An FPGA Implementation of Wait-Free Data Synchronization Protocols

Benjamin Nahill[1], Ari Ramdial[1], Haibo Zeng[1], Marco Di Natale[2], Zeljko Zilic[1]
[1] McGill University, email: {benjamin.nahill, ari.ramdial}@mail.mcgill.ca, {haibo.zeng, zeljko.zilic}@mcgill.ca
[2] Scuola Superiore Sant'Anna, email: marco@sssup.it

***Abstract*—The synchronization of accesses to shared memory buffers in multi-core platforms can be realized through lock-based synchronization protocols. If the embedded application executing on the system has hard real-time constraints, the worst-case blocking times for accessing remotely shared resources can negatively impact the schedulability guarantee. In this case, wait-free communication protocols can be an effective alternative. In addition, in a model-based development process, wait-free buffers allow the realization of communication that provably preserves the signal flows and guarantees a correct implementation. Flow-preserving wait-free communication primitives require (in the general case) the execution of buffer updates procedures at task activation time, either by the kernel or by a hook procedure executing at the highest priority level. To minimize the interference of such procedures on the application-level tasks, we present and evaluate an FPGA implementation. Our FPGA implementation is compared with implementations of lock-based policies in terms of memory, time, and area overhead.**

I. Introduction

In the last decade, major advances in VLSI manufacturing allowed to scale down architectures to fractions of previous generations. This scaling and the corresponding cost reduction, together with opportunities for power reduction and performance increase are leading to the fast adoption of multicore architectures in embedded devices. Applications running on multiple cores are common in the automotive domain and are expected to increasingly support hard real-time functionality.

Real-time embedded and cyber-physical systems must provide computation tasks with a deterministic worst-case response time. In multi-core architectures, tasks can be scheduled with *partitioned* or *global* scheduling. Under *partitioned scheduling*, tasks are statically assigned to processors and each processor is scheduled separately. Under *global scheduling*, tasks are scheduled using a single shared task-queue. They are allocated dynamically and inter-processor migration is allowed. Global scheduling algorithms can provide more flexibility and load adaptation. However, the required job migration can incur significant overheads [9]. Also, partitioned scheduling is adopted and supported by commercial standards like AUTOSAR, and commercial real-time operating systems (e.g. VxWorks, LynxOS, and ThreadX). In this work, *we assume partitioned scheduling*.

The response time of a task includes not only its execution time, but also interferences from higher priority tasks and any possible blocking time when waiting for lower priority tasks. An important possible source of blocking time is waiting for the access to shared resources, since operations on them must be atomic. One important category of such shared resources is memory buffers, used for communication among tasks residing on the same or different cores. The procedures that write into or read from the buffers must guarantee the consistency of the data content and buffer management structures.

Data consistency protocols fall into one of the following three categories.

- *Lock-based*: tasks are prevented from accessing communication data unless they hold the lock. When a task wants to access the shared resource while another task holds the lock, it blocks. In multicore architectures with globally shared resources, the blocked task can either suspend operation and enter a global queue or spin on the lock [10] [7].
- *Lock-free*: each reader can access the communication data without blocking but repeats the process if it detects a concurrent write and has read an inconsistent value. The total number of repeats can be bounded.
- *Wait-free*: when the global shared resource is a communication buffer, readers and writers are protected against concurrent access by replicating communication buffers.

Wait-free (WF) communication protocols not only can be used to guarantee data consistency, but can be extended to provide flow-preservation: a property of interest when the system tasks are the implementation of a synchronous reactive (SR) model such as Simulink or SCADE. In model-based development, a flow-preserving implementation guarantees that the data values exchanged by tasks at run time are the same values that are validated (by simulation or model checking) on the functional model.

Several implementations of wait-free buffers with flow-preservation (WF-FP) are possible. In the general case in which no assumptions can be made on the activation events of the writer and the reader, the implementation requires procedures that are executed at task activation time, with highest priority. Since these procedures interfere with all application tasks, an efficient implementation is of highest

importance. In this paper, we present an FPGA implementation of wait-free communication mechanisms with flow preservation and analyze it against the implementation of lock-based methods with respect to memory and timing overheads. The implementation as programmable hardware enables parallelism at the cost of additional chip area (logic units).

The paper is organized as follows. We give a brief overview of methods for achieving data consistency and flow preservation in Sections II and III, summarizing the available approaches and algorithms. We then present our programmable hardware implementation of lock-based methods in Section IV and the FPGA implementation of wait-free mechanisms with flow preservation in Section V, including the design choices and issues. The analysis of the results and the evaluation of the implementation is in Section VI. The paper ends with the conclusions in Section VII.

II. Methods for Data Consistency

In this section, we discuss lock-based and wait-free mechanisms for data consistency in intercore communication. Lock-free mechanisms are less relevant for hard real-time embedded systems and are not considered.

A. Lock-based Methods

Among lock-based mechanisms, the multiprocessor priority ceiling protocol (MPCP) was developed in [10] for dealing with the mutual exclusion problem in the context of shared-memory multiprocessors. MPCP is the extension of the well-known Priority Ceiling Protocol (PCP) [11] for sharing resources in single processor systems.

During normal execution, tasks use their assigned priorities. When they enter a critical section, they inherit the ceiling priority of the shared resource. For local resources, the ceiling priority is the highest priority of any task that can possibly use it. In PCP, once a task begins its execution, it is guaranteed never to be blocked. In MPCP, a base priority higher than any task is applied to all global ceilings, so that the priority ceiling of any global resource is higher than any task priority in the system. Tasks that fail to lock a resource shared with remote tasks (global resource) are *suspended*, and placed in a global priority-based queue, allowing other local (possibly lower priority) tasks to execute. Unfortunately, this disrupts the guarantee of a single blocking time interval of PCP.

The MSRP (Multiprocessor Stack Resource Policy) [7] is the multiprocessor variant of the Stack Resource Policy (SRP) [2]. Locally, tasks access resources via SRP-protected critical sections (using a ceiling protocol similar to PCP). Global resources have a ceiling with highest priority in the system. Tasks that fail to acquire the lock on a global resource do not release their CPU but spin locally and are kept in a global First-In-First-Out queue.

MSRP has been compared with MPCP in terms of worst-case blocking times [7]. It performs better for short critical sections and worse for large access times to the global resources. The Flexible Multiprocessor Locking Protocol (FMLP) [3] combines the strengths of the two approaches. In FMLP, short resource requests use a busy-wait mechanism, while long resource requests are handled using a suspension approach.

Algorithm 1: Wait Free Method for Data Consistency - Writer [13]

```
   Data: BUFFER [1,...,NB]; // NB: Num of buffers
   Data: READING [1,...,n]; // n : Num of readers
   Data: LATEST
 1 GetBuf();
 2 begin
 3     bool InUse [1,...,NB];
 4     for i=1 to NB do InUse [i]=false;
 5     InUse[LATEST]=true;
 6     for i=1 to n do
 7         j = READING [i];
 8         if j !=0 then InUse [j]=true;
 9     end
10     i=1;
11     while InUse [i] do ++i;
12     return i;
13 end
14 Writer();
15 begin
16     integer widx, i;
17     widx = GetBuf();
18     Write data into BUFFER [widx];
19     LATEST = widx;
20     for i=1 to n do CAS(READING [i],0,widx);
21 end
```

Algorithm 2: Wait Free Method for Data Consistency - Readers [13]

```
   Data: BUFFER [1,...,NB]; // NB: Num of buffers
   Data: READING [1,...,n]; // n: Num of readers
   Data: LATEST
 1 Reader();
 2 begin
 3     constant id; // Each reader has its unique id;
 4     integer ridx;
 5     READING [id]=0;
 6     ridx = LATEST;
 7     CAS(READING [id],0,ridx);
 8     ridx = READING [id];
 9     Read data from BUFFER [ridx];
10 end
```

B. Wait-free Method

Wait-free methods can avoid blocking at the expense of additional memory buffers. In [4], an asynchronous protocol is proposed for preserving data consistency with execution-time freshest value semantics in the single-writer to single-reader communication on multiprocessors. An atomic Compare-And-Swap (CAS) instruction (required for atomic resource access in multicores [8]) is used to guarantee atomic reading position assignments and pointer updates.

Algorithms 1 and 2 outline the wait-free communication protocol for the writer and reader respectively [13]. The

writer locates an unused buffer and begins writing. Each reader obtains the index of the LATEST written buffer entry and sets it to be in use so that the writer cannot access it. A CAS operation is used to ensure that the buffer indexes in use by the readers (READING) are consistent. The number of buffers is sized to ensure that there is always a free buffer for the next value produced by the writer. While this method guarantees data consistency, it does so at increased runtime overhead as well as memory overhead since it requires buffer replicas.

III. Model-based development and Flow Preservation

Model-based design is very popular in the development of embedded control systems, because of the possibility to verify the functionality of controls by simulation and formal methods. The functional model is defined according to the Synchronous Reactive semantics using tools like MathWorks Simulink [1], and a software implementation is automatically obtained from a code generator.

A Simulink model is a network of communicating blocks. Each block operates on a set of input signals and produces a set of output signals [6]. The domain of the input function can be a set of discrete points (discrete-time signal) or it can be continuous (continuous-time signal). Continuous blocks are implemented by a solver, executing at the base rate. Eventually, every block has a sample time, with the restriction that the discrete part is executed at the same rate or at an integer fraction of the base rate.

A fundamental part of the model semantics is the rules dictating the evaluation order of the blocks. Any block for which the output is directly dependent on its input (i.e., any block with direct feedthrough) cannot execute until the block driving its input has executed. The set of topological dependencies implied by the direct feedthrough defines a partial order of execution among blocks.

Before Simulink simulates a model, it orders all blocks based upon their topological dependencies, and chooses one total execution order that is compatible with the partial order constraints. Then, the virtual time is initialized at zero. The simulator engine scans the precedence list in order and executes all the blocks for which the current virtual time is an integer multiple of the period of their inputs. Executing a block means computing the output function, followed by the state update function. When the execution of all the blocks that need to be triggered at the current virtual time instant is completed, the simulator advances the virtual clock by one base rate cycle and resumes scanning the block list.

The code generation framework follows the general rule set of the simulation engine and must produce an implementation with the same behavior (preserving the semantics). The Simulink/Embedded Coder code generator of MathWorks allows two different code generation options: single task and fixed-priority multitask. Single task implementations are guaranteed to preserve the simulation-time execution semantics. However, this comes at the cost of a very strong condition on task schedulability and a single-task implementation can be terribly inefficient in multicore systems. In multitask implementations, the run-time execution of the model is performed by running the code in the context of a set of threads under the control of a priority-based real-time operating system (RTOS). The code generator assigns each block a task priority according to the Rate Monotonic scheduling policy.

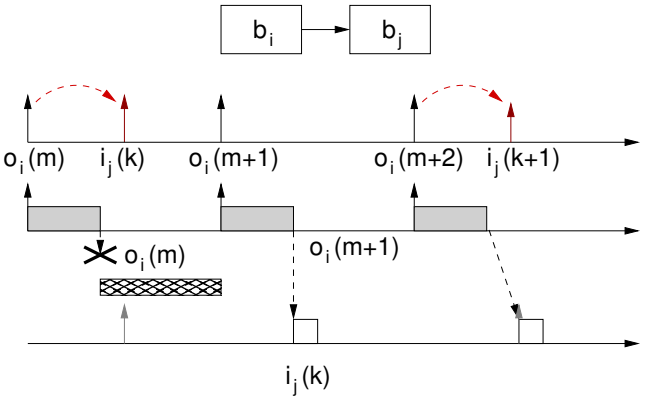

Fig. 1. A code implementation that does not preserve the flow of data from the writer to the reader block.

Because of preemption and scheduling, in a multi-rate system, the signal flows of the implementation can differ from the model flows. The bottom side of Figure 1 shows the possible problems with flow preservation in multi-task implementations because of preemption and scheduling. The timeline on top represents the execution of the block reactions in logical time. The schedule at the bottom shows a possible execution when the computation of the update function is performed in real-time by software code, and the code (task) implementing the k-th instance of the reader is delayed because of preemption.

In this case, the writer finishes its execution producing the output $o_i(m)$. If the reader performs its read operation before the preemption by the next writer instance, then (correctly) $i_j(k) = o_i(m)$. Otherwise, it is preempted and a new writer instance produces $o_i(m+1)$. In the latter case, the reader reads $o_i(m+1)$, in general different from $o_i(m)$.

Lock-based mechanisms do not guarantee the preservation of SR flows and are not meant to. A correct implementation of SR flows requires changes to the wait-free methods in Section II. The data item used by the reader must be defined based on the writer and reader task activation times. Both tasks, however, are not guaranteed to start execution right after their activation because of scheduling delays. Therefore, the assignment of the buffer index must be delegated to the operating system. At execution time, the writer and readers will use the buffer positions defined at their activation times. With reference to the algorithm presented in the previous sections, the writer

Algorithm 3: Wait Free Method for SR Flow Preservation - Writer [13]

```
   Data: BUFFER [1,...,NB]; NB: Num of buffers
   Data: READINGLP [1,...,n_lp]; // n_lp: Num of lower priority readers
   Data: READINGHP [1,...,n_hp]; // n_hp: Num of higher priority readers
   Data: PREVIOUS, LATEST
 1 GetBuf();
 2 begin
 3     bool InUse [1,...,NB];
 4     for i=1 to NB do InUse [i]=false;
 5     InUse[LATEST]=true;
 6     for i=1 to n_lp do
 7         j = READINGLP [i];
 8         if j !=0 then InUse [j]=true;
 9     end
10     for i=1 to n_hp do
11         j = READINGHP [i];
12         if j !=0 then InUse [j]=true;
13     end
14     i=1;
15     while InUse [i] do ++i;
16     return i;
17 end
18 Writer_activation();
19 begin
20     integer widx, i;
21     PREVIOUS = LATEST;
22     widx = GetBuf();
23     LATEST = widx;
24     for i=1 to n_hp do CAS(READINGHP [i], 0, PREVIOUS);
25     for i=1 to n_lp do CAS(READINGLP [i], 0, LATEST);
26 end
27 Writer_runtime();
28 begin
29     Write data into BUFFER [widx];
30 end
```

Algorithm 4: Wait Free Method for SR Flow Preservation - Readers [13]

```
 1 ReaderLP_activation();
 2 begin
 3     constant id; // Each lower priority reader has its unique id;
 4     integer ridx;
 5     READINGLP [id]=0;
 6     ridx = LATEST;
 7     CAS(READINGLP [id],0,ridx);
 8     ridx = READINGLP [id];
 9 end
10 ReaderHP_activation();
11 begin
12     constant id; // Each higher priority reader has its unique id;
13     integer ridx;
14     READINGHP [id]=0;
15     ridx = PREVIOUS;
16     CAS(READINGHP [id],0,ridx);
17     ridx = READINGHP [id];
18 end
19 Reader_runtime();
20 begin
21     Read data from BUFFER [ridx];
22 end
```

and reader protocols need to be partitioned in two sections, one executed at task activation time, managing the buffer positions (READINGLP[], READINGHP[], PREVIOUS and LATEST), the other at runtime, executing the write and read operations using the buffer positions defined at their activation time as shown in Algorithms 3 and 4.

Readers are divided in two sets. The ones with priority lower than the writer read the value produced by the latest writer instance activated before their activation (the communication link is *direct feedthrough*). Readers with priority higher than the writer read the value produced by the previous writer instance (the communication link has a *unit delay*). The two corresponding buffer entries are indicated by the LATEST and PREVIOUS variables. Two separate arrays, READINGLP[] and READINGHP[], contain one entry for each low and high-priority readers respectively. The writer updates all zero-valued elements of READINGHP[] and READINGLP[] with the value of PREVIOUS and LATEST respectively (lines 24 and 25 in Algorithm 3). When the reader executes on a different core than the writer, additional mechanisms need to be used to ensure the reader starts execution after the data is written. The execution order can be enforced by an activation signal sent to the reader (an inter-core interrupt signal), or by synchronized activations of the writer and reader. The buffer bounds for the SR flow-preserving wait-free methods are computed in a similar way to their non flow-preserving counterparts [12].

IV. FPGA Implementation of MSRP

MSRP consists of two distinct synchronization mechanisms for local and global resources respectively. Locally, we implement the SRP in software since its time- and memory-overheads are limited. For global synchronization, we implement a FIFO queue to serve spinning tasks from different cores on a "First-Come-First-Serve" basis.

The access to the MSRP lock is memory-mapped on the local buses of the cores. A single address is required for each resource since no data is encoded in the address. A read from a core attempts to lock it and returns a non-zero value if successful. To unlock the resource, the core must write 1 back to the same address. This interface is compatible with many processor and bus architectures.

Central to the MSRP FPGA implementation for global resources is a FIFO queue which keeps track of the cores that are requesting access to the resource. At any point in time, only one task can wait on each core. At the head of the queue is the core which currently holds the resource. The hardware implementation is straightforward for an arbitrary number of cores. To simplify write accesses to the FIFO, only one core is considered at a time in a round-robin fashion. As such, it is possible in the worst case for a response to take as many cycles as the number of cores (NC). The worst-case response time is a function of the number of cores.

$$t_{msrp} = \text{NC} + t_{spin} + t_{bus\_l} \tag{1}$$

Algorithm 5: FPGA Implementation of Wait Free Method - Writer

```
   Data: BUFFER [1,...,NB]; NB: Num of buffers
   Data: READING [1,...,n]; // n: Num of readers
   Data: LATEST, CURRENT
1  GetBuf();
2  begin
3      // Writer buffer request arrives from any one bus;
4      LATEST = CURRENT // Commit current write buffer;
5      i = 0 ;
6      buffer_found = false ;
7      while !buffer_found do
8          // Wait one clock cycle ;
9          for j = 0 → 2^MW − 1 do
10             buff = i × 2^MW + j ;
11             if buff ∉ {READING[0 :NR−1],LATEST} then
12                 buffer_found = true ;
13                 CURRENT = buff ;
14                 return buff ;
15             end
16         end
17         i = i + 1 ;
18     end
19 end
```

Algorithm 6: FPGA Implementation of Wait Free Method - Readers

```
   Data: id; // Each reader has its unique id
   Data: READING [1,...,n]; // n: Num of readers
   Data: LATEST
1  Reader_activation();
2  begin
3      // Reader buffer request arrives from any one bus ;
4      buff = LATEST ;
5      READING[id] = buff ;
6      return buff ;
7  end
```

Algorithm 7: FPGA Implementation of Wait Free Method for SR Flow Preservation - Writer

```
   Data: BUFFER [1,...,NB]; NB: Num of buffers
   Data: READING [1,...,n]; // n: Num of readers
   Data: PREVIOUS, LATEST
1  GetBuf();
2  begin
3      // Writer buffer request arrives from any one bus;
4      PREVIOUS = LATEST ;
5      i = 0 ;
6      buffer_found = false ;
7      while !buffer_found do
8          // Wait one clock cycle ;
9          for j = 0 → 2^MW − 1 do
10             buff = i × 2^MW + j ;
11             if buff ∉ {READING[0 :NR−1],PREVIOUS} then
12                 buffer_found = true ;
13                 LATEST = buff ;
14                 return buff ;
15             end
16         end
17         i = i + 1 ;
18     end
19 end
```

Algorithm 8: FPGA Implementation of Wait Free Method for SR Flow Preservation - Readers

```
   Data: id; // Each reader has its unique id
   Data: IS_HP; // Is reader higher priority than writer
   Data: READING [1,...,n]; // n: Num of readers
   Data: PREVIOUS, LATEST
1  Reader_activation();
2  begin
3      // Reader buffer request arrives from any one bus ;
4      if IS_HP then buff = PREVIOUS ;
5      else buff = LATEST ;
6      READING[id] = buff ;
7      return buff ;
8  end
```

where t_{spin} is the spin-lock period, and $t_{buslocal}$ is the time cost of a local bus access.

V. FPGA Implementation of Wait-free Protocol

In our FPGA implementation, a significant effort has been dedicated to find an efficient hardware implementation of the wait-free communication algorithms in [13] (Algorithms 1–4). The resulting algorithms are detailed for readers and writers in Algorithms 5–8 respectively. Key state variables, namely LATEST, PREVIOUS, and READING[], preserve their meaning, as defined in [13].

A critical element of the software wait-free methods implementation is the atomic CAS operation. Since tasks can be preempted for arbitrary amounts of time, there is a risk that a reader may be blocked before it assigns to its READING slot, thus resulting in *GetBuf()* selecting the buffer which the reader is about to read. Ideally, lines 5–7 in Algorithm 2 could be rewritten as `READING[id] = LATEST`. In a software implementation, since both READING and LATEST are stored in shared memory, this cannot be done in a single memory operation. In Algorithms 1–4, a CAS operation is used to ensure that the appropriate buffer index is used for the read operation. In our custom hardware block, we no longer require the CAS operation because we can perform the assignment `READING[id] = LATEST` in a single clock cycle.

A. WF Hardware Design

Our proposed hardware design targets at the Xilinx MicroBlaze soft processors on a Virtex 6 FPGA, but the design can easily be adapted to other FPGA vendors, CPUs, and bus architectures. The WF device acts as a memory-mapped bus slave. Operations are performed through reads at addresses that provide the desired parameters. The 32-bit address format for wait-free methods (with or without flow preservation) is shown in Figure 2. For the flow preservation case (WF-FP), the READINGHP[] and READINGLP[] arrays are merged, and the priority of the reader is simply indicated by the HP/LP field in the address. Thus, the address format contains the ID and priority of the reader, as well as the ID of the buffer being accessed. It also indicates the requested operation (read or write).

We used the MicroBlaze softcore processors provided in Xilinx ISE 14.3 to synthesize a multicore system. Figure 3 illustrates how the wait-free block communicates with other

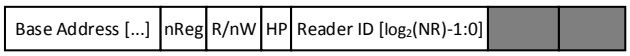

Fig. 2. Addressing Convention Used in Wait-free Implementation

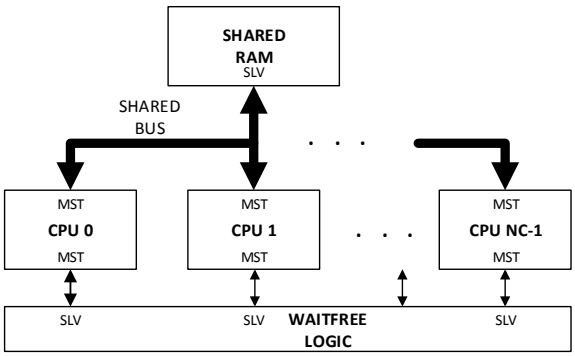

Fig. 3. Block Diagram of the proposed Wait-free Implementation

peripherals on the system. The wait-free block is designed to have an arbitrary number of slave bus interfaces to make the design easily scalable and is connected directly to the MicroBlaze processors. We used a dedicated bus rather than a shared one, to avoid the non-determinism in time when accessing a shared bus.

The device is intended to connect to an arbitrary number of buses as a slave (not capable of initiating transfers), providing a communication bridge among the cores. The single-master buses (referred to as *local buses*) require no arbitration and have minimum overhead. As in the simplified overview of the connections (Figure 3), such a bus structure is also more tolerant to a single master holding a lock for multiple cycles. The wait-free block provides the writer and readers with their index to access a buffer in shared memory. The communication data may be stored in a single-port memory on a shared bus as in Figure 3 or in a multi-port memory on multiple local buses.

1) Readers: A reader activation request, as shown in Algorithm 8, is the simpler of the two operations and is performed in a single cycle. An overview of the hardware design for the reader activation operation is shown in Figure 4. The returned value is either the value of LATEST or PREVIOUS, depending on the priority of the reader. To reduce resource usage, requests are handled in a round-robin fashion and each bus gets a single cycle to perform its operation with a bounded worst case response time equal of NC bus cycles.

2) Writers: Computing the index to be assigned to the writers is a more complex task, as outlined in Algorithm 7. The algorithm has $O(\mathrm{NR}^2)$ complexity, where NR is the number of readers, to search for an available buffer. This is undesirable in terms of time and combinational logic resources (LUTs). Instead of a single-cycle implementation with NR$\times$NB comparators and large combinatorial logic, the operation is performed as cycles on $2^{\mathrm{MW}}\times$NR com-

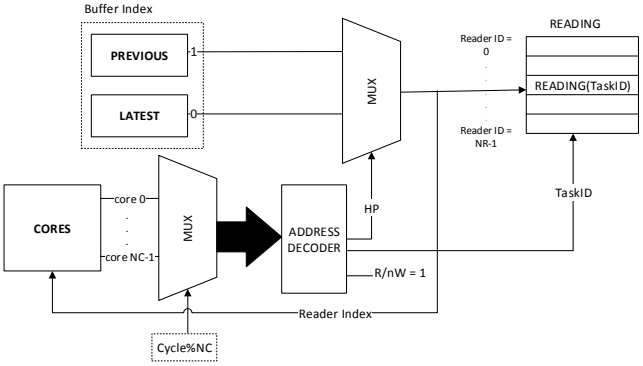

Fig. 4. Hardware design for the implementation of the reader activation operation

parators, where MW (Match Width) is the number of bits (buffer positions) examined at each cycle. This results in an upper bound of

$$t_{writer} = t_{bus\_l} + 1 + 2^{\log_2(\mathrm{NB})-\mathrm{MW}} \quad (2)$$

bus cycles at the start of each writer activation to compute the assigned buffer index. In the majority of our experiments, MW was set to 2. An overview of the hardware design for the assignment of the writer index is shown in Figure 5. In the proposed design, the bus is locked during the writer index computation. This is not strictly necessary. It is possible to free the bus and let the writer poll until the operation is completed. The upper bound on the response time would be higher by one additional local bus access.

VI. Results and Discussion

We synthesized the WF-FP and MSRP blocks for a quad-core MicroBlaze design on a Xilinx Virtex 6 ML605 XC6VLX240T-1FFG1156 Evaluation Board. We evaluate WF-FP for 8, 16, 32, 64, 128, and 256 buffer cases and analyze the timing and resource usage when different levels of parallelism are desired (by increasing MW).

A. Timing Analysis

The timing behavior depends heavily on both the architecture and tools being used. In addition, the performance is subject to the clock speed and throughput requirements imposed by the designer. As such we measure the timing performance as the capability of the design to meet clock rate constraints rather than exhaustively testing every possible configuration. We used the SmartXplorer tool to synthesize and route 5 different strategies, and selected the one with the best performance. Table I shows the worst case clock speed for each of the WF-FP configurations. As shown in the table, increased parallelism (increasing MW) negatively affects the maximum clock rate. However, such a high level of parallelism is not required if the FPGA is used for soft cores or other structures, but may be desirable if the device is only used for the implementation of wait-free buffers and schedulability is improved by having the indexes available in a small number of cycles.

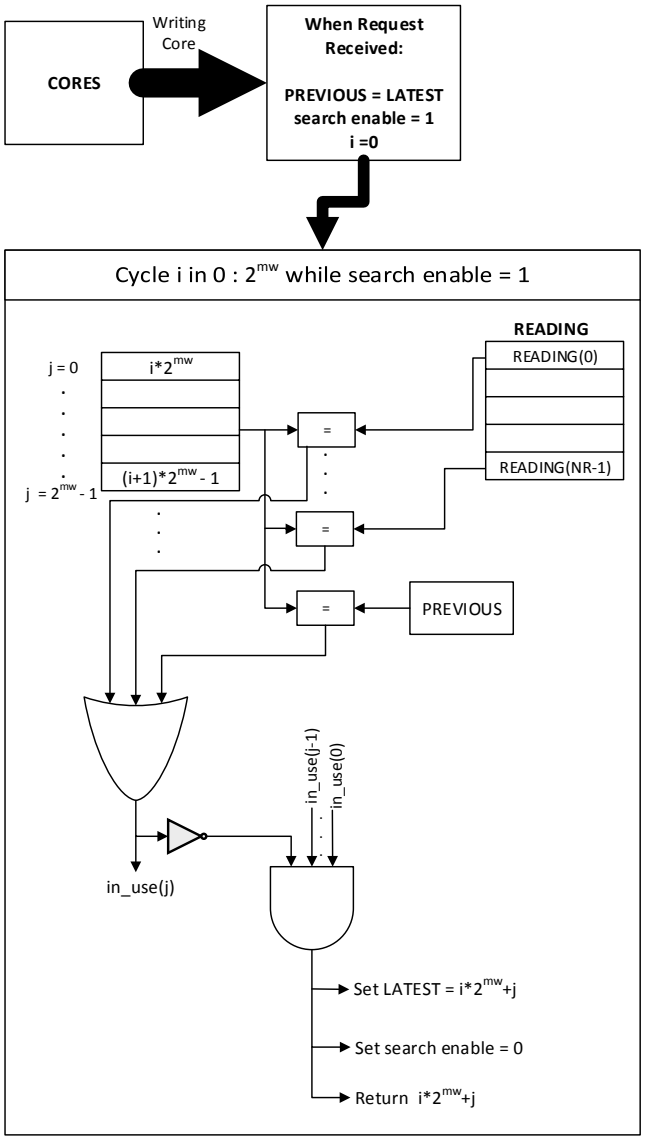

Fig. 5. Hardware design for the assignment of the writer index

Due to its simplicity, the MSRP implementation can operate at speeds approaching that of the interconnect fabric of the FPGA.

a) Hardware Optimization: There are many other possible designs to realize the algorithm that searches for an available index, with trade-offs between the maximum clock rate, resource usage, and t_{writer}. The design in this paper was chosen to balance performance and area at around 200MHz, which is a reasonable target for many complex soft-SoC applications. Integration with significantly faster buses would require a different approach.

A possible example of an alternate implementation of the writer index assignment algorithm uses reference counting to reduce the complexity of the operation to $O(\mathrm{NR})$ at the expense of additional complexity on the reader side. An additional set of buffers is maintained, REFCOUNTS[0:NB-1], where NB is the number of buffers, which tracks the number of readers using each buffer. Whenever a reader is assigned a buffer, the reference count for that buffers is incremented and the reference count of the previous reference counter is decremented. However, in our FPGA implementation, the speed benefits ($\sim 10\%$) are limited in comparison to the resource usage ($\sim 2\times$ increase in both registers and LUTs). The number of required registers increases because REFCOUNTS[] is implemented as either a register array or RAM. The number of LUTs increases because of the addition operations and combinatorial logic.

B. Resource Utilization

Figures 6 and 7 illustrate the slice register and slice LUT utilizations for each configuration. Register usage is not heavily impacted by an increasing MW. However, for large designs (where MW and NB are large) it does exhibit significant growth. The LUT utilization, on the other hand, depends on both MW and NB and it can be seen on Figure 7 that increasing NB and MW by the same factor produce about the same change in the number of LUTs. However, this is not a hard conclusion, since the FPGA may place and route designs irrespective of the computational load associated with an increased MW or NB.

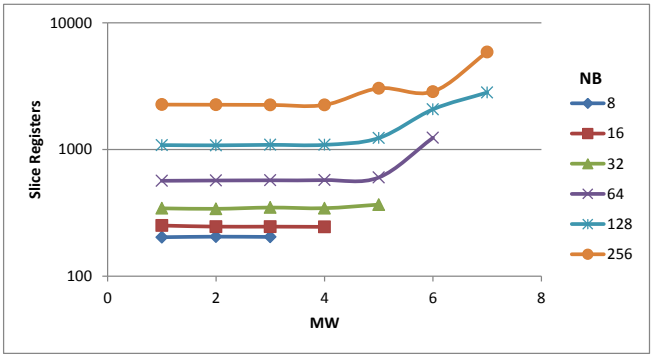

Fig. 6. Slice Register Utilization

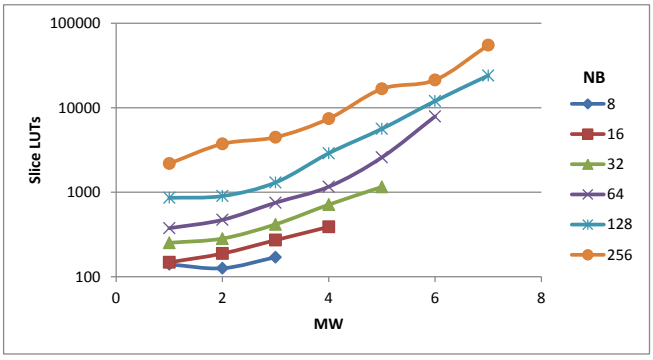

Fig. 7. Slice LUT Utilization

C. Basis for Comparison with Software Implementations

It is difficult to compare the FPGA implementations with their equivalent software implementations due to the variety

TABLE I
CONFIGURATIONS AND THEIR WORST CASE TIMING CONSTRAINTS

Configuration {NB, MW}	Constraint (MHz)
{8,1}	400
{16,2} {8,2}	300
{64,2} {32,2} {32,1} {16,3} {16,1} {8,3}	250
{128,2} {128,1} {64,3} {32,3} {16,4}	200
{256,1} {128,3} {64,4} {32,4}	150
{256,4} {256,3} {256,2} {128,4}	100

of ways to arrange the shared memory, affecting the design and (time/space) cost of the software implementations. We provide a comparison in terms of clock cycles and shared memory access cost in the hope that the results can hold their significance for arbitrary system configurations. As previously stated, the current implementation assumes that the bus is locked by the slave while preparing its response.

1) Writer: In a software implementation of the wait-free methods in Algorithm 1, the writer must make at least NR+1 accesses to shared memory for GetBuf() and NR CAS operations at activation, in addition to local memory operations. It requires a few more in the case of WF-FP (Algorithm 3). Any overhead for shared memory accesses (arbitration or signaling overhead) will linearly scale the time required for the operation. By connecting to a local bus with little or no arbitration overhead and holding that bus for the duration of the operation, we incur the cost of the bus access only once. The polling WF-FP implementation described in Section V-A2 incurs the additional cost of two bus accesses in its worst-case response time.

2) Reader: A reader requesting a buffer in either hardware or software performs a relatively simple operation as listed in Algorithms 2 and 4. In software, the reader writes '0' to its READING buffer, reads LATEST, and then attempts to write that data back to its READING buffer using a CAS operation. It then reads the result of the CAS operation. This is a total of 4 bus operations, one of which is a CAS, incurring 4 units of shared bus access overhead or more depending on the CAS implementation. Our hardware implementation does this in a single local access which holds the bus for up to NC (number of cores) cycles before returning, incurring the overhead of only that single bus access. The worst case time of this is that of a local bus access plus NC clock cycles.

Unlike the writer, which only has one active instance, the reader operation is not easily re-designed to avoid holding the bus. It can be implemented for a single-cycle return if the bus structure does not allow for multi-cycle operations. Such an implementation does not affect the speed of the circuit considerably due to a discrepancy between the worst paths in the reader and writer sub-circuits.

VII. CONCLUSIONS AND FUTURE WORK

We describe the issues in the FPGA implementation of shared resource protection mechanisms. The requirements for the consistency of communication and state variables and the possible additional requirements of flow preservation can be satisfied using several methods. These methods offer tradeoffs between time overheads and additional memory. We implement them on an FPGA, to measure their time and memory overheads.

REFERENCES

[1] MathWorks. *The MathWorks Simulink and StateFlow User's Manuals*. web page: http://www.mathworks.com.

[2] T. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. 11th IEEE Real-Time Systems Symposium*, 1990.

[3] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[4] J. Chen and A. Burns. A fully asynchronous reader/write mechanism for multiprocessor real-time systems. Technical Report YCS 288, Department of Computer Science, University of York, May 1997.

[5] J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Proc. International Conference on Real-Time Computing Systems and Applications*, 1999.

[6] M Di Natale Optimizing the multitask implementation of multirate Simulink models In *Real-Time and Embedded Technology and Applications Symposium*, 2006

[7] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. 22nd IEEE Real-Time Systems Symposium*, 2001.

[8] M. Herlihy. A methodology for implementing highly concurrent structures. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[9] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proc. 30th IEEE Real-Time Systems Symposium*, 2009.

[10] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proc. International Conference on Distributed Computing Systems*, 1990.

[11] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[12] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli. Improving the size of communication buffers in synchronous models with time constraints. *IEEE Transactions on Industrial Informatics*, 5(3):229–240, Aug. 2009.

[13] H. Zeng and M. Di Natale. Mechanisms for Guaranteeing Data Consistency and Flow Preservation in AUTOSAR Software on Multi-Core Platforms. In *Proc. 6th IEEE International Symposium on Industrial Embedded Systems*, 2011.