Experimental Evaluation and Selection of Data Consistency Mechanisms for Hard Real-Time Applications on Multicore Platforms

Gang Han, Haibo Zeng, *Member, IEEE*, Marco Di Natale, *Senior Member, IEEE*, Xue Liu, *Member, IEEE*, and Wenhua Dou

***Abstract*—Multicore platforms are increasingly used in real-time embedded applications. In control systems, including automotive, avionics, and automation, resources shared by tasks on different cores need to be protected by mechanisms that guarantee access in a mutually exclusive way with bounded worst case blocking time. The evaluation of the tradeoffs among the possible protocols for mutual exclusion requires an estimate of their implementation overheads. In this paper, we summarize the possible protection mechanisms and provide code implementations in real-time operating systems executing on a multicore platform. We discuss the tradeoffs among the different mechanisms based on experimental evaluation of their memory and timing overheads as well as their impact on system schedulability. We propose a heuristic algorithm to select the optimal combination of mechanisms for shared resources in systems with time constraints to minimize their memory requirements. The effectiveness of the optimization procedure is demonstrated by synthetic systems as well as industrial case studies.**

***Index Terms*—Data consistency, flow preservation, hard real-time, multicore, multiprocessor stack resource policy (MSRP), multiprocessor priority ceiling protocol (MPCP), wait-free, optimization.**

I. Introduction

MULTICORE architectures have become commonplace in general computing and multimedia applications and are rapidly advancing in other typical embedded computing systems, including automotive and controls. Partitioning the computing tasks over multiple (on-chip) cores presents several advantages with respect to power consumption, reliability, and scalability, but it often requires significant changes to the application tasks to leverage the availability of parallel processing. In real-time embedded (control) systems, especially those following a model-based design, the migration to multicore architectures brings additional challenges. Not only do the worst case timing properties of inter-core communication and synchronization primitives need to be accurately estimated, but communication primitives that can provably preserve the properties of the functional models (*flow preservation*) also need to be developed and evaluated with respect to their time and memory performance.

In multicore architectures, real-time scheduling techniques are classified into *partitioned*, *global*, or *hybrid* [15]. Under *partitioned scheduling*, tasks are statically assigned to processors, and the tasks within each processor are scheduled by a local scheduler. Under *global scheduling*, all tasks are scheduled by a single scheduler. Task management can be done using a single queue or more complex queue systems [21]. Tasks are dynamically allocated to cores (inter-processor migration is allowed), and job migration results in significant overheads [22]. On the other hand, fully partitioned approaches may result in under-utilization, where no single processor has sufficient spare processing time to schedule further tasks even if in total a large amount of capacity is unused [5]. *Hybrid scheduling* combines the strengths of both partitioned and global scheduling. It can be further categorized into *semi-partitioned scheduling* and *clustering*. In *semi-partitioned scheduling*, most tasks are allocated to specific processors to reduce the number of migrations, while other tasks are allowed to migrate to balance processors utilization. *Clustering* groups a smaller number of faster processors into a cluster, and each cluster is scheduled with a different global scheduler. Interested readers may refer to [5] for a survey on the topic of hard real-time scheduling for multicore systems.

Partitioned scheduling is adopted and supported by domain-specific standards like AUTOSAR [1] and by commercial real-time operating systems (e.g., VxWorks, LynxOS, and ThreadX). In addition, most automotive controls are designed with *static priority-based scheduling* of tasks. The OSEK [6] and AUTOSAR [1] operating system standards support this model. Other scheduling policies, including Earliest Deadline First (EDF) and its multiprocessor versions, while quite popular in the research community, are not supported by the industry. In this work, we assume **partitioned scheduling with static priority**, mostly for its practical relevance. Moreover, it should be noted that the utilization bounds available from the real-time theory for global scheduling policies are still quite pessimistic

Manuscript received September 03, 2012; revised March 24, 2013, June 20, 2013; accepted October 31, 2013. Date of publication November 12, 2013; date of current version May 02, 2014. Paper no. TII-12-0665.

G. Han and W. Dou are with the School of Computer Science, National University of Defense Technology, Changsha 410073, China. E-mail: ganghan@nudt.edu.cn.

H. Zeng is with the Department of Electrical and Computer Engineering, McGill University, Montreal QC, Canada H3A 0E9 (e-mail: haibo.zeng@mcgill.ca).

M. Di Natale is with Scuola Superiore Sant'Anna, Pisa 56124, Italy (e-mail: marco@sssup.it).

X. Liu is with the School of Computer Science, McGill University, Montreal QC, Canada H3A 0E9 (e-mail: xueliu@mcgill.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2013.2290585

(compared with their counterparts for partitioned scheduling, which offer necessary-and-sufficient tests) [15].

In this work, we are interested in applications with hard real-time constraints, where the worst case response time of the computation task must be predictable based on the worst case execution time of its code. The response time of a task clearly depends not only on the CPU scheduling policy and task priority, but also on stalls when accessing shared (locked) resources. In real-time control systems, blocking times occur while accessing shared resources (i.e., a mailbox that is used for data communication) which are already owned by some other task. Hence, the mechanisms for ensuring the consistency of shared data need to be carefully designed, so that they guarantee an upper bound on the blocking time. In multicore architectures, intra-core as well as inter-core shared resources need to be suitably protected (possibly using different type of access policies).

Moreover, in model-based design development flows, which are very popular today in the automotive, avionics, and control domains [33], [34], [36], the signal flow of the implementation (the sequence data values exchanged over time among two system functions) should match the one defined in the functional Synchronous Reactive model [33], [35]. In the following, we summarize the mechanisms for data consistency in multicore systems and analyze the subset of mechanisms that can guarantee flow preservation. A classification of protocols with a detailed discussion can be found in [37]. Protocols for predictable access to nested global critical sections are discussed in [41].

A. Data Consistency

Several protocols have been proposed in the past for supporting communication over shared memory or, in general, protecting shared resources. The available mechanisms belong to three main categories, given here.

- *Lock-based*: when a task wants to access the shared resource (such as a shared datum) while another task holds the lock, it blocks. When the lock is released, the task is restored in the ready state and can access the resource. In multicore architectures with global locks, two options are possible. The blocked task is suspended and transferred to a (global) waiting list to allow another task to execute, or the task may spin on the lock (busy-waiting) [26], [17], [12].
- *Lock-free*: each reader accesses the communication data without blocking. At the end of the operation, it performs a check. If the reader realizes there was a possible concurrent operation by the writer, it repeats the operation. The number of retries can be upper bounded [18], [7].
- *Wait-free*: when the global shared resource is a communication buffer (the case of interest for our study), another possibility is wait-free methods. The writer and readers are protected against concurrent access by replicating the communication buffer (the memory used to hold the communication data) and by leveraging information on the time instant and order (such as priority and scheduling) of the access to the buffer [14], [19].

Among lock-based mechanisms, the multiprocessor priority ceiling protocol (MPCP) [26] and the multiprocessor stack resource policy (MSRP) [17] are the most popular for guaranteeing a predictable worst case blocking time. Their behavior can be radically different depending on the execution time of the global critical sections. The flexible multiprocessor locking protocol (FMLP) [12] has been proposed to combine the strengths of the two by managing short resource requests using a busy-wait mechanism (as in MSRP) and long resource requests using a suspension approach (as in MPCP). Parallel-PCP is proposed in [16] for tasks scheduled using a global fixed-priority preemptive algorithm. The multiprocessor hierarchical synchronization protocol (MHSP) [25] uses the hierarchical scheduling framework to handle both partitioned and global scheduling approaches. The main subject of the papers is the clustering of tasks into components so that all communications occur between tasks allocated onto the same core. Each component is scheduled with a lower level scheduler, while the higher level system scheduler is either the global or the core scheduler. In this case, there are simply no data resources shared among cores, thus there is no need for inter-core data synchronization mechanisms. However, as acknowledged by the authors, in several cases, their method is simply not applicable. The multiprocessor synchronization protocol for real-time open systems (MSOS) [25] is proposed as a suspension-based synchronization protocol for independently developed systems on multicore platforms. Each core supports a set of applications developed with possibly different techniques (and scheduling policies). Since our focus is on static priority systems with partitioned scheduling, we only consider MPCP and MSRP in our implementation and comparison, the other mechanisms are not applicable or similar in behavior (FMLP).

In wait-free protocols, the only shared resource is the buffer index, which can be updated atomically or with a very short critical section. However, this comes at the price of replicating the resource, which may be costly for large communication mailboxes or altogether impossible for physical shared resources (such as I/O or dedicated hardware peripherals).

The performance tradeoffs between spin-based and suspension-based synchronization protocols have been studied in several papers. Brandenburg *et al.* [13] show that, even under an assumption of zero preemption costs, spin-based protocols impose a smaller scheduling penalty than suspension-based. In response, Lakshmanan *et al.* [22] developed new schedulability analysis. With the new analysis, the authors found that [22] "the suspension-based protocols in fact behave better than spin under low preemption costs (less than 160 μs per preemption) and longer critical sections (15 μs)" than those studied in [13].

Related topics also include algorithms for the synthesis of tasks from functional models [38], [39] and the allocation of tasks to cores in a multicore platform [40]. A field-programmable gate array (FPGA) implementation of wait-free semantics-preserving mechanisms like those discussed in this paper is presented in [42].

There is also a large body of work on synchronization protocol for many-core embedded architectures (Cell, STM P2012/STHORM, NUMA, and GPUs). For example, the authors of [23] perform a scalability analysis targeting synchronization mechanisms on multicluster embedded NUMA architectures, where each cluster is interconnected through a scalable communication medium [typically a network-on-chip (NoC)]. The work in [24] explores and compares synchronization latencies for various network sizes, topologies, and lock position in the NoC. The work in [29] presents hardware implementations of lock-based synchronization mechanisms. While interesting, these works present (scalable) communication mechanisms characterized by their average execution times. Consequently, none of these approaches are suitable for worst case formal analysis as required by hard real-time systems. In addition, *we consider multicore architectures that are typical of control applications (automotive, avionics, automation) and real-time systems, with memory shared among cores (typically 2 to 16)*. Scalability to many-core systems (GPUs, many-core NUMA, NoC, or other massively parallel systems) is not the focus of our work. Finally, our purpose is not the invention of new resource-sharing protocols, but rather the synthesis of an optimal selection of existing ones, based on worst case timing and memory performance.

B. Flow Preservation

Data consistency in a shared communication buffer is a typical requirement for hand-written code, but is not sufficient for software developed using a *Model-based design flow*, very popular in automotive, avionics, and controls domains, because of the possibility to verify the functionality by simulation and formal methods. Typically the functional model is defined according to the semantics of Synchronous Reactive (SR) models [11], like those created in MathWorks Simulink [4]. The functional model is a network of communicating blocks. Each block reads a set of (discrete-time) input signals and produces a set of output signals. Any block for which the output is directly dependent on the input (i.e., with *direct feedthrough* input) cannot execute until the blocks driving its input have executed. The set of topological dependencies implied by the direct feedthrough relation defines a partial order of execution among blocks.

When Simulink simulates a model, it orders all blocks based on their topological dependencies and chooses one total execution order that is compatible with the partial order. Then, the virtual time is initialized at zero. The simulator engine scans the blocks and executes the ones for which the virtual time is an integer multiple of their period. Executing a block means computing the output function, followed by the state update function. When the execution of all of 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 (the greatest common divisor of block periods) and resumes scanning the block list.

The code generation framework must produce an implementation with the same behavior (preserving the semantics) of the simulation. 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). Fig. 1 shows the difference between the model behavior (top of the figure) and a possible multitask implementation. The kth instance of block b_j should use as input the output of the mth instance of $b_i (i_j(k) = o_i(m))$. However, if blocks are executed by tasks as in the bottom part of the figure, the execution of $b_j(k)$ may be delayed by interferences from higher priority tasks (striped box), resulting in $i_j(k) = o_i(m+1)$. Even worse, if the read/write to the communication data is not atomic, preemption may compromise the data integrity. To solve these problems, wait-free mechanisms can be adapted to provide flow preservation [33] (detailed in Section II-C).

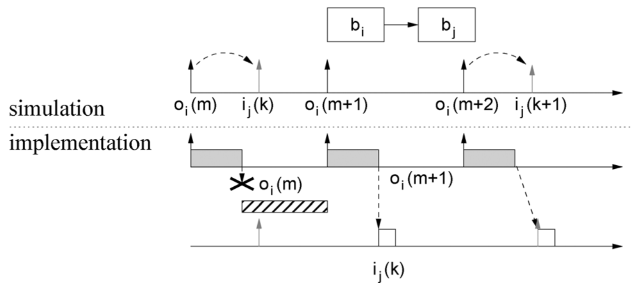


Fig. 1. Issues with the preservation of communication flows (the periods of the blocks are $T_i = 2T_j$).

C. Our Contributions

In this paper, we consider the data synchronization protocols for systems with static priority and partitioned scheduling. To the best of our knowledge, previous comparisons like [13] focus on dynamic priority scheduled systems. Other studies (e.g., [17], [22]) only compare lock-based mechanisms and do not attempt an evaluation of alternative implementations using wait-free methods. The protocol behaviors are strongly dependent on overheads and especially on the length of critical sections, but there is limited experimental data on what is the expected length of a critical section when accessing a shared memory resource in an actual embedded platform. In order to perform an experimental comparison of the performance of different synchronization protocols, *we implemented MPCP, MSRP, and wait-free methods* on Erika [2], an open-source RTOS which has been ported to the MPC5668G dual-core platform [5] for automotive applications. For demonstrating the generality of the proposed mechanisms, we also implemented them in another open-source RTOS, Trampoline [10], on the same dual-core platform. In addition, our paper provides two additional contributions. First, *we propose an algorithm that leverages the complementary characteristics* of the lock-based and wait-free protection mechanisms to select a combination of protection mechanisms that satisfies the schedulability constraints and is memory-efficient. Second, *we discuss the implementation options when the system requires the preservation of communication flows derived from a functional model, and we compare the possible solutions*.

The remainder of this paper is organized as follows. In Section II, we discuss the mechanisms for data consistency in multicore systems and analyze the subset that can guarantee flow preservation. In Section III, we describe their implementation on the Erika and Trampoline RTOSs for the MPC5668G multicore platform. In Section IV, we evaluate and compare the schedulability and memory performance of the mechanisms on both RTOSs. In Section V, we use industrial case studies and randomly generated task sets to discuss the tradeoffs of these mechanisms and define an optimization procedure to minimize the memory usage while guaranteeing schedulability. Finally, a conclusion and future work are discussed in Section VI.

II. Mechanisms for Data Synchronization

We evaluate and compare wait-free methods and semaphore locks. Lock-free mechanisms have been introduced for the sake of providing a classification of methods for data consistency. However, their use in hard real-time systems and model-based design is inappropriate, due to the large bound on the worst case blocking time and high time penalty (because of the need to repeat the operation in case of a concurrent access). As a matter of fact, there has been very little work on the use of lock-free methods for hard real-time systems since the late 1990s.

In the following, we summarize the lock-based and wait-free mechanisms together with their associated timing analysis. We assume that each task τ_i *is activated by a periodic or sporadic event stream with period or minimum interarrival* T_i. The execution of task τ_i is defined as a set of alternating critical sections and sections in which the task executes without using a (global or local) shared resource, defined as *normal execution segments*. The worst case execution time (WCET) is defined by a tuple $\{C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, \ldots, C'_{i,s(i)-1}, C_{i,s(i)}\}$, where $s(i)$ is the number of normal execution segments, $s(i)-1$ is the number of critical sections, $C_{i,j}(C'_{i,j})$ is the WCET of the jth normal execution segment (critical section) of τ_i, and π_i denotes the nominal priority of τ_i (*the higher the number, the lower the priority*, thus $\pi_i < \pi_j$ means τ_i has a higher priority than τ_j), and E_i its core. The global shared resource associated with the jth critical section of τ_i is $\mathcal{S}_{i,j}$. The WCET C_i of τ_i is

$$C_i = \sum_{1\le j\le s(i)} C_{i,j} + \sum_{1\le j< s(i)-1} C'_{i,j}. \tag{1}$$

A. Lock-Based Mechanisms

The MPCP [26] is the first multiprocessor extension of the priority ceiling protocol (PCP) [27]. In MPCP, tasks use their nominal priorities for normal execution but inherit the ceiling priority of the shared resource whenever they execute a critical section on it. The ceiling priority of a local resource is defined as the highest priority of any task that can possibly use it. The priority ceiling of a global resource must be higher than any task priority in the system. Thus, a base priority higher than any task is applied to all global ceilings. When tasks try to access a locked global resource, they are suspended and added to a priority queue. The suspension of a task blocked on a global resource allows other tasks (possibly with a lower priority) to be executed and possibly lock other resources. The worst case remote blocking time of a task is a function of the duration of critical sections for other tasks, and does not depend on the normal execution segments.

The MSRP [17] is a multiprocessor synchronization protocol, extended from the stack resource policy (SRP) [8]. For local resources, the protocol is the same as SRP. Tasks are allowed to access local resources through nested critical sections, but global critical sections cannot be nested. A task that fails to lock a global resource keeps *spinning* (instead of suspending as in MPCP), thus keeping its processor busy. To minimize the spin lock time (wasted CPU time), tasks cannot be preempted when executing a global critical section. MSRP uses a first-in-first-out queue (as opposed to the priority-based queue in MPCP) for tasks that fail to lock global resources.

The FMLP [12] is a flexible approach which combines the strengths of MPCP and MSRP. It manages short resource requests by a busy-wait mechanism (as in MSRP) and long requests using suspension (as in MPCP). The threshold to determine whether a resource access is long or short is specified by the user. Since FMLP allows resource requests be nested, deadlock is prevented by grouping resources and allowing only one task to access resources in any given group at any time. Each group contains either only short (protected by a nonpreemptive queue lock) or only long (protected by a semaphore) resources.

The mixture of spin lock and suspension makes the analysis of FMLP complex. For simplicity, in the following, we only summarize the timing analysis of MSRP and MPCP. *With respect to memory*, MPCP is a suspension-based mechanism and does not allow to share the stack space. With MSRP, the execution of all tasks allocated to the same core can be perfectly nested (once a task starts execution, it does not block; it can only be preempted by higher priority tasks which finish before it resumes). The tasks can share the same stack.

1) Timing Analysis of MPCP [22]: In MPCP, a global shared resource $\mathcal{S}_{i,j}$ is associated with a **remote priority ceiling** $\Pi_{i,j}$, which is the highest priority among all the tasks that can access job $\mathcal{S}_{i,j}$, offsetted by a base priority level greater than that of any normally executing task in the system. The normal execution segment of a task can be blocked by the critical section of each lower priority task on the same core. For each of the $s(i)$ normal execution segments, the worst case local blocking time is the longest critical section used by any lower priority task. Thus, the total local blocking time of τ_i is

$$B_i^l = s(i) \times \sum_{k:\pi_k>\pi_i \wedge E_k=E_i} \max_{1\le m<s(k)} C'_{k,m}. \tag{2}$$

Once it enters its critical section, τ_i can only be interfered by critical sections with a higher remote priority ceiling. Also, since the critical section has a higher priority than the normal execution segment, in each critical section there can be at most one such interference from each task on the same core. Thus, the response time of the jth critical section is bounded by

$$W'_{i,j} = C'_{i,j} + \sum_{k\ne i:E_k=E_i} \max_{1\le m<s(k)\wedge \Pi_{k,m}<\Pi_{i,j}} C'_{k,m}. \tag{3}$$

The remote blocking time $B_{i,j}^r$ for the jth critical section can be computed by the following iterative formula:

$$B_{i,j}^r = \max_{\pi_k>\pi_i\wedge \mathcal{S}_{k,m}=\mathcal{S}_{i,j}} W'_{k,m} + \sum_{\pi_h<\pi_i\wedge \mathcal{S}_{h,n}=\mathcal{S}_{i,j}} \left(\left\lceil \frac{B_{i,j}^r}{T_h}\right\rceil + 1\right) W'_{h,n}.$$

The total remote blocking time is

$$B_i^r = \sum_{1\le j<s(i)} B_{i,j}^r. \tag{4}$$

The worst case response time R_i of τ_i can be calculated as the convergence of the following iterative formula:

$$R_i = C_i + B_i^l + B_i^r + \sum_{\pi_h < \pi_i \wedge E_h = E_i} \left\lceil \frac{R_i + B_h^r}{T_h} \right\rceil C_h \quad (5)$$

2) Timing Analysis of MSRP [17]: The spin block time $L_{i,j}$ that a task τ_i needs to spend for accessing a global resource $\mathcal{S}_{i,j}$ can be bounded by

$$L_{i,j} = \sum_{E \neq E_i} \max_{k: E_k = E, 1 \leq m < s(k)} C'_{k,m}. \quad (6)$$

This time is added to the execution of the jth critical section of τ_i. Thus, its worst case execution time becomes

$$C_i^* = C_i + \sum_{1 \leq j < s(i)} L_{i,j}. \quad (7)$$

MSRP maintains the same basic property of SRP, that is, a task cannot be blocked once it starts executing. The local blocking time B_i^l and remote blocking time B_i^r are

$$B_i^l = \max_{k: \pi_k > \pi_i \wedge E_k = E_i} \left\{ \max_{1 \leq m < s(k)} C'_{k,m} \right\} \quad (8)$$

$$B_i^r = \max_{k: \pi_k > \pi_i \wedge E_k = E_i} \left\{ \max_{1 \leq m < s(k)} \left(C'_{k,m} + L_{k,m} \right) \right\}. \quad (9)$$

The blocking time B_i can be computed as

$$B_i = \max\left(B_i^l, B_i^r\right). \quad (10)$$

The worst case response time R_i of τ_i can be computed as

$$R_i = C_i^* + B_i + \sum_{\pi_h < \pi_i \wedge E_h = E_i} \left\lceil \frac{R_i}{T_h} \right\rceil C_h^*. \quad (11)$$

B. Wait-Free Methods for Data Consistency

Wait-free methods avoid blocking by ensuring that each time a writer needs to update the communication data, it is reserved with an unused buffer. At the same time, readers use dedicated buffers that are guaranteed not to be accessed by the writer. Fig. 2 shows the typical stages performed by the writer and the reader in a wait-free protocol. These stages have been implemented in [14] by means of an atomic compare-and-swap (CAS) operation. The CAS takes three operands

```
Compare-and-Swap(mem, v1, v2)
```

where the value `v2` is written into the memory location `mem` only if the current value of `mem` is equal to `v1`. Otherwise, the value at `mem` is left unchanged.

The algorithm in [14] uses three global sets of data. An array of buffers (BUFFER[]) is sized so that there is always an available buffer for the writer to write new data. An array READING[] keeps track in the ith position what is the buffer index in use by the ith reader (where multiple readers may use the data in the same buffer index). When the reader is in the process of updating this information, a value 0 is stored in this entry. Finally, a variable LATEST keeps track of the latest BUFFER entry that has been updated by the writer. The code for the writer is shown in Algorithm 1. The writer updates the view of what buffers are used by readers, then picks a free buffer using the procedure GetBuf(). Next, it uses the buffer item to write the newly produced data item and updates LATEST. Finally, it uses a CAS operation to ensure the consistency in the updates of the READING[] indexes.

On the reader side, the code is in Algorithm 2. The reader first looks for the latest entry updated by the writer (which uses a CAS operation to ensure the consistency in the update of its READING[] index), stores it in the local variable `ridx`, and then reads its contents. It is possible that more than one readers may access the same buffer.

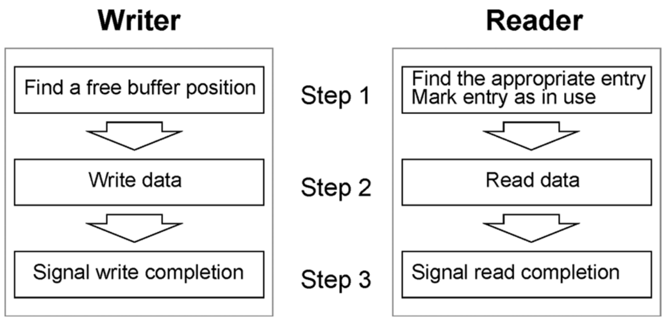


Fig. 2. Stages for writer and readers in a wait-free protocol.

Algorithm 1: Wait-Free Method for Data Consistency - Writer [14]

```
Data: BUFFER [1,...,NB]; // NB: Num of buffers
Data: READING [1,...,n]; // n : Num of readers
Data: LATEST
GetBuf(); // Dynamic Buffering Protocol (DBP) version;
begin
    bool InUse [1,...,NB];
    for i=1 to NB do InUse [i]=false;
    InUse[LATEST]=true;
    for i=1 to n do
        j = READING [i];
        if j !=0 then InUse [j]=true;
    end
    i=1;
    while InUse [i] do ++i;
    return i;
end
GetBuf(); // Temporal Concurrency Control Protocol (TCCP) version;
begin
    return LATEST% NB + 1;
end
Writer();
begin
    integer widx, i;
    widx = GetBuf();
    Write data into BUFFER [widx];
    LATEST = widx;
    for i=1 to n do CAS(READING [i],0,widx);
end
```

Algorithm 2: Wait-Free Method for Data Consistency - Readers [14]

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

This mechanism ensures data consistency with some runtime overhead, but avoids the introduction of blocking times. However, it requires a number of buffer replicas and, thus, additional memory. The buffer size can be defined by the **reader instance method**, which relies on the computation of an *upper bound for the maximum number of buffers that can be used at any given time by reader tasks* [14] or by the **lifetime bound method** based on the computation of an *upper bound on the number of times the writer can produce new values while a given data item is used by at least one reader* [20], [14].

Using the **reader instance method**, the required buffer size is equal to the maximum number n of reader task instances that can be active at any time (the number of reader tasks if task deadlines are not greater than periods), plus two more buffers [14]. If all readers have a priority lower than the writer, then only $n+1$ buffers are needed [28]. The proposed protocol is called **Dynamic Buffering Protocol (DBP)**. This mechanism ensures data consistency with $O(n)$ runtime complexity.

The **lifetime bound method** for buffer sizing has been first introduced in [20] (as part of the non-blocking write protocol) and [14]. The corresponding communication protocol is also referred to as **temporal concurrency control protocol (TCCP)**. Its runtime complexity is $O(1)$, and the number of buffers depends on the lifetime of the data. For a reader τ_i, the lifetime can be upper bounded by $l_i = R_i + O_{wi}$, where R_i is the worst case response time of τ_i, O_{wi} the maximum offset between any consecutive activations of the writer τ_w and τ_i. The number of buffers for writer τ_w is

$$NB_w = \max_{i \in \text{readers(w)}} \left\lceil \frac{l_i}{T_w} \right\rceil .$$

A combination of the two buffer sizing methods can be used to obtain a better bound, as proposed first in [19] and [9] and then improved in [32]. Reader tasks are partitioned into two groups: fast and slow. The buffer bound for the fast readers uses the lifetime bound method, and the bound for the slow ones leverages the reader instance method.

C. Wait-Free Methods for Flow Preservation

Lock-based mechanisms do not guarantee the preservation of SR flows and are not meant to. Wait-free methods that

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

```
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
GetBuf(); // DBP version;
begin
    bool InUse [1,...,NB];
    for i=1 to NB do InUse [i]=false;
    InUse[LATEST]=true;
    for i=1 to $n_{lp}$ do
        j = READINGLP [i];
        if j !=0 then InUse [j]=true;
    end
    for i=1 to $n_{hp}$ do
        j = READINGHP [i];
        if j !=0 then InUse [j]=true;
    end
    i=1;
    while InUse [i] do ++i;
    return i;
end
GetBuf(); // TCCP version;
begin
    return LATEST % NB + 1;
end
Writer_activation();
begin
    integer widx, i;
    widx = GetBuf();
    PREVIOUS = LATEST;
    LATEST = widx;
    for i=1 to $n_{hp}$ do CAS(READINGHP [i], 0, PREVIOUS);
    for i=1 to $n_{lp}$ do CAS(READINGLP [i], 0, LATEST);
end
Writer_runtime();
begin
    Write data into BUFFER [widx];
end
```

provide a correct implementation of SR flows require changes to the ones in Section II-B. The data item used by the reader must be defined based on the writer and reader task activation times. However, both tasks are not guaranteed to start execution right after their activation because of scheduling delays. Therefore, the assignment of 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. Similar to the case of data consistency only, the buffer sizes can be defined by analyzing the relationship between the writer and its reader task instances. With reference to the algorithm presented in the previous sections, the writer 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. The pseudo code is shown in Algorithms 3 and 4.

Readers are divided in two sets. The ones that has 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

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

```
ReaderLP_activation();
begin
    constant id; // Each lower priority reader has its unique id;
    integer ridx;
    READINGLP [id]=0;
    ridx = LATEST;
    CAS(READINGLP [id],0,ridx);
    ridx = READINGLP [id];
end
ReaderHP_activation();
begin
    constant id; // Each higher priority reader has its unique id;
    integer ridx;
    READINGHP [id]=0;
    ridx = PREVIOUS;
    CAS(READINGHP [id],0,ridx);
    ridx = READINGHP [id];
end
Reader_runtime();
begin
    Read data from BUFFER [ridx];
end
```

PREVIOUS variables. Two separate arrays, READINGLP[] and READINGHP[], contain one entry for each low and high-priority readers, respectively, even if they are managed in the same way. The writer updates all zero-valued elements of READINGHP[] and READINGLP[] with the value of PREVIOUS and LATEST respectively (lines 28 and 29 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 nonflow-preserving counterparts.

III. Implementation of Communication Mechanisms

Here, we describe the implementation of the communication mechanisms on the Erika [2] and Trampoline RTOSes [10] for the MPC5668G dual-core platform [5].

A. MPC5668G Architecture

MPC5668G is a heterogeneous dual-core system-on-chip (SoC) 32-bit microcontroller by Freescale. The main core, an *e200z6*, has higher computation power and typically performs the bulk of the application functionality. The smaller core (an *e200z0*), also called a co-processor, performs less complex operations like I/O, redundancy checks, or corrections on the main functional path. Both cores have access to all memories, bus masters, the FlexRay and the eDMA controllers, and other peripherals. A crossbar switch is used to control access and arbitration, and an interrupt controller is provided to forward the interrupt requests to any core or both (see Fig. 3).

The MPC5668G provides two hardware mechanisms for the implementation of mutexes, or the protection of resources globally shared between cores. The first is typical of Power architectures, consisting of a pair of instructions `lwarx` (Load Word and Reserve Indexed) and `stwcx` (Store Word Conditional Indexed). `lwarx` loads a word from a memory location and creates a reservation. Any store operation to the specified location will cancel the reservation. `stwcx` performs a store to the location only if the reservation still exists. Together they can be used to implement atomic *compare-and-swap* operations, which are universal for achieving consensus in multi-master architectures [18].

To further simplify access to any type of shared resources, including peripherals and I/O registers, the MPC5668G provides 16 hardware semaphores. The Erika kernel allocates two of them for remote (inter-core) notifications, including sending events and task activation signals. The remaining 14 semaphores are available for mutex operations on application-level resources, including shared memory buffers. Erika provides the library functions LockHardSemaphore() and UnlockHardSemaphore() for managing the semaphore locks with a spin-based mechanism as in Fig. 4. A core accessing a shared resource first needs to acquire the semaphore lock for that resource. If the lock is obtained, the core has access; otherwise, the resource is in use by the other core, and the requesting task enters a spin lock.

B. Implementation of Wait-Free Methods on Erika

In our implementation, we use the hardware semaphore and the library functions in Erika (Fig. 4), to replace the Compare-and-Swap atomic instruction in Algorithms 1–4 for the consistent update of the buffer indexes.

Wait-free implementations can be of two types. The communication mechanism that returns the latest value written by the writer is simpler as it can be implemented at the application-level, but it only guarantees data consistency. The implementation of wait-free communication with flow preservation for synchronous reactive models, as described in Section II-C, requires support at the kernel level. As all of the tasks in our experiments are triggered by the timer, we implement the operations at activation time (Writer_activation() and Reader_activation() functions in Algorithm 3 and 4) in the timer interrupt service routine.

In addition, flow preservation requires enforcing a partial order of execution between the writer and its readers. Therefore, if the readers and the writer are allocated to different cores, the relative priority order is no longer sufficient to enforce an execution order, and it is necessary for the lower priority readers to block until the writer finishes writing to the buffer. This is achieved using the WaitEvent and SetEvent pair of primitives supplied by the Erika kernel. When a task calls WaitEvent, it suspends. When a task sends an event to a waiting task by calling SetEvent, the waiting task is inserted into the ready queue, or executed immediately. Readers with lower priority than the writer call WaitEvent before entering their critical sections, while the writer invokes the SetEvent function after it finishes writing into the shared buffer.

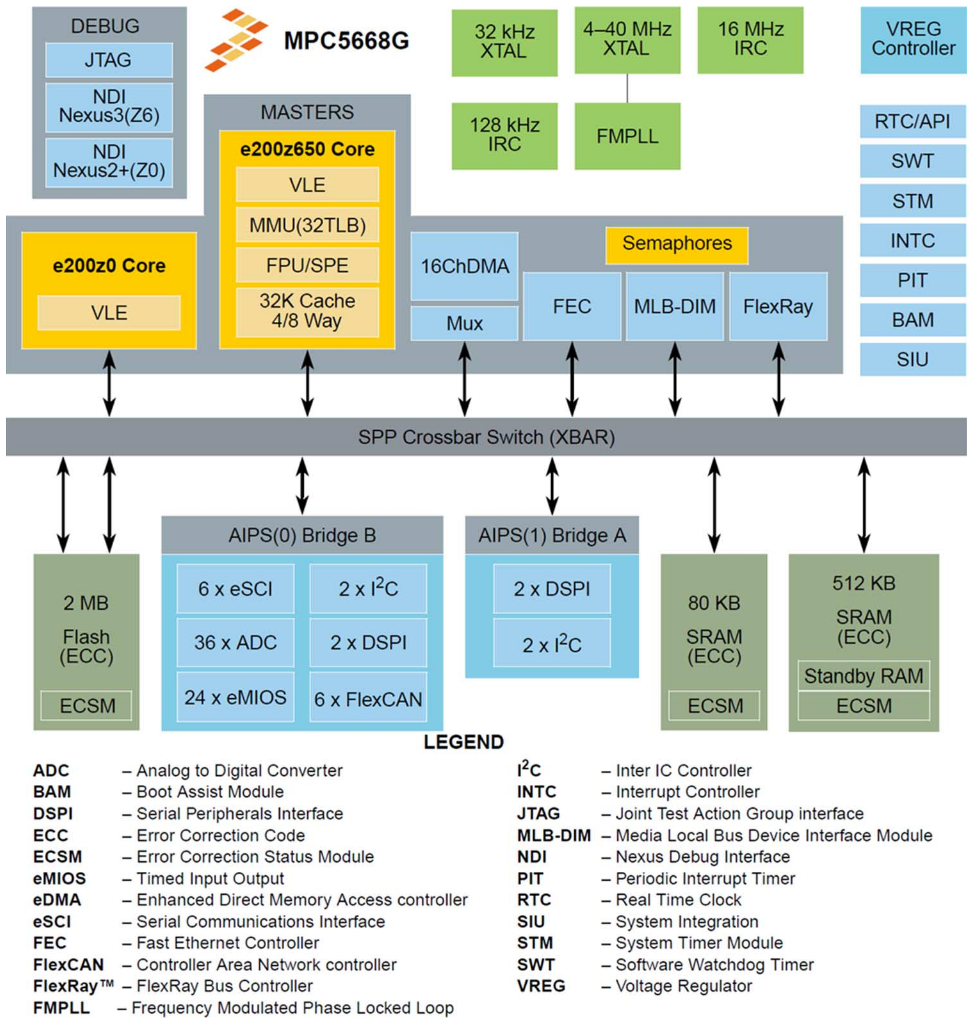

DEBUG
JTAG
NDI Nexus3(Z6)
NDI Nexus2+(Z0)
MPC5668G
32 kHz XTAL
4–40 MHz XTAL
16 MHz IRC
VREG Controller
128 kHz IRC
FMPLL
MASTERS
e200z650 Core
VLE
MMU(32TLB)
FPU/SPE
32K Cache 4/8 Way
e200z0 Core
VLE
16ChDMA
Mux
Semaphores
FEC
MLB-DIM
FlexRay
RTC/API
SWT
STM
INTC
PIT
BAM
SIU
SPP Crossbar Switch (XBAR)
2 MB Flash (ECC)
ECSM
AIPS(0) Bridge B
6 x eSCI
2 x I^{2}C
36 x ADC
2 x DSPI
24 x eMIOS
6 x FlexCAN
AIPS(1) Bridge A
2 x DSPI
2 x I^{2}C
80 KB SRAM (ECC)
ECSM
512 KB SRAM (ECC)
Standby RAM
ECSM
LEGEND
ADC – Analog to Digital Converter
BAM – Boot Assist Module
DSPI – Serial Peripherals Interface
ECC – Error Correction Code
ECSM – Error Correction Status Module
eMIOS – Timed Input Output
eDMA – Enhanced Direct Memory Access controller
eSCI – Serial Communications Interface
FEC – Fast Ethernet Controller
FlexCAN – Controller Area Network controller
FlexRay™ – FlexRay Bus Controller
FMPLL – Frequency Modulated Phase Locked Loop
I^{2}C – Inter IC Controller
INTC – Interrupt Controller
JTAG – Joint Test Action Group interface
MLB-DIM – Media Local Bus Device Interface Module
NDI – Nexus Debug Interface
PIT – Periodic Interrupt Timer
RTC – Real Time Clock
SIU – System Integration
STM – System Timer Module
SWT – Software Watchdog Timer
VREG – Voltage Regulator

Fig. 3. MPC5668G multicore architecture [5].

```
LockHardSemaphore(id) {          // Acquire the spin lock
    locked_val = LOCK;
    do {
             SEMAPHORE.GATE[id].value = locked_val;
    } while (SEMAPHORE.GATE[id].value != locked_val);
}

UnlockHardSemaphore(id) {        // Release the spin lock
    SEMAPHORE.GATE[id].value = UNLOCK;
}
```

Fig. 4. Pseudo code for hardware semaphore operations in the Erika [2].

C. Implementation of MSRP on Erika

The critical sections for lock-based mechanisms (including MPCP and MSRP) begin with the API GetResource() and end with a call to ReleaseResource(), as follows:

```
GetResource(ResId);
DO_CS();//criticalsection
ReleaseResource(ResId);
```

As MPC5668G is a dual-core architecture, when a resource is locked, only one task can be spinning. However, only 14 hardware semaphores are available to manage the access to possibly more shared resources. Therefore, we implemented a set of software semaphores sharing a single hardware semaphore as shown in Algorithm 5. Two methods LockSoftSemaphore() and UnlockSoftSemaphore() are provided to lock and unlock a software semaphore. The array SOFTSEMAPHORE[] maintains the states of all software semaphores, "1" representing the lock by core z6, "2" for core z0, and "0" for the unlocked state. All of the entries of SOFTSEMAPHORE[] are initialized as "0." Our software layer uses the Erika APIs LockHardSemaphore() and UnlockHardSemaphore() to guarantee that the two cores do not access SOFTSEMAPHORE[] at the same time. All interrupt requests should be disabled before trying to access the software semaphore and reenabled after the access is finished, to ensure the consistency of SOFTSEMAPHORE[] from tasks on the same core.

D. Implementation of MPCP on Erika

As each resource in MPCP has a priority-based waiting queue for the management of blocked tasks, we use LockHardSemaphore() and UnlockHardSemaphore() to preserve its consistency. When a task tries to access a locked resource, it suspends itself by invoking WaitEvent() after it is inserted into the queue. When a task finishes the execution of the global

Algorithm 5: Implementation of Software Semaphore

```
Data: SOFTSEMAPHORE [1,...,NR]; // NR: Num of resources
Data: hardsemID; // id of hardware semaphore used to implement
      software semaphores
Data: constant cpuID; // ID of current CPU, 0 for z6, 1 for z0
LockSoftSemaphore(resourceID);
begin
    integer locked_val = cpuID + 1;
    while SOFTSEMAPHORE[resourceID] != locked_val do
        if SOFTSEMAPHORE[resourceID] == 0 then
            LockHardSemaphore(hardsemID);
            if SOFTSEMAPHORE[resourceID] == 0 then
                SOFTSEMAPHORE[resourceID] = locked_val;
            end
            UnlockHardSemaphore(hardsemID);
        end
    end
end
UnlockSoftSemaphore(resourceID);
begin
    if SOFTSEMAPHORE[resourceID] == cpuID + 1 then
        SOFTSEMAPHORE[resourceID] = 0;
    end
end
```

critical section, it is removed from the waiting queue, regains its nominal priority and then informs the first task in the queue by invoking SetEvent(). The task at the head of the queue wakes up, raises its priority to the global ceiling, and enters the global critical section.

E. Implementation on Trampoline

To show the generality of the time/space overhead comparison of the proposed mechanisms, we ported Trampoline [10], an open-source OSEK/VDX RTOS, to the MPC5668G microcontroller and then implemented in it the communication and resource protection mechanisms. We leveraged the resource access APIs (GetResource, ReleaseResource, WaitEvent, SetEvent), supported by Trampoline as kernel-level function calls, to implement the MSRP and MPCP protocols. Whenever an API is invoked, an `sc` (system call) instruction is executed to jump to an exception handler. When entering or leaving the exception handler, a context switch occurs between the user and the kernel levels. The wait-free methods are implemented similarly as in Erika by using the hardware semaphore.

IV. Experimental Evaluation

We compare the performance of MPCP, MSRP, and wait-free algorithms on a large number of randomly generated task configurations, by experiments on an MPC5668G evaluation board. A set of 4 to 20 tasks is randomly generated on each core. Tasks are activated by timer events with zero offsets, and their periods are randomly selected from the set $\{5, 10, 20, 40, 50, 100, 200, 400, 500, 1000\}$ ms. Task priorities are assigned according to rate-monotonic policy. Different mechanisms have different time and memory overheads depending on the amount of communication, and we consider three communication schemes: light, medium, and heavy.

In the **light** communication scheme, the output of a task is shared with 1 to 3 other tasks, with a probability p of 50%, 40%, and 10% respectively. The size of the output is randomly selected from 1 (with $p = 30\%$), $4\ (p = 30\%), 24\ (p = 20\%)$ and $128\ (p = 20\%)$ bytes.

TABLE I
Worst Case Access Time to Buffers From Each Core (Unit: CPU Cycle)

Buffer size (byte)	e200z6		e200z0	
	write	read	write	read
1	9	8	22	20
4	16	14	60	48
24	102	93	280	276
48	200	180	544	540
128	522	480	1322	1318
256	1011	944	2632	2606
512	2074	1845	5246	5182

In the **medium** communication scheme, the output of a task is shared with one to four readers, with a probability of 20%, 30%, 30%, and 20%, respectively. The size of the output is randomly selected from 1 (with $p = 10\%$), $4\ (p = 30\%), 24\ (p = 30\%), 128\ (p = 20\%)$, and $256\ (p = 10\%)$ bytes.

In the **heavy** communication scheme, the output of a task is shared by one to five other tasks, with a probability of 10%, 20%, 30%, 30%, and 10%, respectively. The size of the output is 1 (with $p = 10\%$), $4\ (p = 20\%), 24\ (p = 20\%), 48\ (p = 10\%), 128\ (p = 20\%), 256\ (p = 10\%)$, and $512\ (p = 10\%)$ bytes.

The total utilization of the tasks (without the overhead from the communication primitives) is uniformly distributed from 40% to 95%. The length of critical sections is defined in two ways. For the first set of experiments, we measure the worst case time consumed by accessing (writing or reading) the communication buffers, as in Table I. In the second set, we assign each critical section a random worst case execution time, such that the total length of all critical sections in each task ranges from 1% to 10% of the task WCET.

A. Time Overhead

For MPCP and MSRP, time overhead is incurred when entering and leaving a critical section, i.e., for calls to the GetResource() and ReleaseResource() APIs. For wait-free methods, overhead is incurred when finding a buffer and updating the reader and writer pointers (lines 21, 23, and 24 in Algorithm 1 for the writer and lines 5–8 in Algorithm 2 for the readers). For wait-free methods with flow preservation, the overheads are at task activation time, i.e., Writer_activation() and Reader_activation() functions in Algorithms 3 and 4. We use the Lauterbach tracing tool [3] to measure the execution time overheads on more than 100 million random system configurations on an MPC5668G with 120 MHz clock frequency. The timing overheads are reported on each call to these functions (Tables II and III and Fig. 5), which is independent from the communication scheme (light, medium, or heavy). We also measure the accumulated overhead for the task system (Figs. 6–9). In the following, we use *WF and WFFP as the short names for wait-free method and wait-free method with flow preservation*, respectively.

Table II summarizes the overhead value for *a single execution of the API functions*. For the MPCP implementation of GetResource(), the time overhead depends on the number of tasks in the priority-based waiting queue, and the measured value refers to one execution in which there is already one task queued. Likewise, the time by DBP to find a free buffer depends on the

TABLE II
RANGE OF EXECUTION TIMES FOR THE DATA COMMUNICATION API FOR GLOBAL RESOURCES (TIME UNIT: CPU CYCLE)

Method	GetResource	ReleaseResource	Writer Overhead	Reader Overhead
MSRP	82-91	116-129	–	–
MPCP	105-115	157-168	–	–
WF-DBP	–	–	319-353	93-104
WF-TCCP	–	–	43	28
WFFP-DBP	–	–	348-382	116-127
WFFP-TCCP	–	–	29-38	47-52

TABLE III
PROBABILITY DISTRIBUTION OF OVERHEAD FOR MPCP AND MSRP

Time Interval	MSRP		MPCP	
(μs)	GetResource	ReleaseResource	GetResource	ReleaseResource
(0.525,0.575]			4.58%	
(0.575,0.625]			10.81%	
(0.625,0.675]		0.17%		
(0.675,0.725]		18.96%		
(0.725,0.775]		80.87%		2.44%
(0.775,0.825]				12.96%
(0.825,0.875]	0.63%			
(0.875,0.925]	14.85%		83.78%	
(0.975,1.025]	0.48%		0.48%	
(1.025,1.075]	84.02%			
(1.325,1.375]				54.77%
(1.375,1.425]				29.38%
(1.975,2.025]			0.17%	
(3.975,4.025]				0.18%

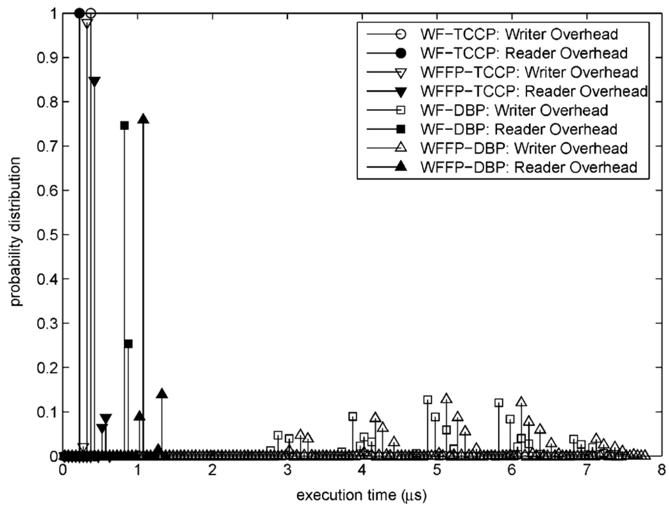


Fig. 5. Probability distribution of overhead for WF and WFFP.

number of buffers and readers, and the value in Table II refers to the case of one reader and a single buffer.

Table III and Fig. 5 summarize the probability distribution of timing overheads for these communication mechanisms. Each point (x, y) in Fig. 5 (or each row in Table III) indicates that there is a probability y that the timing overhead is in the interval $(x - 0.025, x + 0.025]$. The execution time is almost constant for WF-TCCP and WFFP-TCCP (as expected). There is some more variability in the time overheads of WF-DBP and WFFP-DBP as well as the GetResource and ReleaseResource of MPCP and MSRP. The writer overheads (mainly the cost to find a free buffer) in the WF-DBP and WFFP-DBP protocols depend on the number of buffers being used, and are clustered accordingly. For MSRP and MPCP, the overhead is clustered around two values. This is because local and global resources are treated in different ways.

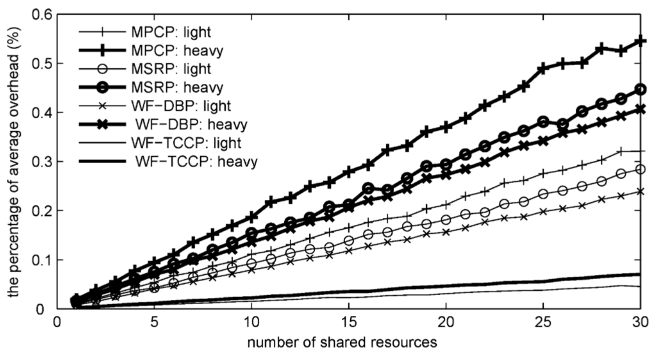


Fig. 6. Average overhead of MPCP, MSRP, WF-DBP, and WF-TCCP in each timer period, measured in Erika.

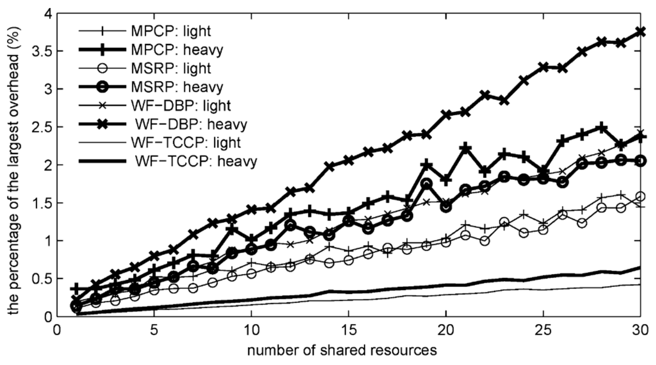


Fig. 7. Largest overhead of MPCP, MSRP, WF-DBP, and WF-TCCP in each timer period, measured in Erika.

The *accumulated time overhead* in the system (for a given time interval) is mainly determined by the number of shared resources, the synchronization policy, and the communication scheme (light, medium or heavy). We conduct experiments with 1 to 30 shared resources. We generate 500 task configurations for each number of resources according to the setup previously described. We measure the maximum and average accumulated overheads within each 5 millisecond interval, the greatest common divisor of the task periods. For better readability, we only plot light and heavy schemes.

The distribution of the average overhead for MPCP, MSRP, WF-DBP, and WF-TCCP is shown in Fig. 6. For a given synchronization protocol, the light communication always leads to a smaller overhead while the heavy communication leads to a larger one. Among the mechanisms, WF-TCCP has the smallest average overhead while MPCP has the largest. For MPCP, a waiting queue should be maintained for each global resource, to store all the tasks currently waiting. When a task is about to release a global resource, it informs the first task in the waiting queue to get the global resource. These operations in the software implementation of the waiting queue incur large overhead to MPCP. Overall, the average overhead is approximately linear to the number of shared resources, with the maximum value of 28 μs for 30 shared resources or 0.56% of the utilization.

Fig. 7 presents the largest accumulated overhead during each timer period. Again, WF-TCCP performs best, while WF-DBP has greater maximum overhead than the ones of MPCP and

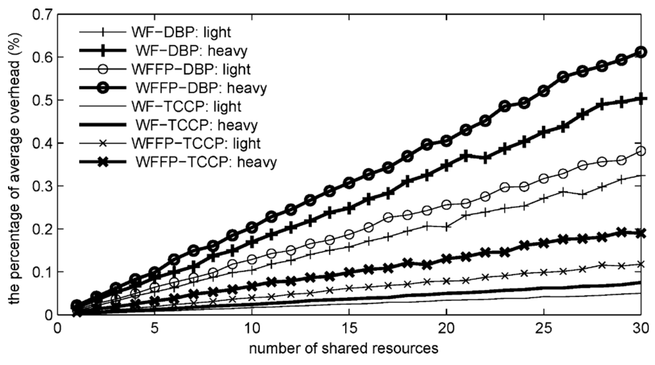


Fig. 8. Average overhead of WF and WFFP in each timer period, in Erika.

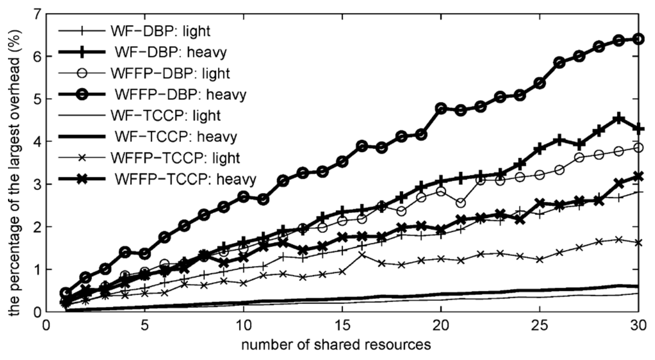


Fig. 9. Largest overhead of WF and WFFP in each timer period, in Erika.

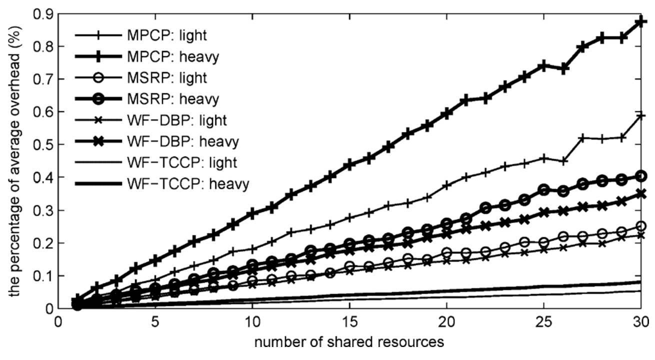


Fig. 10. Average overhead of MPCP, MSRP, WF-DBP, and WF-TCCP in each timer period, measured in Trampoline RTOS.

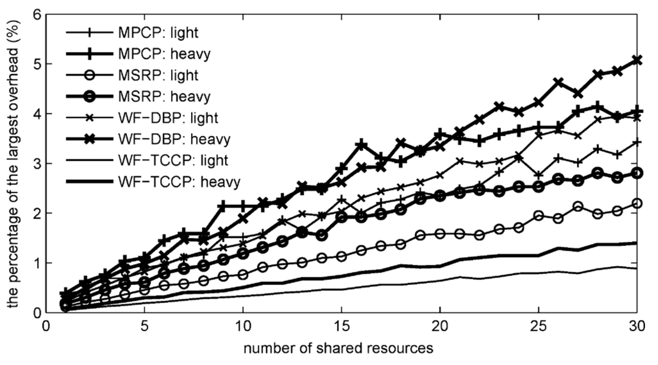


Fig. 11. Largest overhead of MPCP, MSRP, WF-DBP, and WF-TCCP in each timer period, measured in Trampoline RTOS.

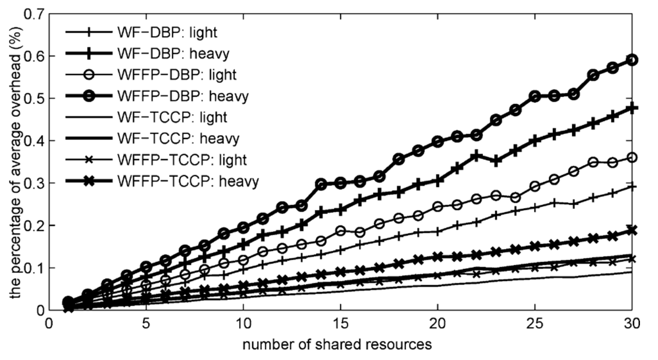


Fig. 12. Average overhead of WF and WFFP in each timer period, measured in Trampoline RTOS.

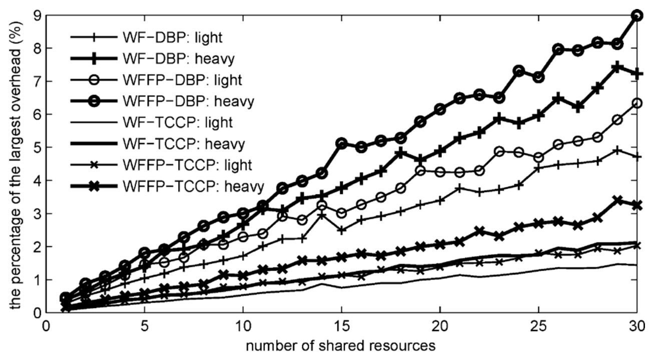


Fig. 13. The largest overhead of WF and WFFP in each timer period, measured in Trampoline RTOS.

MSRP. This is because the overhead of wait-free protocols happens at activation time. In our experiments, the tasks are assumed to be periodic with zero offsets. Thus, there are time instants where all tasks are activated and the WF-DBP overhead accounts for calling the wait-free procedures for all tasks. For MPCP and MSRP, the overhead incurs when actually trying to access the shared resources, thus is distributed over the execution of the task.

The WF-TCCP, WF-DBP, MSRP, and MPCP protocols only provide data consistency, but they do not guarantee the preservation of SR flows. Figs. 8 and 9 show the comparison of the average and largest overheads of wait-free methods with and without flow preservation. The average overhead of WFFP is slightly larger than that of WF in each type of communication scheme. This is mainly because of the need for the lower priority readers to block (using WaitEvent) until the notification (using SetEvent) that the writer finishes writing to the buffer. Nevertheless, the increased overhead is relatively small (less than 20% for 30 resources).

Figs. 10–13 show the overheads of MSRP, MPCP, and the wait-free methods implemented in the Trampoline RTOS. MPCP uses all the kernel APIs and, because of the need to execute all of the required kernel traps, the average time overhead is noticeably larger than that of Erika, as shown in the figures. The time overheads of the other mechanisms are approximately the same as in Erika. In addition, the relative comparison on the overheads of these mechanisms remains the same.

B. Memory Overhead

We also measured the memory overhead by compiling the code and analyzing the memory image of the executable. The

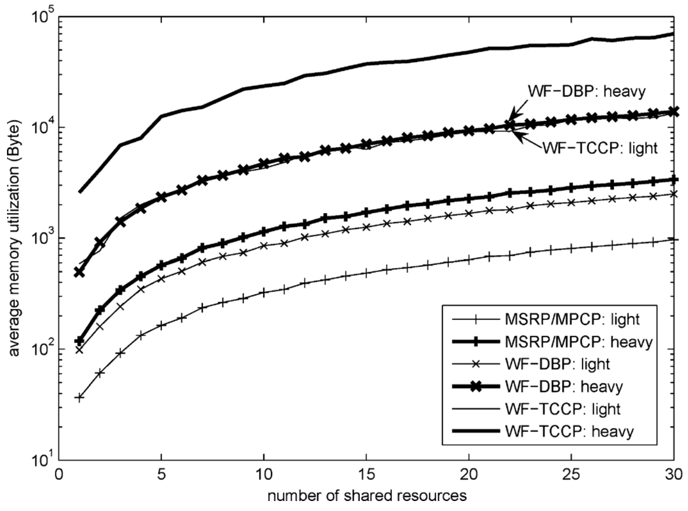


Fig. 14. Memory utilization of MSRP/MPCP, WF-DBP, and WF-TCCP.

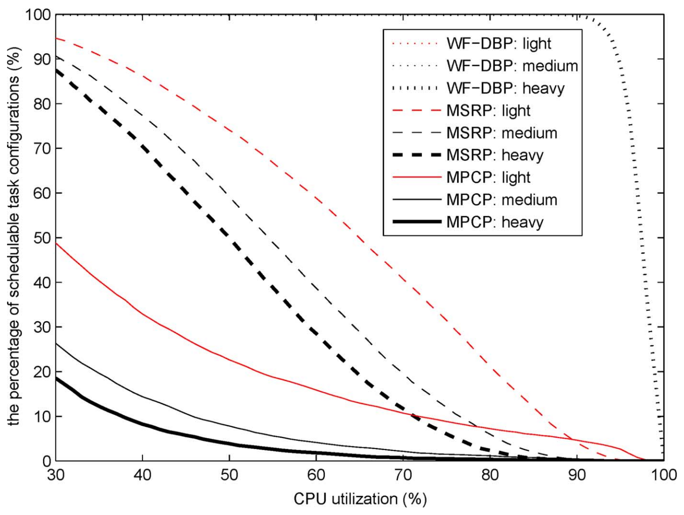


Fig. 15. Percentage of schedulable task configurations with random critical section length.

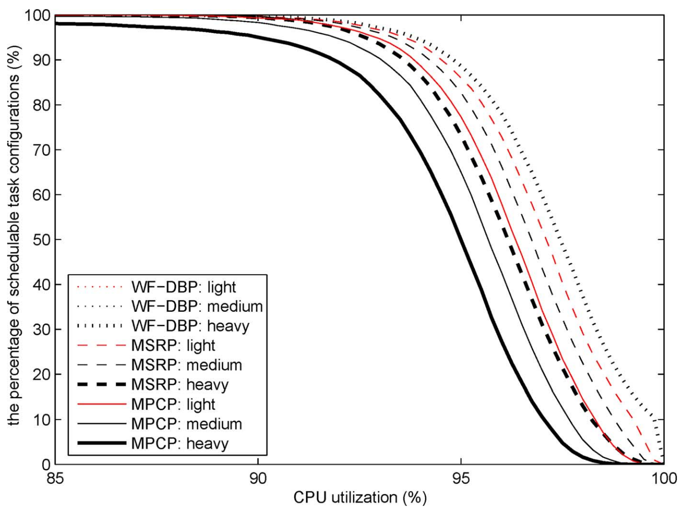


Fig. 16. Percentage of schedulable task configurations with buffer size-related critical section length.

measured memory overhead is the same as what predicted by the estimates (in Section II-B) and shown in Fig. 14. In each case, we generated 500 task configurations, and, for each task configuration, we computed the memory utilization of different protocols. For wait-free methods, each shared resource is assigned with multiple replicas; while for MSRP/MPCP, no additional replicas are required. Accordingly, as we can see in the figure, MSRP/MPCP consumes the least amount of memory. On average, WF-TCCP needs more memory than WF-DBP, although it has less time overhead (as shown in Fig. 6).

C. System Schedulability

An additional comparison is performed with respect to the worst case schedulability performance. As previously shown, the overheads of the communication mechanisms are comparable on both RTOSes with a slight advantage to Erika. In the discussion on system schedulability, we use the time overheads measured on Erika, but the conclusions hold similarly for the Trampoline implementation. The available protocols have a quite different worst case blocking time bound. Among the possible wait-free methods, we select WF-DBP, as its timing overhead is larger than WF-TCCP. For each value of CPU utilization (the sum of the ratio between the WCET and period for all tasks, without including the overhead from the mechanisms), 10 000 task configurations are generated according to the aforementioned task configuration generation scheme. We assume task deadlines are equal to their periods. The metric for system schedulability is the percentage of task sets that are found schedulable among the generated configurations.

The first set of experiments is performed on task configurations with a random critical section length. Fig. 15 shows the results. For the wait-free methods, the three curves (light, medium, and heavy schemes of WF-DBP) coincide with each other, because the difference of timing overheads for different communication schemes is far less than the period and the schedulability results are close. Among the three policies, WF-DBP outperforms the other two. This is expected, since tasks do not block when using a wait-free protocol. For a given communication scheme, MSRP performs better than MPCP at a low CPU utilization. However, MPCP overtakes MSRP when CPU utilization exceeds 90%. This is mainly because the length of critical sections is related to the task WCET, which grows when the CPU utilization increases, and MPCP is better than MSRP for long critical sections [17].

In the second set of experiments, the length of the critical sections is the actual measured time (as in Table I) for accessing (writing or reading) the corresponding communication buffer. The results are shown in Fig. 16. Again, WF-DBP is generally better than MPCP and MSRP, and the three curves of WF-DBP overlap. However, the difference between WF-DBP and MSRP/MPCP is much smaller than in Fig. 15. This is because the measured length of critical sections when accessing shared buffers is generally short and imposes only a small blocking time. Based on the same reason, MSRP performs better than MPCP in the entire range of CPU utilization. The significance of this result is: for a realistic implementation of the critical sections for accessing shared buffers (rather than hypothetic lengths of the critical sections as a fraction of the task WCET), MSRP outperforms MPCP, and the only competitor is wait-free method.

V. Optimization

As discussed in the previous sections, the available protection mechanisms have different timing and memory overheads, and different impact on the system schedulability. MSRP and MPCP cause blocking times in the worst case response time of tasks whenever a lower priority (local) tasks may use a global resource. Wait-free methods avoid the blocking time altogether by making sure that the writer and its readers do not access the same buffer at the same time. Their timing overhead is in the same magnitude as MPCP and MSRP, thus they are typically advantageous to system schedulability. However, they need to make copies of the shared data buffers with a possibly large amount of memory overhead.

In summary, these methods offer tradeoffs between the amount of additional memory, the runtime overheads, and the blocking time imposed over tasks. Accordingly, the optimal system configuration require to use a combination of these methods, depending on the memory availability of the system, the size of the data to be transmitted, the task response times and deadlines, and the duration of the critical sections.

We consider an optimization problem in which the task allocation and priority assignment are given, and the designers must select the mechanism to protect each shared resource. We only focus on the problem of guaranteeing data consistency, as MPCP and MSRP are not applicable for the purpose of flow preservation. The optimization objective is to use the smallest amount of memory while guaranteeing system schedulability.

Between MPCP and MSRP, as pointed out in previous studies [13], [22], MSRP (or in general spin-based mechanisms) is preferable for shared resources with short global critical sections. We use the same idea as in FMLP [12], where short resource requests use a busy-wait mechanism, while long resource requests use a suspension approach. For the two wait-free methods, WF-DBP has a slightly larger timing overhead, but requires less memory than WF-TCCP in most cases.

Based on the above observations, we propose the optimization procedure in Algorithm 6 to minimize the memory usage while guaranteeing schedulability. The four available mechanisms are classified as lock-based and wait-free, and each shared resource is assigned with a preferred mechanism for each type. For wait-free methods (WF-TCCP and WF-DBP), the preference is given to the one with a lower memory cost. For lock-based methods (MPCP and MSRP), the shared resources are ordered in decreasing length of their longest critical sections and the list is partitioned in two subsets. As mentioned before, MPCP has a smaller blocking time than MSRP for shared resources with long critical sections, so MPCP is assigned as the preferred lock-based mechanism for the first subset (the resources with longer critical sections) and MSRP is the default one for the remaining resources. However, the resource partitioning is not fixed as in FMLP, but we exhaustively explore all the possible partitions (the list is tentatively split at each possible position, i.e., i from 0 to n where n is the number of resources).

Algorithm 6: Algorithm for Selection of Communication Mechanisms

```
 1: Assign a preferred wait-free method to the shared
    resources
 2: for $i = 0$ to $n$ do
 3:   Assign MPCP as the preferred mechanism to the first
      $i$ resources and MSRP for the remaining resources
 4:   Use wait-free method for all resources
 5:   Order shared resources by decreasing memory saving
 6:   for each shared resource $\xi_i$ do
 7:     Use the preferred mechanism (MPCP or MSRP)
        to protect $\xi_i$
 8:     if system is unschedulable then
 9:       Use the other lock-based mechanism to protect $\xi_i$
10:       if system is unschedulable then
11:         Reset the mechanism to WF-TCCP/WF-DBP
12:       end if
13:     end if
14:   end for
15: end for
```

As an initial solution, all resources are protected by their preferred wait-free method. Then, for each resource, we compute the amount of *memory saving* if the wait-free method is replaced by MSRP/MPCP. The resources are sorted according to the amount of memory that can be saved by using a lock-based method. Starting from the first resource in the list, we first try to apply to it its preferred locking mechanism (MPCP or MSRP), and then try the other method if the preferred one fails (resulting in an unschedulable solution). If both attempts fail, the protection mechanism is set back to the preferred wait-free method (WF-TCCP or WF-DBP).

This greedy algorithm can produce a local optimum if some resource is protected with a lock-based method at the cost of a large blocking time, which prevents the use of MPCP or MSRP for the following resources in the list. The decisions taken in the early steps (the resources with largest memory saving) can lead to limited options later. For this reason, we refine the algorithm with an exhaustive search limited to a selected neighborhood of the first solution found by the algorithm. After finding the first feasible solution S, we exhaustively explore the use of wait-free and lock-based mechanisms for the first k resources using a lock-based mechanism in S. k is the *depth of the exhaustive search* in this refinement step. The resulting complexity of the algorithm is $O(n \cdot 3^k)$, as compared with the total number of possible solutions $O(4^n)$. As an alternative, we also tried a backtracking search with limited depth, but it did not perform equally well, as shown in our experimental results.

A. Evaluation of the Optimization Algorithm

To evaluate the quality of the results produced by Algorithm 6, we first perform experiments on 10 000 random task configurations with utilization ranging from 45% to 95% and randomly generated task-critical sections. The results of our algorithm are compared against the optimum results obtained by an exhaustive search. Our heuristic produces solutions close to the op-

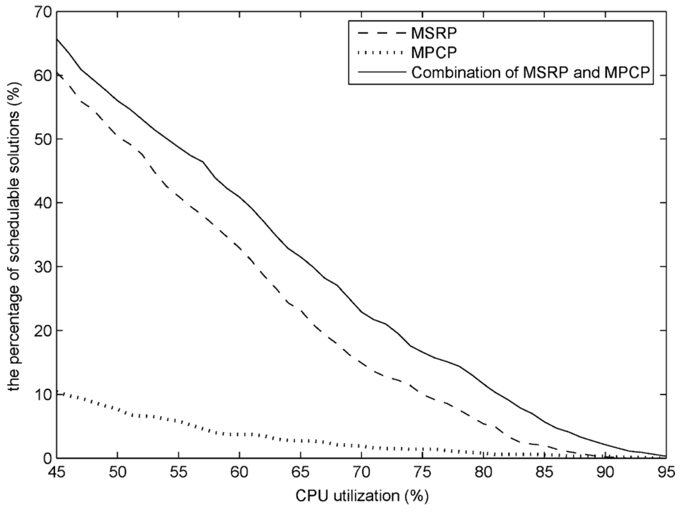


Fig. 17. Schedulability with MPCP and MSRP only.

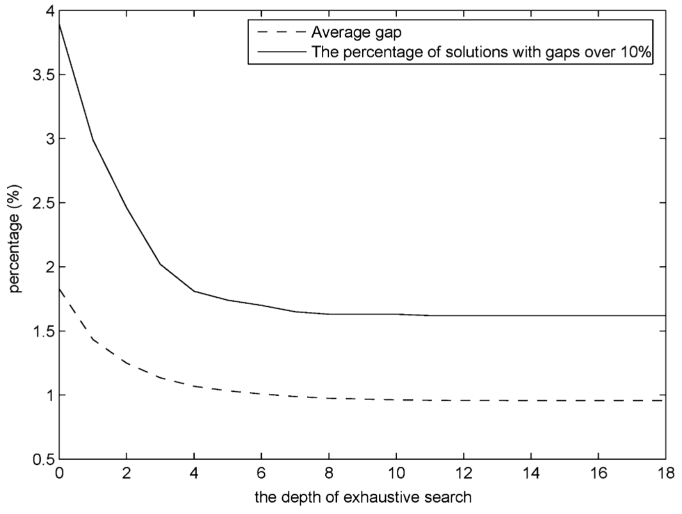


Fig. 18. Gap between the heuristic and optimal solution.

timum in most cases. For example, for systems with 20 globally shared resources, in 55.9% of the cases, the result is exactly the optimum; 22.9% of the cases, the gap to the optimum is <1%; 16.0% of the cases, the gap is between 1% and 5%; in 3.6% of the cases, it is between 5% and 10%; and only in 1.6% of the cases, the gap is larger than 10%, with a maximum value of 47.1%. The average difference is 0.96%.

Fig. 17 shows the fraction of systems that are schedulable by using only MSRP and MPCP. The percentage of feasible configuration starts to drop quite early (for utilizations that are lower than 50%!) and becomes extremely low at 80%. All of the task sets considered in this experiment are schedulable using wait-free methods. The purpose of the graph is to show the improvement on system schedulability that can be obtained by using wait-free methods together with lock-based methods (restoring schedulability at the expense of memory).

Fig. 18 shows the gap between the memory requirements from Algorithm 6 and the optimum solution for different settings of the depth k. As expected, the gap reduces as k gets larger, but very little improvements can be obtained for k larger

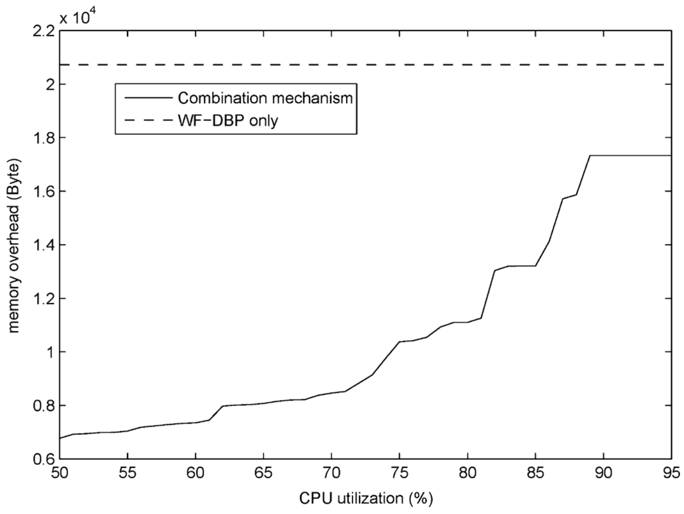


Fig. 19. Memory usage with respect to system utilization for the case study in [32].

than 5. As a comparison, using backtracking can only get similar results ($\mathrm{gap} = 0\%$ for 49.9% cases, $\mathrm{gap} < 1\%$ for 21.8%, $\mathrm{gap} < 5\%$ for 18.8%, $\mathrm{gap} < 10\%$ for 5.4%, $\mathrm{gap} \geq 10\%$ for 4.1%) with a backtracking depth of 10.

We also apply the optimization procedure to two real case studies. The first one, provided by a car electronics supplier, consists of a fuel injection system [32]. The system is a complex network of functions, in which approximately two hundred AUTOSAR runnables (functions called in response to events) execute at different rates and communicate by exchanging data signals. The runnables are mapped into 16 tasks, with periods of 4, 5, 10, 12, 50, 100, and 1000 ms, respectively. These tasks share a total of 46 resources, with size from 1 to 512 bytes and 5032 bytes in total. If WF-TCCP is applied to all resources, 278468 bytes of memory are needed, while WF-DBP only requires 20 722 bytes of memory. We generate variations of the system configuration by scaling the WCET of tasks so that the utilization on each core is between 50% and 95%. As shown in Fig. 19, the memory requirements of the system configuration produced using Algorithm 6 range from 6773 bytes to 17 330 bytes. Compared with the use of wait-free methods (WF-DBP and WF-TCCP) only, our algorithm significantly reduces the overall memory requirements (more than 50% and 95% of memory is saved, respectively, when CPU utilization is below 75%). When the CPU utilization is 89% (or more), no resources can be protected by using MSRP or MPCP and every resource must be protected using either WF-DBP or WF-TCCP, depending on which one requires less memory. As a result, the memory requirements are practically unchanged after this point.

The second case study is a publicly available system description, defined in [30]. It consists of 42 tasks with periods of 14, 20, 35, and 60 ms, respectively. There are 36 resources in the case study, with buffer size ranging from 20 to 150 bytes, and 2240 bytes in total. The original WCETs of the tasks refer to a system configuration that was used to demonstrate a task allocation algorithm. For our experiments, we use the original WCETs, but we also scale them to generate additional task sets

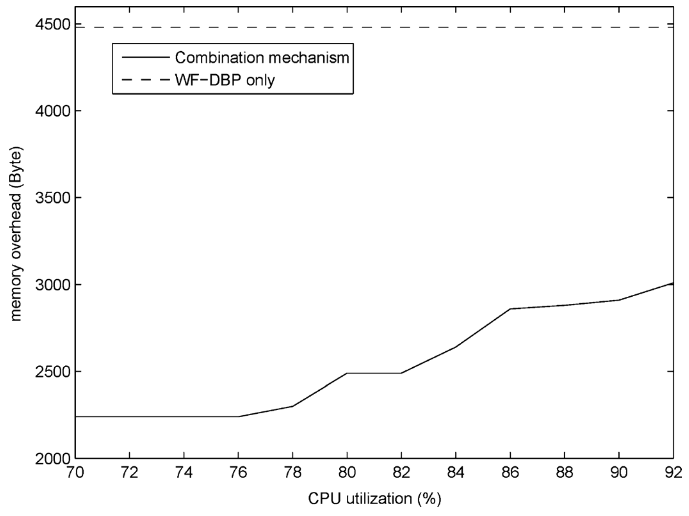


Fig. 20. Memory usage with respect to system utilization for the case study in [30].

with an average core utilization between 70% and 92% (the system is unschedulable with a utilization higher than 92%). The memory required for running the system after our optimization algorithm selects the protection method for each shared memory buffer is shown in Fig. 20. For this example, the least amount of memory that can be possibly required is 2240 bytes when using only MSRP or MPCP. The implementation with minimum time overheads and without blocking times, using WF-DBP (with best schedulability) needs 4480 bytes. Since each resource in the case study is shared just by two tasks, the blocking time of MPCP/MSRP is relatively short. As a result, all systems with average core utilization up to 76%, can be scheduled with only MPCP and MSRP (using only 2240 bytes of memory). When the CPU utilization for the application increases, so does the required memory. But at 92%, our optimization algorithm still finds that many resources can be protected by using MPCP/MSRP with an overall memory requirement of 3010 bytes, very close to the best scenario and far from the amount required by a system-wide use of WF-DBP (4480 bytes).

VI. Conclusion

We describe the issues in the implementation and selection of shared resource protection mechanisms on multicore platforms. 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 response (and time overheads) and demand for additional memory. We implement them on a real-time operating system, to measure time and memory overheads and compare their impact on system schedulability. We propose an algorithm to optimally select the communication mechanisms with minimum memory requirements executing within the time constraints. The comparison and optimization are validated using synthetic systems and an industrial case study.

References

[1] "The AUTOSAR Standard," AUTOSAR consortium, spec. v. 4.0 [Online]. Available: http://www.autosar.org
[2] ERIKA Enterprise [Online]. Available: http://erika.tuxfamily.org
[3] "TRACE 32 In-Circuit Debugger," Lauterbach [Online]. Available: http://www.lauterbach.com/
[4] *"MathWorks Simulink and StateFlow User's Manuals,"* MathWorks [Online]. Available: http://www.mathworks.com
[5] *"MPC5668x Microcontroller Reference Manual,"* Freescale [Online]. Available: http://www.freescale.com
[6] OSEK/VDX Operating Systems . ver. 2.2.3, OSEK, 2006 [Online]. Available: http://www.osek-vdx.org
[7] J. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Trans. Comput. Syst.*, vol. 15, pp. 134–165, May 1997.
[8] T. Baker, "A stack-based resource allocation policy for realtime processes," in *Proc. 11th IEEE Real-Time Systems Symp.*, 1990, pp. 191–200.
[9] M. Baleani, A. Ferrari, L. Mangeruca, and A. S. Vincentelli, "Efficient embedded software design with synchronous models," in *Proc. 5th ACM Conf. Embedded Software*, 2005, pp. 187–190.
[10] J. Béchennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline—An open source implementation of the OSEK/VDX RTOS specification," in *Proc. 11th Int. Conf. Emerging Technol. Factory Automation*, 2006, pp. 62–69.
[11] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.
[12] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Proc. IEEE Conf. Embedded and Real-Time Computing Syst. Applic.*, 2007, pp. 47–56.
[13] B. Brandenburg and J. Anderson, "A comparison of the M-PCP, D-PCP, and FMLP on litmus^{rt}," in *Proc. 12th Int. Conf. Principles of Distrib. Syst.*, 2008, pp. 105–124.
[14] J. Chen and A. Burns, "Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties," in *Proc. Int. Conf. Real-Time Computing Syst. Applic.*, 1999, pp. 236–246.
[15] R. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, pp. 35:1–35:44, 2011.
[16] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *Proc. 30th IEEE Real-Time Syst. Symp.*, 2009, pp. 377–386.
[17] 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 Syst. Symp.*, 2001, pp. 73–83.
[18] M. Herlihy, "A methodology for implementing highly concurrent structures," in *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 1990, pp. 197–206.
[19] H. Huang, P. Pillai, and K. G. Shin, "Improving wait-free algorithms for interprocess communication in embedded real-time systems," in *Proc. USENIX Annu. Tech. Conf.*, 2002, pp. 303–316.
[20] H. Kopetz and J. Reisinger, "The non-blocking write protocol NBW: A solution to a real-time synchronization problem," in *Proc. IEEE Real-Time Syst. Symp.*, 1993, pp. 131–137.
[21] S. Kumar, C. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proc. 34th Annu. Int. Symp. Comput. Architecture*, 2007, pp. 162–173.
[22] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proc. 30th IEEE Real-Time Syst. Symp.*, 2009, pp. 469–478.
[23] A. Marongiu, P. Burgio, and L. Benini, "Supporting OpenMP on a multi-cluster embedded MPSoC," *Microproces. Microsyst.—Embedded Hardware Design*, vol. 35, no. 8, pp. 668–682, 2011.
[24] A. Naeem, X. Chen, Z. Lu, and A. Jantsch, "Scalability of weak consistency in NoC based multicore architectures," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2010, pp. 3497–3500.
[25] F. Nemati, M. Behnam, and T. Nolte, "Independently-developed real-time systems on multi-cores with shared resources," in *Proc. 23th Euromicro Conf. Real-Time Syst.*, 2011, pp. 251–261.
[26] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. Int. Conf. Distrib. Computing Syst.*, 1990, pp. 116–123.
[27] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[28] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," in *Proc. Int. Conf. Embedded Software*, 2006, pp. 21–33.

[29] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase, "Hardware synchronization for embedded multi-core processors," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2011, pp. 2557–2560.

[30] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating hard real-time tasks: An NP-Hard problem made easy," *Real-Time Syst.*, vol. 4, no. 2, pp. 145–165, June 1992.

[31] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi, "Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or EDF schedulers," in *Proc. 5th ACM Conf. Embedded Software*, 2005, pp. 353–360.

[32] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli, "Improving the size of communication buffers in synchronous models with time constraints," *IEEE Trans. Ind. Inf.*, vol. 5, no. 3, pp. 229–240, Aug. 2009.

[33] 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 Int. Symp. Ind. Embedded Syst.*, 2011, pp. 140–149.

[34] E. Estevez and M. Marcos, "Model-based validation of industrial control systems," *IEEE Trans. Ind. Inf.*, vol. 8, no. 2, pp. 302–310, May 2012.

[35] B. Alecsa, M. N. Cirstea, and A. Onea, "Simulink modeling and design of an efficient hardware-constrained FPGA-based PMSM speed controller," *IEEE Trans. Ind. Inf.*, vol. 8, no. 3, pp. 554–562, Aug. 2012.

[36] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Trans. Ind. Inf.*, vol. 9, no. 3, pp. 1234–1249, Jul. 2013.

[37] B. Brandenburg, "A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications," in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 292–302.

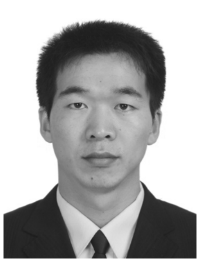
[38] Z. Al-Bayati, H. Zeng, M. Di Natale, and Z. Gu, "Multitask implementation of synchronous reactive models with earliest deadline first scheduling," in *Proc. 8th IEEE Int. Symp. Ind. Embedded Syst.*, 2013, pp. 168–177.

[39] H. Zeng and M. Di Natale, "Efficient implementation of AUTOSAR components with minimal memory usage," in *Proc. 7th IEEE Int. Symp. Ind. Embedded Syst.*, 2012, pp. 130–137.

[40] A. Wieder and B. Brandenburg, "Efficient partitioning of sporadic real-time tasks with shared resources and spin locks," in *Proc. 8th IEEE Int. Symp. Ind. Embedded Syst.*, 2013, pp. 49–58.

[41] B. Ward and J. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *Proc. 24th Euromicro Conf. Real-Time Syst.*, 2012, pp. 223–232.

[42] B. Nahill, A. Ramdial, H. Zeng, M. Di Natale, and Z. Zilic, "An FPGA implementation of wait-free data synchronization protocols," in *Proc. 18th IEEE Int. Conf. Emerging Technol. Factory Autom.*, 2013, pp. 1–8.



Gang Han received the B.E. and M.E. degrees in computer science and technology from National University of Defense Technology, Changsha, China, in 2006 and 2008, respectively, where he is currently working toward the Ph.D. degree.

He was a Visiting Scholar with McGill University, Montreal, QC, Canada, in 2011–2012. His research interests include design and optimization of real-time systems, wireless sensor networks, and datacenter networks.

Haibo Zeng (M'08) received the B.E. and M.E. degrees in electrical engineering from Tsinghua University, Beijing, China, in 1999 and 2002, respectively, and the Ph.D. degree in electrical engineering and computer sciences from the University of California, Berkeley, CA, USA, in 2008.

He is currently an Assistant Professor with McGill University, Montreal QC, Canada. He was a Senior Researcher with General Motors R&D until October 2011. He has authored or coauthored over 50 papers in the above fields. His research interests are design methodology, analysis, and optimization for embedded systems, cyber-physical systems, and real-time systems.

Dr. Zeng was the recipient of three Best Paper Awards.

Marco Di Natale (SM'13) received the B.S. degree from the University of Pisa, Pisa, Italy, in 1991, and the Ph.D. degree from Scuola Superiore S.Anna, Pisa, in 1995.

is an Associate Professor with the Scuola Superiore Sant'Anna, Pisa, Italy, where he leads the area on embedded architectures and models. He was a Visiting Researcher with the University of California, Berkeley, CA, USA, in 2006 and 2008–2009. He has authored or coauthored more then 150 papers. He was a Senior Scientist and Group Leader for automotive architectures exploration and selection at General Motors R&D and is currently a Visiting Fellow with the United Technologies corporation. His research interests are in embedded systems architecture and behavior models, real-time systems and system analysis.

Dr. Di Natale is a member of the editorial board of the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS and chair of the Real-Time Systems subcommittee of the IEEE IES TCFA. He was the recipient of five Best Paper Awards and one Best Presentation Award. He has served as a Program and General Chair for the Real-Time Application Symposium and Track Chair for the DATE Conference.

Xue Liu (M'06) received the B.S. degree in mathematics and M.S. degree in automatic control from Tsinghua University, Beijing, China, in 1996 and 1999, respectively, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana, IL, USA, in 2006.

He is an Associate Professor with the School of Computer Science, McGill University, Montreal QC, Canada. He has also worked as the Samuel R. Thompson Associate Professor with the University of Nebraska-Lincoln, Lincoln, NE, USA, and Hewlett-Packard Labs, Palo Alto, CA, USA. His research interests are in computer networks and communications, smart grid, real-time and embedded systems, cyber-physical systems, data centers, and software reliability. He holds one U.S. patent, has filed four other U.S. patents, and has authored or coauthored more than 150 research papers in major peer-reviewed international journals and conference proceedings,

Dr. Liu was the recipient of the 2008 Best Paper Award from the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS and the First Place Best Paper Award of the ACM Conference on Wireless Network Security.

Wenhua Dou received the B.E. degree in computer from Harbin Institute of Engineering, Harbin, China, in 1970.

He is a Professor with the School of Computer Science, National University of Defense Technology, Changsha, China. He has authored or coauthored more than 200 papers in journals and conferences. His current research interests include high-performance computing, photonic interconnection and communication, wireless networks, network calculus, and network coding.