combination of both methods. Consequently, in the following benchmarks, the initial delay is set to zero if the size of the locks queue is one; otherwise, the initial delay is always one increasing each attempt.

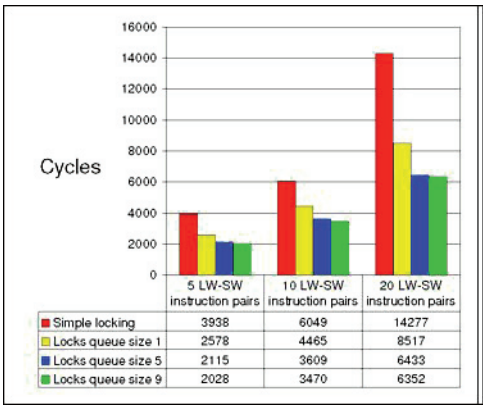

Figure 5: Locking: Simulation results for a 10-processor system

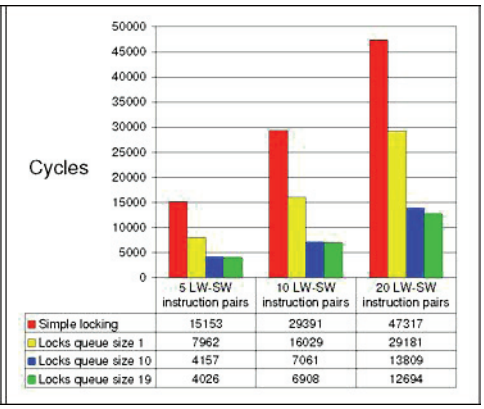

Figure 6: Locking: Simulation results for a 20-processor system

After determine principles for the exponential backoff dependent on the size of the locks queue, in turn the size of the locks queue dependent on the number of involved processors is examined. Fig. 5 presents the results for a 10-processor system and different critical section sizes. The unoptimized lock implementation, denoted simple locking, does not employ exponential backoff, nor any queuing locks techniques. The results confirm the expectations. Hence, being able to queue even just one lock request increases the overall performance: the longer the critical section, the higher the absolute performance improvements. Furthermore, increasing the queuing capacity of SaM-Memory pays off in better performance. However, a saturation effect can be noticed. Thus, being able to queue all the lock requests (i.e., a locks queue size of nine in this case) does not bring the expected benefits compared to a system with a much smaller queuing capacity. This is mainly due to the exponential-backoff mechanism, which effectively reduces the network traffic.

Benchmarking results for a 20-processor system (see Fig. 6) show that exponential backoff combined with queuing locks techniques can perform up to two times better than simple locking: the achieved speedup depends on the number of lock issues, which can be queued, and the initial delay value employed by the exponential backoff mechanism.

The implementation of atomic instructions was tested thoroughly. Different scenarios (e.g., illegal memory accesses, context switches) and system configurations (e.g., single- and multi-processor systems) were considered.

Concluding from the results presented in this section, the initial delay for the exponential backoff mechanism is determined to be one if more than one lock can be queued. With this setup, a reasonable compromise between initial delay and performance gains is achieved. In addition, the size of the locks queue should be scaled with the number of processors in the system to obtain the best performance. Considering the saturation effect of queuing locks, the size of the locks queue is determined according to:

$$\text{size(locks queue)} = \left\lfloor \frac{\text{\# processors}}{2} \right\rfloor \tag{1}$$

With the parameters determined in this paper, a well-matched combination of both methods is provided. This parameter setting causes the smallest runtime of applications using shared memory in such a decentralized system and exhibits the best scalability in the number of processors.

The obtained results of the evaluation match the theoretical considerations, because a trade-off between the waiting time, dependent on the exponential backoff, and additional hardware for the locks queues is achieved, in order to get high performance with least possible overhead, restricting the additional network load by the synchronization messages.

5 Conclusion and Outlook

In this paper, extensions to a trend-setting decentralized memory management system are proposed for enabling synchronization of shared memory accesses. Further, mechanisms to improve performance while reducing the network traffic are presented. For the evaluation a SystemC-based prototype is utilized, using selected memory access traces. With this evaluation of the proposed techniques, fitting parameters for the size of the locks queue and the exponential backoff are determined. At the present time the presented results are repeatedly evaluated using traces, generated with the SPLASH-2 or PARSEC benchmark suites.

This work contributes to efficient synchronization in MPSoCs. Nevertheless, new synchronization paradigms favoring optimistic instead of pessimistic synchronization will be needed for further scalability improvements. Thus, transferring the semantics of transactional memory will be the next step on our agenda.

References

[And90] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors. In Transactions on Parallel and Distributed Systems, IEEE, 1(1):6–16, Jan 1990.

[BMK08] R. Buchty, O. Mattes, and W. Karl. Self-aware Memory: Managing Distributed Memory in an Autonomous Multi-master Environment. In Architecture of Computing Systems - ARCS 2008, Lecture Notes in Computer Science, Volume 4934/2008:98–113, Springer Berlin / Heidelberg, 2008.

[Dua09] José Duato. Beyond the power and memory walls: The role of HyperTransport in future system architectures. In *First International Workshop on HyperTransport Research and Applications (WHTRA)*, February 2009.

[HP03] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufman Publishers, San Francisco, CA, USA, Third edition, 2003.

[Int10] Intel Labs. *The SCC Platform Overview*. Intel Corporation, May 2010.

[Loh08] Gabriel H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[MPSV06] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. An Efficient Synchronization Technique for Multiprocessor Systems On-Chip. In SIGARCH Computer Architecture News, 34(1):33–40, 2006.

[SS86] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In SIGARCH Computer Architecture News, 14(2):414–423, 1986.

[UNI] UNISIM. UNIted SIMulation Environment. `https://unisim.org/site/start`.