Customizable Datapath Integrated Lock Unit

Pekka Jääskeläinen, Erno Salminen, Otto Esko and Jarmo Takala
Tampere University of Technology
Department of Computer Systems
Tampere, Finland
Email: {pekka.jaaskelainen, erno.salminen, otto.esko, jarmo.takala}@tut.fi

***Abstract*— *Multicore Application-Specific Instruction-Set Processors (MCASIP)* offer an interesting alternative for implementing parallel applications in MPSoCs. Flexible MCASIP architecture templates allow matching the instruction and task level parallelism provided by the processor to the requirements of the application at hand.**

The processing throughput provided by shared memory (SM) multicores is commonly limited by the SM bandwidth. Synchronizing the execution of multiple threads using lock variables residing in the SM further adds to the bottleneck. In this paper we present a technique to reduce the SM contention in the case of MCASIPs where application-specific hardware customization can be used. The proposed solution is to use customized *Datapath Integrated Lock Units (DILU)* that enable the implementation of light weight synchronization primitives which minimize SM traffic.

The paper presents an experiment with a 48-core MCASIP which shows that the SM impact of the proposed fast barrier based on DILU in comparison to a basic SM polling one is up to 64% smaller. The size of the DILU hardware is negligible.

I. Introduction

Multicore Application-Specific Instruction-Set Processors (MCASIP) extend the design space of single core ASIPs to multicores. The additional customization point of core count enables matching the task level parallelism provided by the processor to the requirements of the application at hand. However, improving the performance by adding more cores to the design is often hindered by the overheads due to the need to synchronize the execution of multiple threads running in multiple cores.

Locks, barriers, and semaphores are basic synchronization primitives used to orchestrate the execution of a multithreaded program. Lock variables typically reside in shared memory and atomic *Read-Modify-Write (RMW)* instructions are needed for manipulating them without corruption. A lock performs mutual exclusion to ensure critical sections that manipulate shared data structures are executed only by one thread at a time.

A common synchronization primitive heavily used in system code is the *spin lock*. It performs busy wait until the *lock variable* is marked "free" (usually by writing 0 to the variable). It can be implemented with a loop that "spins" until it manages to write 1 the target lock variable before other cores do. The atomicity of the lock acquiring operation can be implemented with RMW instructions in the instruction set.

Another common synchronization primitive used especially in massively parallel programs is the *barrier*. Barriers are used to synchronize the control flow of all threads co-operating in the execution of the multithreaded program. The semantics of the barrier is to wait at the barrier call site until all other threads have reached the barrier. After all threads have reached the barrier, the execution of the threads continues freely. Simplest barrier implementations use counter variables that are protected with locks. They count how many threads have reached the barrier and block the waiting threads until the counter reaches the total number of threads.

Unfortunately, the polling during spinning the lock variables causes spurious traffic to the shared memory hierarchy causing unnecessary slowdown to other threads. The need to reduce shared memory contention due to synchronization led us to design the proposed hardware lock unit. The novel feature is to expose a simple address lock book keeping hardware to the programmer by integrating it to the processor datapath. Using this *Datapath Integrated Lock Unit (DILU)* the programmer is free to implement various atomic operations in software using the lock variable book keeping functionality for mutual exclusion.

Exposing the lock unit has the benefit of avoiding shared memory accesses altogether in some synchronization scenarios. The flexibility of DILU enables tailoring both the software and the hardware for implementing the synchronization primitives to match the varying synchronization demands of MCASIP designs. The results with a 48-core MCASIP prove the benefits of the flexibility. With a shared memory heavy workload the proposed fast barrier alternative which consumes two lock registers per barrier reduces shared memory contention up to 64% in comparison to a simpler version that consumes only one lock register.

The rest of the paper is organized as follows. The next section studies the related work, which is followed by a description of the context of the proposed work. Section IV describes the proposed solution that is evaluated in Section V. Finally, the paper is concluded in the last section.

II. Related Work

Techniques to reduce overheads of software based synchronization implementations have been studied widely. For example, *Adaptive Backoff Techniques* use simple heuristics to compute a time to wait before polling the barrier variables again to reduce the traffic [1], [2], [3], [4]. These approaches usually rely on a cache coherent memory hierarchy to enable fast spinning on a local cached copy of the lock variable. However, cache coherence hardware brings additional chip

area costs to the design which are avoided in embedded multicores with explicitly accessed local memories.

Another approach is to use dedicated synchronization hardware. One of the earliest works is presented by *Beckmann* and *Polychronopoulos* in [5] that supports barriers by means of a barrier register hardware. Each 1-bit register denotes whether each processor has reached the barrier or not. However, their work relies fully on hardware, while our proposal goes a step towards the software side, thus adding more flexibility.

The *Synchronization-operation Buffer (SB)* [6] reduces the spinning overheads by performing the polling independently from the processor core using dedicated hardware unit in the memory block. The unit sends notifications to the processor after a memory location changes its content to a desired value. Like our method, also SB avoids the need for coherent caches. However, SB monitors the shared memory bus for updates to the interesting variables, in contrast to DILU which separates itself completely from the shared memory.

Distributed Synchronization Controller (DSC) [7] is an approach close to ours. Both consume two lock registers per barrier. The main difference is that in DSC the monitoring synchronization traffic and updating the synchronization registers happens in hardware while DILU generalizes the concept of lock registers and pushes the main synchronization implementation logic to software side. Similar ideas are used in the *test-and-set registers* of *Intel's Single-chip Cloud Computer (SCC)* [8]. However, in case of SCC, the lock registers are mapped to memory and their number is limited to one per core which reduces the options for fast synchronization primitive implementation.

DILU resembles SCC, DSC and SB in their idea of isolating the synchronization logic to an independent hardware block. The distinctive feature of DILU is that it makes the hardware as simple as possible and exposes the lock register manipulation operations to the instruction set of the processor. DILU does not use a shared bus for monitoring or communicating the synchronization activities, but an arbitrated access to a set of synchronization registers. Moving complexity from hardware to software provides more flexibility for implementing the synchronization software library.

III. Context of the Work

The experiments for this paper have been conducted using an MCASIP design environment called TTA-based Co-design Environment Multicore (TCEMC) [9]. TCEMC allows the co-design of application-specific multicores based on a customizable single core template. Using TCEMC it is easy to experiment with custom instructions such as the lock unit instructions proposed in this paper and to produce multicores with an arbitrary number of cores.

The overview picture of the TCEMC MCASIP architecture template is shown in Fig. 1. The template is homogeneous in its single core instruction set architecture. However, the memory architecture is optimized for distributed execution of threads using fast core local memories. As shown in Fig. 1, each core has access to a local memory and the shared

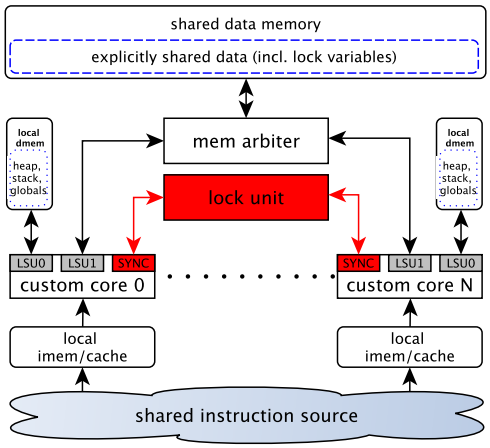

Fig. 1: TCEMC MCASIP template, the Local Default Data Memory (LDDM) version. In LDDM each tailored homogeneous core (number of cores can be customized) uses a private disjoint local memory for thread stacks and heaps. Shared memory is strictly for shared data which includes the possible synchronization variables. The proposed lock unit is accessed via the SYNC function units integrated to each core. LSU stands for *Load-Store Unit*.

memory. The address spaces of the two memories are disjoint and accessed with separate load-store units (and instructions). This configuration allows the stack, the heap and the global data of each thread to reside in the fast local memory while thread scheduler book keeping data and shared application data can be stored in the slower shared memory. The programs are either hand-optimized or compiler-optimized to explicitly use the local memories for speeding up the execution. In other words, caches are omitted from the microarchitecture and the shared memory accesses are assumed to be always very slow. This enables implementations with tens of cores with minimal complexity increase to the memory system.

Fig. 1 also includes the proposed lock unit which can be used to reduce shared memory accesses due to synchronization. The lock unit is accessible from each core's datapath using a *SYNC* function unit.

IV. Datapath Integrated Lock Unit

The key concept in the proposed lock unit is datapath integration. In practice it means that the basic operations to handle locks are presented to the programmer as processor instructions. The goal in the hardware design was to produce a simple enough lock unit that can be easily generated automatically according to the needs of the MCASIP design at hand while providing enough flexibility to support various synchronization primitive implementations in software.

A. Lock Unit Hardware Design

The *Lock Unit (LU)* hardware consists of a customizable sized *lock register* file. The number of registers, connected cores and address bits are configurable. Each lock register stores the status of a shared memory address currently being locked. Unlocked shared memory addresses do not consume

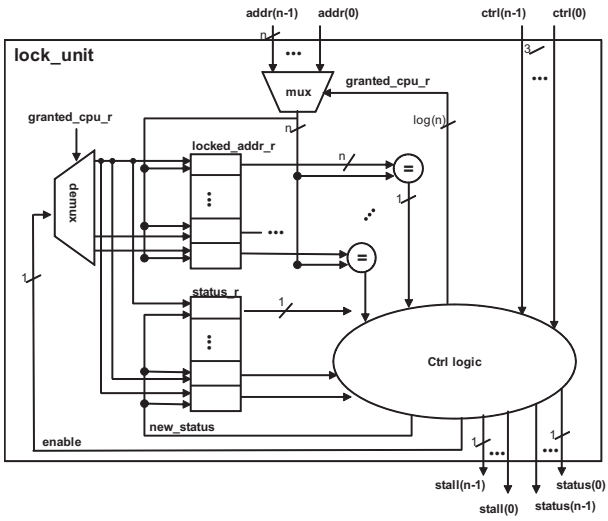

Fig. 2: The hardware design of the lock unit.

```
// spinlock_fast

lock_lu(lock_var* A) {
    while (!TRY_LOCK(A));
}

wait_lu(lock_var* A) {
    while (!READ_LOCK(A));
}
```

```
// spinlock_basic:

lock_var global_lock;

spin_lock(lock_var* A) {
 try_lock:
    lock_lu(&global_lock);

    if (*A == 1) {
        unlock_lu(&global_lock);
        goto try_lock;
    } else {
        *A = 1;
        unlock_lu(&global_lock);
    }
}
```

Fig. 3: Two spin lock implementations using the lock unit instructions. The *fast* version uses the lock unit operations solely while the basic version uses a single global lock register to guard accesses to all lock variables. *wait_lu()* will be used in a later barrier example to implement spin waiting on a lock register without lock acquisition.

any registers. LU ensures there is at most one lock register reserved per address at a time.

The hardware design of the LU is shown in Fig. 2. The inputs from the cores are shown on the top and outputs on the bottom. Each core provides an address and a 3-bit command. LU can stall the core until it can be served, and after that, LU gives the status of the completed operation.

The current implementation of the LU needs two cycles for each operation: register update and status reporting. During the status cycle it uses round-robin to arbitrate which core gets the next access to the lock register file.

Most of the chip area of the LU hardware is consumed by the register file and the comparators that find the slot to be reserved or released. In addition, the comparators are used to prevent double-locking of an address. The lock register file is split to two blocks in the picture: locked address and associated status bit. The lock status(i) is 1 in case the address register (i) contains a valid locked address and 0 otherwise. The control logic is rather small. It controls the stalling of the cores and produces the lock status based on the comparator outputs and the lock status bit.

The HW unit supports 4 basic operations: *read*, *lock*, *unlock*, and *wait until unlocked*. *Read* and *unlock* operations are non-blocking, while the *wait until unlocked* and *lock* can be implemented in both ways. The instruction set presented in this work utilizes only the non-blocking operations.

B. Synchronization Software

Three instructions are used to access the lock unit. All of them are non-blocking and take a memory address of a shared memory lock variable as an operand. The instructions use the address as a key to query the lock status, to attempt to acquire a lock, and to free a lock. Using these lock instructions it is possible to implement variations of synchronization functions that use a varying number of lock registers in the lock unit with the tradeoff of required shared memory accesses.

The instruction set in the *SYNC* function unit included in all cores is as follows:

got_lock := TRY_LOCK A	Tries to acquire a lock at address A.
UNLOCK A	Unlocks the lock at address A.
status := READ_LOCK A	Reads the lock status of the address A.

Two alternative implementations for both the spin lock and the barrier were implemented for this paper: a basic one that reduces lock register usage and one that minimizes shared memory accesses by relying more on the lock unit registers.

The spin lock implementations are shown in Fig. 3. *spinlock_basic* uses a single global lock unit register to guard accesses to all lock variables residing in the shared memory. First, function *lock_lu()* is called and it blocks until the global lock is acquired. If the actual lock variable contained in shared memory address A is locked (contains value 1), the global lock is unlocked and the lock acquisition is retried by jumping back. This consists the "spin loop" which is retried until the lock variable is 0 and can thus be locked by the core. Depending on the guarded critical section length and a potential "spin backoff algorithm" used, this version generates heavy traffic to the shared memory bus during its spin wait.

On the other extreme, *spinlock_fast* in *lock_lu()* assumes that lock information can be stored fully into the lock unit registers, thus it does not access shared memory at all during mutual exclusion. All spinning is done by using the instructions of the lock unit. Thus, the number of these lock primitives that can be "alive" at the same time in a program is limited by the number of lock registers in the DILU. Another restriction is that the program should not rely on the shared memory value of the lock as that value is not updated by the spin lock implementation at all.

The programmer or the compiler is responsible for deciding which synchronization function version to use in which occasion. A middle ground implementation between the *basic* and *fast* that would reduce contention on a single global lock register would be to use two or more lock registers that each guard an even share of lock variable memory addresses.

```
barrier_fast(barrier_t* b) {
  int executing;

  lock_lu(&b->count_l);
  b->running -= 1
  executing = b->running;

  if (executing == -1) {
  /* init barrier */
      executing =
         b->running =
              b->total-1;
      lock_lu(&b->barrier_l);
  }
  unlock_lu(&b->count_l);
  if (executing == 0)
      unlock_lu(&b->barrier_l);
  else /* spin wait */
      wait_lu(&b->barrier_l);
}
```

```
lock_var global_lock;

barrier_basic(barrier_t* b) {

  int executing;

  lock_lu(&global_lock);
  b->running -= 1
  executing = b->running;

  if (executing == -1) {
  /* init barrier */
      b->running =
           b->total - 1;
  }
  unlock_lu(&global_lock);

  /* spin wait */
  while (b->running > 0) {}
}
```

Fig. 4: Two barrier implementations using the lock unit instructions.

TABLE I: Required lock registers and shared memory read (r) and write (w) accesses for the alternative lock unit based synchronization implementations. The shared memory accesses are per thread participating in the synchronization. C is a variable depending on the critical section length or the thread imbalance in case of the barrier. L is the number of locks or barriers in use at the same time during the program execution.

	# of shared memory accesses			Required
lock style	acquire	release	spin wait	# of lock regs
basic	$r+w$	w	$C \times r$	≥ 1
fast	0	0	0	$1 \times L$
barrier	init	reach	wait others	# of lock regs
basic	w	$r+w$	$C \times r$	≥ 1
fast	w	$r+w$	0	$2 \times L$

The barrier implementation alternatives are shown in Fig. 4. *barrier_basic* uses a counter variable in shared memory. The counter is used to count how many threads are still to reach the barrier. Updates to it are protected with a single global lock register. After acquiring the lock, a thread decrements the counter to denote that it has arrived. If it was the first one, the counter goes to -1 and the barrier must be set up. After releasing the global lock, it spin waits until the counter goes to zero.

The fast version, *barrier_fast*, uses a counter variable similarly, but consumes two lock unit registers per barrier, named *count_l* and *barrier_l*. The first lock guards the counter variable updates and the latter records the whole barrier status. The first steps are similar to the basic algorithm except the barrier initialization locks also the *barrier_l*. It will be unlocked by the last thread reaching the barrier, i.e. when the count of threads still to reach the barrier goes to 0. This version accesses the shared memory when each thread reaches the barrier the first time (read, decrement, write). However, after that it spins on the second lock register by calling *wait_lu()* which minimizes the shared memory traffic while waiting for the other threads.

Table I summarizes the shared memory and lock unit register costs of the alternative implementations. Locks can be implemented without shared memory access at all. In the both cases, the biggest difference is during spinning.

```
for (int round = 0; round < ROUNDS; ++round) {
    if (core_id == 0) {
       for (int delay = 0;
            delay < smAccesses;
            ++delay, ++shared_value) {}
    }
    barrier(&b);
}
```

Fig. 5: The microbenchmark that "stress tests" the shared memory overheads of the barrier alternatives.

V. Evaluation

The effect of using the proposed alternatives to the final performance of the application depends on various factors. First, it is affected by the synchronization operation per program operation ratio of the program. Furthermore, the shared memory pressure of the program is one important factor. In case the program performs frequent accesses to the shared memory, it is hindered more by the avoidable traffic caused by the synchronization.

A. MCASIP Hardware

For the evaluation of the proposed lock unit we designed a 48-core MCASIP using TCEMC. It should be noted that the results should be reproducible with any multicore architecture as DILU is designed to be generic enough to be integrated to any multicore's datapath.

All the cores in the designed MCASIP use the DILU for synchronization operations and have a relatively simple single core architecture with an integer ALU, an integer multiplier and a register file with 16 32-bit registers. The memory configuration of the MCASIP was as shown in the Figure of Section III. That is, each core had its own fast private data memory large enough for local stack and data in addition to an arbitrated uncached shared memory. The shared memory access time is from 4 to 96 cycles, depending on how many cores are competing for access.

The design was synthesized to an Altera Stratix II FPGA with a clock rate of 50 MHz. As expected, the additional area overhead of DILU was very low. DILU with four lock registers consumed about 1% (925 ALM) of the total logic utilization of the multicore (72 kALM).

B. Benchmark program

The scalability of the proposed barrier alternatives was measured by implementing a synthetic microbenchmark that performs a tight barrier synchronized loop and executing it in the MCASIP programmed to the FPGA. The benchmark loop is shown in Fig. 5.

The benchmark represents a "stress test" where the computation to synchronization ratio is extremely low and where the progress is heavily limited by concurrent shared memory accesses from different cores. Artificial shared memory traffic was generated by adding an update to a counter residing in shared memory to the first core. This forces the other cores to wait that time in the barrier spin wait loop. While being a

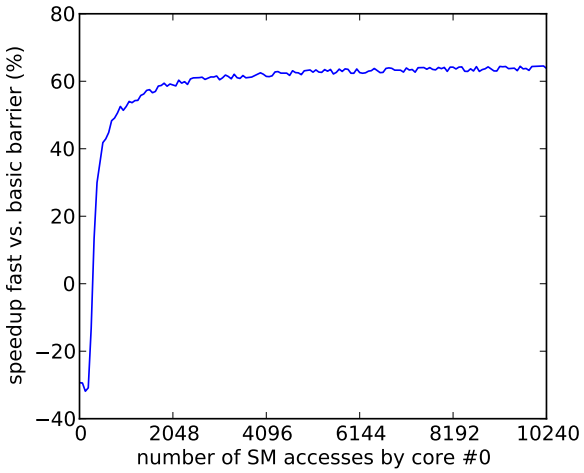

Fig. 6: The speedups obtained by using the *fast* barrier in comparison to the *basic* barrier version. The run times were measured as wall clock time with the FPGA implementation of the 48-core MCASIP.

synthetic example, the case represents the worst case of *thread imbalance* where one shared memory heavy thread takes more time to complete than the others and the completed threads cause it to slow down due to the SM traffic from barrier spin waiting.

In this benchmark, the number of threads equals the number of cores, thus the context switch overheads were ruled out. If there were more threads per core, the barrier call would induce a thread context switch to allow the waiting threads to reach the barrier. This would reduce the total "unsuccesful" spin waiting for the time of each context switch, possibly reducing SM noise. However, the context switch time and its effect to the performance depends at least on a) the size of the thread context to store/restore b) the location of the context data (SM or a local storage) c) the thread scheduler overhead d) if a pre-emptive scheduler is used, the number of pre-emptions, etc. Therefore, we decided to simplify this benchmark to emphasize the effect of the synchronization overheads alone.

The results illustrated in Fig. 6 show that the speedup from using the fast barrier that eliminates shared memory polling is drastic. The speedup increases up to about 3000 shared memory accesses after which it stabilizes to around 64%. For less than 320 accesses the *basic* barrier is faster as the additional barrier software complexity of the *fast* barrier dominates the shared memory access reduction benefits.

VI. Conclusion

A flexible *Datapath Integrated Lock Unit (DILU)* was proposed. DILU provides a middle ground solution between hardware based and software based synchronization implementations. The proposed hardware unit is simple enough to be customized according to the synchronization needs of the application at hand by varying the number of lock registers in the unit. Therefore, it is suitable to be used as a building block in co-design of MCASIPs. The unit does not require a coherent fast local memory to implement very low overhead spin lock and barrier synchronization, making it useful for multicore designs with simplified memory hierarchies.

The flexibility and scalability of the approach was shown with a 48-core MCASIP design. The results showed that in case of workloads with high shared memory access demands, the faster of the proposed barrier alternatives places about 60% less stress to the shared memory in comparison to the basic version with shared memory spinning.

In the future we plan to study compiler techniques to automatically optimize the usage of the different synchronization implementation alternatives and to further optimize the hardware implementation of the DILU.

Acknowledgment

This work has been supported by Academy of Finland. The authors wish to thank Mr. Lasse Lehtonen for his help in verifying and synthesizing the first implementation of the lock unit.

References

[1] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, pp. 21–65, February 1991. [Online]. Available: http://doi.acm.org/10.1145/103727.103729

[2] D. S. Nikolopoulos and T. S. Papatheodorou, "The architectural and operating system implications on the performance of synchronization on ccNUMA multiprocessors," *Int. J. Parallel Program.*, vol. 29, pp. 249–282, June 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=608719.608775

[3] T. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, no. 1, pp. 6 –16, jan 1990.

[4] A. Agarwal and M. Cherian, "Adaptive backoff synchronization techniques," in *Proceedings of the 16th annual international symposium on Computer architecture*, ser. ISCA '89. New York, NY, USA: ACM, 1989, pp. 396–406. [Online]. Available: http://doi.acm.org/10.1145/74925.74970

[5] C. J. Beckmann and C. D. Polychronopoulos, "Fast barrier synchronization hardware," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '90. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 180–189. [Online]. Available: http://portal.acm.org/citation.cfm?id=110382.110433

[6] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Efficient synchronization for embedded on-chip multiprocessors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, pp. 1049–1062, October 2006. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2006.884147

[7] C. Yu and P. Petrov, "Low-cost and energy-efficient distributed synchronization for embedded multiprocessors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 8, pp. 1257 –1261, aug. 2010.

[8] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.53

[9] P. Jääskeläinen, E. Salminen, C. Sánchez de La Lama, J. Takala, and J. Matrinez, "TCEMC: A co-design flow for application-specific multicores," in *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation 2011 (SAMOS XI)*, July 2011.