Task Placement and Selection of Data Consistency Mechanisms for Real-Time Multicore Applications

Zaid Al-bayati[†], Youcheng Sun[‡], Haibo Zeng[¶], Marco Di Natale[‡], Qi Zhu[§], and Brett Meyer[†]
McGill University[†], Scuola Superiore Sant'Anna[‡], Virginia Tech[¶], Univ. of California at Riverside[§]
E-Mails: {zaid.al-bayati@mail.,brett.meyer@}mcgill.ca, {y.sun,marco}@sssup.it, hbzeng@vt.edu, qzhu@ee.ucr.edu

***Abstract*—Multicores are today used in automotive, controls and avionics systems supporting real-time functionality. When real-time tasks allocated on different cores cooperate through the use of shared communication resources, they need to be protected by mechanisms that guarantee access in a mutual exclusive way with bounded worst-case blocking time. Lock-based mechanisms such as MPCP and MSRP have been developed to fulfill this demand, and research papers are today tackling the problem of finding the optimal task placement in multicores while trying to meet the deadlines against blocking times. In this paper, we provide algorithms that improves on existing task placement algorithms for systems that use MSRP to protect shared resources. Furthermore, we leverage the additional opportunity provided by wait-free methods as an alternative data consistency mechanism for the case that the shared resource is communication or state memory. The selective use of wait-free communication can further significantly extend the range of schedulable systems and consequently the design space, at the cost of memory.**

I. INTRODUCTION

Multicore architectures have become commonplace in general computing and multimedia applications, and are rapidly advancing in typical embedded computing systems, including automotive and controls. Partitioning 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 design flow to leverage the availability of parallel processing.

In this paper, we consider *partitioned fixed-priority scheduling*, where tasks are statically assigned to cores and each core is scheduled by a local fixed-priority scheduler. This is widely used in embedded multicore real-time systems today. Such scheduling policy is supported by the AUTOSAR standard for automotive systems [1], as well as most commercial RTOSes, including VxWorks, QNX, LynxOS, and all POSIX-compliant ones. The designer of a partitioned system with real-time constraints needs to define a static mapping of tasks to cores, such that all tasks are guaranteed to meet their deadlines. When tasks share (communication) resources, the problem requires the consideration of the possible blocking times, which are a function of the task allocation and the selection of the data consistency mechanisms. The problem of finding such a feasible solution is demonstrated to be NP-hard (even in the special case of no shared resources, the problem is an instance of bin-packing [15]).

In real-time applications on multicore architectures, communication resources can be shared using lock-based or wait-free methods. **Lock-based methods** include locks with suspension, as in MPCP (Multiprocessor Priority Ceiling Protocol [24]), and spin locks, as in MSRP (Multiprocessor Stack Resource Policy [17]). They both provide a bounded worst-case blocking time, with MSRP being simpler to implement and providing better performance for short critical sections (see e.g. [7]). Also, the AUTOSAR operating systems standard [1] mandates the use of spin locks for lock-based inter-core synchronization.

When the shared resource is a **communication buffer** (memory used for communicating data), another possibility is to use **wait-free methods**. The writer and readers are protected against concurrent access by replicating the communication buffer and leveraging information on the time instant and order (priority and scheduling) of the buffer access [13], [20]. Wait-free methods have virtually no blocking time (in reality often negligible) and may enhance the schedulability of the system at the cost of additional memory. Of course, they require the replication of the shared resource, thus are not applicable if the shared resource is a hardware device. We will discuss how the two methods compare in the next sections.

State of the Art

Allocation problems are very common in multicore systems, and in most cases they are proven to be special instances of the general bin-packing [21] problem. In the context of real-time systems without considering shared resources, several heuristics have been proposed, e.g., [6], [15], [16]. For the case of independent tasks, Baruah and Bini presented an exact ILP (Integer Linear Programming) based approach for partitioning [5]. Chattopadhyay and Baruah showed how to leverage lookup tables to enable fast, yet arbitrarily accurate partitioning [11]. Finally, Baruah presented a polynomial-time approximation scheme [4]. Similarly, solutions for the partitioned scheduling of independent sporadic tasks are presented in [4], [14].

MPCP [24] and MSRP [17] are protocols for sharing resources with predictable blocking times in multicore platforms. The response time bound in [17] for MSRP has been recently improved using an ILP formulation, as presented in [28]. MPCP and MSRP have been compared in a number of research works (with respect to the worst-case timing guarantees), with general consensus that MSRP performs best for short (global) critical sections and MPCP for large ones. A more detailed discussion on the design options and the characteristics of lock-based mechanisms can be found in [8]. There is also abundant research on the development of new locked-based protocols, most recently the FMLP$^+$ by Brandenburg [9].

Wait-free mechanisms are an alternative solution for preserving data consistency in multicore communication [13]. They can be extended to guarantee semantics preservation of synchronous models [29] and sized according to the time properties of communicating tasks [25], [26].

With respect to the real-time task partitioning problem, when considering the possible access to global shared resources, Lakshmanan et al. [22] presented a partitioning heuristic tailored to MPCP. This heuristic organizes tasks sharing resources into groups in order to assign them to the same processor. In subsequent work, Nemati et al. presented BPA [23], another partitioning heuristic for MPCP. It tries to identify communication clusters such that globally shared resources are minimized and the blocking time is reduced.

The first work to provide solutions for the partitioning problem when using spin locks (MSRP) is [27]. Two solutions are proposed in the paper. One is the ILP optimization formulation that can provide the optimal solution, but its runtime is exponentially increasing with the problem size. The other is the Greedy Slacker (GS) heuristic, which obtains good-quality solutions in a much shorter time. GS assigns tasks to cores sequentially according to a simple greedy policy that tries to maximize the least slack (i.e., the difference between the deadline and the worst case response time). [18] provides implementations of MPCP, MSRP, and wait-free methods on two open source RTOSes and the selection of these data consistency mechanisms that satisfies the schedulability constraints, but the task placement is assumed to be given.

Our Contributions

In this work, we consider MSRP as the option of lock-based synchronization protocol. We first explore the possibility of improving on the Greedy Slacker algorithm. The proposed algorithm, defined as Communication Affinity and Slack with Retries (CASR), attempts to improve on the GS allocation strategy in two ways: it considers task communication affinity in the allocation and includes a recover-and-retry mechanism in those cases where the slack-based strategy fails to find a feasible allocation.

Next, we consider an additional design option when the shared resource is a communication buffer, by selectively replacing MSRP-protected resources with those protected by a wait-free mechanism. We combine these two methods for managing global resources to find a system configuration that is schedulable with the minimum amount of required memory. We present two approaches that try to find the optimal task placement and selection between MSRP and wait-free methods for global communication resources. The first (GS-WF) extends the GS heuristic by only using wait-free methods when GS fails to allocate a task. The algorithm assumes MSRP as the default mechanism, but uses wait-free methods when the system is unschedulable with MSRP. The second approach, MPA (Memory-aware Partitioning Algorithm), is a heuristic that leverages wait-free methods from the start (while still trying to minimize the overall memory cost). Experiments show that GS-WF and MPA can significantly improve the schedulability of the systems with modest memory cost. Between these two algorithms, MPA provides better results in terms of both schedulability and memory when there is a large number of shared resources.

The paper is organized as follows. We define our system model in Section II, and summarize MSRP and wait-free methods with the schedulability analysis results in III. In Section IV, we present the CASR algorithm. In Section V we describe the two approaches for selecting the data consistency mechanisms. We give an illustrative example in Section VI. In Section VII, we evaluate and compare the results of the four algorithms (GS, CASR, their extension with wait-free methods, and MPA) in terms of system schedulability and memory cost. Finally, Section VIII concludes the paper.

II. System Model

The system under consideration consists of m cores $\mathcal{P} = \{p_1, p_2, \dots, p_m\}$. n tasks $\mathcal{T} = \{\tau_1, \tau_2, \dots \tau_n\}$ are statically allocated to them and scheduled by static priority. 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 composed of a set of alternating critical sections and sections in which τ_i 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 $\{C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, ..., C'_{i,s(i)-1}, C_{i,s(i)}\}$, where $s(i)$ is the number of normal execution segments, and $s(i)-1$ is the number of critical sections. $C_{i,j}$ ($C'_{i,j}$) is the WCET of the j-th normal execution segment (critical section) of τ_i. 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 P_i the core where it executes. The global shared resource associated to the j-th critical section of τ_i is $\mathcal{S}_{i,j}$. The WCET C_i of τ_i is

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

The worst case response time R_i of τ_i can be calculated as the least fixed-point solution of the formula (2), where B_i^l is the local blocking time, and B_i^r the remote blocking time, that is, the worst-case length of the time interval in which the task has to wait because of critical sections executed by lower priority tasks on local and global resources, respectively. C_i^* is the worst-case execution time for task τ_i after considering the possible additional time spent while spinning on a global lock (in the case of spin lock based protocols).

$$R_i = C_i^* + B_i^l + B_i^r + \sum_{\pi_h < \pi_i \wedge P_h = P_i} \left\lceil \frac{R_i}{T_h} \right\rceil C_h^* \tag{2}$$

III. MSRP and Wait-free Communication: Blocking Time vs. Memory

We propose to combine MSRP and wait-free methods to improve system schedulability while keeping the memory overhead low. We first introduce them and highlight the tradeoff between the associated blocking time and memory cost.

A. Multiprocessor Stack Resource Policy

MSRP (Multiprocessor Stack Resource Policy) [17] is a multiprocessor synchronization protocol, derived by extension from the single-core Stack Resource Policy (SRP) [3]. In SRP, each resource has a ceiling equal to the highest priority of any tasks that may access it. At runtime, a task executing a critical section immediately changes its priority to the ceiling of the corresponding resource. After the task starts, it never blocks when requesting a shared resource, but can possibly wait for lower priority tasks to terminate their critical sections while being in the ready queue. The local blocking time B_i^l of task τ_i is bounded as the longest critical section of a lower priority task on the same core accessing a resource with a ceiling higher than π_i.

$$B_i^l = \max_{k:\pi_k > \pi_i \wedge P_k = P_i} \left\{ \max_{1 \leq m < s(k)} C'_{k,m} \right\} \tag{3}$$

In the multicore extension MSRP, global resources are assigned with a ceiling that is higher than that of any local resource. A task that fails to lock a global resource (in use by another task) *spins* on the resource lock until it is freed, keeping the processor busy. (In the MPCP protocol, for comparison, the task is suspended and yields the CPU.) To minimize the spin lock time (wasted CPU time), tasks cannot be preempted when executing a global critical section, in an attempt to free the resource as soon as possible. MSRP uses a First-Come-First-Serve queue (as opposed to a priority-based queue in MPCP) to manage the tasks waiting on a lock for a given busy resource.

In this paper, we adopt the sufficient analysis on the global blocking time [17]. Recently, a more accurate analysis of the blocking time in MSRP based on an ILP formulation has been proposed [28]. However, for the use in an optimization problem in which many solutions need to be evaluated quickly, the significant runtime makes it unscalable to large systems, as demonstrated in Section VII.

The spin time $L_{i,j}$ that a task τ_i may spend for accessing a global resource $\mathcal{S}_{i,j}$ can be bounded by [17]

$$L_{i,j} = \sum_{E \neq E_i} \left\{ \max_{\tau_k : P_k = P_i, 1 \leq m < s(k)} C'_{k,m} \right\} \tag{4}$$

This is the time increment to the j-th critical section of τ_i. Thus, the total worst case execution time C_i^* is

$$C_i^* = C_i + \sum_{1 \leq j < s(i)} L_{i,j} \tag{5}$$

MSRP maintains the same basic property of SRP, that is, once a task starts execution it cannot be blocked. The local blocking time B_i^l is the same as in SRP (Equation (3)) and the worst-case remote blocking time B_i^r is [17]

$$B_i^r = \max_{k:\pi_k > \pi_i \wedge P_k = P_i} \left\{ \max_{1 \leq m < s(k)} (C'_{k,m} + L_{k,m}) \right\} \tag{6}$$

B. Wait-free Communication Buffers

The objective of wait-free methods is to avoid blocking by ensuring that each time a writer needs to update the communication data, it is reserved with a new buffer. Readers are free to use other dedicated buffers. Figure 1 shows the typical stages performed by the writer and the readers in a wait-free protocol implementation [12].

The algorithm makes use of three global sets of data. An array of buffers is sized so that there is always an available buffer for the writer to write new data. An array keeps in the i-th position the buffer index in use by the i-th reader. Finally, a variable keeps track of the latest buffer entry that has been updated by the writer. Each reader looks for the latest entry updated by the writer, stores its index in a local variable, and then reads its contents.

Consistency in the assignment of the buffer indexes can be guaranteed by any type of hardware support for atomic operations, including Compare-And-Swap (CAS) (as in the original paper by Chen and Burns [12]), but also Test and Set, and hardware semaphores, as available in several architectures today (the required support for atomicity is discussed in detail in [19]). The use of hardware support for atomicity is not limited to wait-free methods. Any implementation of MSRP needs similar instructions for the low-level protection of the lock data structures (including the FIFO queue for tasks waiting for the shared resource).

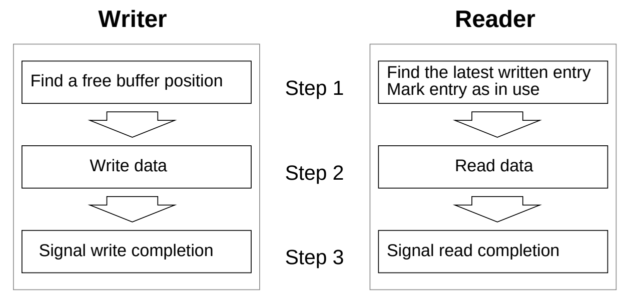

Fig. 1: Writer and readers stages in wait-free methods.

The implementation of a wait-free mechanism is possible in constant time (independent from the number n_r of the readers, as opposed to the $O(n_r)$ solution in [12]) using the Temporal Concurrency Control Protocol (TCCP) discussed in [26]. It does not need *additional blocking times* and has limited overhead (as discussed in Section III-C). However, it requires an array of buffers, thus *additional memory*, compared to lock-based methods. The number of required buffers is

$$\max_{j \in \mathbb{R}} \left\lceil \frac{l_j}{T_w} \right\rceil \tag{7}$$

where $\mathbb{R}$ is the set of readers, T_w is the period of the writer, and l_j is the lifetime of reader j (the sum of its worst-case response time R_j and the largest offset $O_{w,j}$ between the activation of reader j and the previous activation of the writer).

C. Wait-free vs. Lock based, limitation and performance issues

Wait-free methods are not meant to be a simple replacement for lock-based methods. They can only be used for memory resources (communication or state buffers). When the access to the shared resource cannot be simply isolated as a set of reads, such as a counter increment, wait-free methods still apply, upon the condition that the task performing the operation reads the content from a buffer item and writes its updated content onto a different buffer. A counter increment (like the statement `x = x+1`) wrapped inside a lock is a simpler solution, which requires a time overhead of a single lock- operation, smaller than the sum of the overheads for a read and a write if these two operations are protected separately. However, the cases that affect more the worst-case performance are those that require large access times, where a read-then-write pattern is less frequent.

In addition, even though the wait-free implementation requires multiple copies of the buffer, it does not require additional copy operations. Simply, instead of overwriting the same memory section, the writer writes its updates on a different memory buffer every time (in cyclic fashion). Also, readers get a reference to the buffer item they are allowed to use (the freshest one), which is guaranteed not to be overwritten by the writer, and therefore can use the data already in place, *without the need of additional copies*.

To find the associated time overhead, a measurement-based comparison of wait-free (TCCP) and lock-based (MPCP, MSRP) implementations on the Erika open source OSEK OS, running on a Freescale FADO processor (a heterogeneous dual-core) has been performed [18]. The overhead of TCCP is no larger than 43 CPU cycles, or 0.36 microsecond if the CPU frequency is at 120MHz, including the time to execute the hardware semaphore instruction used to achieve atomicity in the implementation. In all the measures, TCCP has an overhead no larger than one third of MSRP (in MSRP, to set up the data structures for acquiring or releasing the lock). Of course this does not related to the blocking time, which is the time spent *after the lock acquisition and before its release*.

Hence, with a limited penalty for the wait-free implementation, we assume that overheads even out or are negligible for both wait-free and MSRP when compared to task executions and resource access times. The task response time while using wait-free methods can be computed using the general formula (2) by setting $C_i^* = C_i$, $B_i^l = 0$, and $B_i^r = 0$, as there is no local or global blocking.

IV. CASR: Task Placement with Affinity

The Greedy Slacker (GS) algorithm [27] presents a method to obtain a resource-aware assignment of tasks to cores. It is based on a simple rule: tasks are ordered by decreasing task density (C_i/D_i) and assigned sequentially to cores. At each step, the candidate task is assigned to the core that guarantees the maximum value of the minimum slack among the tasks already allocated. To determine the minimum slack assuming a task is assigned to a specific core, GS uses a modified version of Audsley's optimal priority assignment scheme [2]. The main problem with GS is that slacks cannot be computed unless all tasks are allocated. Therefore, at each stage, the task slacks can only be estimated by using the subset of tasks already allocated. In addition, GS does not include recursion or retries. If at any point, a task fails to be assigned to any core, no alternative solutions are tried and the algorithm returns failure.

Based on the above observations, we propose a new heuristic, the Communication Affinity and Slack with Retries (CASR), listed in Algorithm 1. It has the following features:
– CASR tries to assign tasks sharing the same resource on the same core by considering their communication affinity.
– When a task is unschedulable on any core, CASR has a recovery mechanism that de-allocates tasks and retries.

A task τ_i is defined to be **affine** to another task τ_j if the two share some common resource. A core p is **affine** to τ_i if at least one task affine to τ_i has been assigned to p. Task affinity is not transitive. For example, if τ_i uses $\{r_1, r_2\}$, τ_j uses $\{r_2, r_3\}$ and τ_k uses $\{r_3, r_4\}$, then τ_i and τ_k are both affine to τ_j. However, τ_i and τ_k are not affine to each other.

CASR has a main loop in which tasks are considered for allocation in order of their density. For each task τ_i in the set to be allocated $\mathcal{NA}$, CASR tries to allocate τ_i on its affine cores first. Intuitively, this is an attempt at minimizing the global blocking time. However, without further constraints, a core could be overloaded by a large cluster of affine tasks and the partitioning procedure results in failure. To prevent such a scenario, an affine core p can only be added to the set of candidate cores $\mathcal{Q}$ considered for the allocation of the task τ_i if

Algorithm 1: Communication Affinity and Slack with Retries

1: $\mathcal{NA} \leftarrow \mathcal{T}$; aff_chk=true; $\mathcal{BL} \leftarrow \{\}$; $\mathcal{PBL} \leftarrow \{\}$;
2: **while** $\mathcal{NA} \neq \varnothing$ **do**
3: $\quad$ $\tau_i \leftarrow$ HighestDensity($\mathcal{NA}$); $\mathcal{Q} \leftarrow \varnothing$
4: $\quad$ **if** aff_chk **then**
5: $\qquad$ **for all** $p \in \mathcal{P}$ **do**
6: $\qquad\quad$ **if** Affine(p, τ_i) and $U(p) \leq U_b$ **then**
7: $\qquad\qquad$ insert p in $\mathcal{Q}$
8: $\qquad\quad$ **end if**
9: $\qquad$ **end for**
10: $\quad$ **end if**
11: $\quad$ **if** $\mathcal{Q} = \varnothing$ **then** $\mathcal{Q} = \mathcal{P}$
12: $\quad$ $C \leftarrow \varnothing$
13: $\quad$ **for all** $p \in \mathcal{Q}$ **do**
14: $\qquad$ **if** tryAssign(τ_i, p) **then** $C \leftarrow (s, p)$
15: $\quad$ **end for**
16: $\quad$ **if** $C \neq \varnothing$ **then**
17: $\qquad$ $p =$ FindMaxS(C)
18: $\qquad$ Allocate(τ_i, p); $\mathcal{NA} = \mathcal{NA} \setminus \{\tau_i\}$
19: $\qquad$ **continue**
20: $\quad$ **end if**
21: $\quad$ **if** $\tau_i \in \mathcal{PBL}$ **then return** failure
22: $\quad$ **if** $\tau_i \in \mathcal{BL}$ **then**
23: $\qquad$ $\mathcal{PBL} = \mathcal{PBL} \cup \tau_i$
24: $\qquad$ aff_chk = False
25: $\quad$ **else**
26: $\qquad$ $\mathcal{BL} = \mathcal{BL} \cup \tau_i$
27: $\quad$ **end if**
28: $\quad$ $\mathcal{D}$ =AffineSet(τ_i);
29: $\quad$ **for all** $\tau_k \in \mathcal{D}$ **do** DeAllocate(τ_k); $\mathcal{NA} = \mathcal{NA} \cup \{\tau_k\}$
30: **end while**

its total utilization $U(p)$ is lower than a predefined utilization bound (U_b) (lines 5–9). If there is no available affine core, e.g. they are all overloaded, CASR simply considers all cores as candidates to host τ_i (line 11).

U_b is a tunable parameter for the algorithm. Unfortunately, the algorithm is sensitive to the value of U_b and there is no single value that performs best in all our experiments. A value that performed very well on average and provides for a good baseline is $U_b = U_{\mathcal{T}}/m$, where $U_{\mathcal{T}}$ is the total task utilization, with the obvious intuitive meaning of trying to achieve load balancing. In our experiments (Section VII) this value performs consistently better than GS. However, given that the runtime of the algorithm allows for multiple executions in a reasonable time, it is possible to execute the algorithm with a set of different values for U_b (e.g., from 0 to 1 in steps of 0.25 or 0.1) and select the best outcome among them. This allows to further and sensibly improve upon the GS solution.

After finding the set of candidate cores $\mathcal{Q}$, the following steps (lines 12–20) are the same as in GS. The method tryAssign is iteratively called on each of these cores. A task is assigned to the core where tryAssign returns the maximum least normalized slack (s). Like [27], tryAssign uses the Audsley's algorithm [2] to find a priority order of the tasks.

The second salient feature of CASR with respect to GS is the recovery procedure that is invoked when no core is available for τ_i (in the algorithm, when the set C is empty). Every time the allocation of a task τ_i fails, CASR de-allocates all its affine tasks previously allocated to some core, and puts them back in the list $\mathcal{NA}$ (lines 28–29). This recovery procedure, however, is not used unconditionally. The first time a task fails to find a suitable core, it is put in the black-list $\mathcal{BL}$ (line 26); the second time it fails, the task is put in the post-black-list $\mathcal{PBL}$ (line 23). When the post-black-list $\mathcal{PBL}$ is used first, continuing to the assign a task to its affine cores will most likely lead to another failure. Hence, when a task enters the list $\mathcal{PBL}$, the CASR algorithm stops considering the affinity between cores and tasks, by setting the aff_chk flag to false (line 24). Only if the allocation of a task inside the $\mathcal{PBL}$ fails, the CASR task partitioning procedure returns failure (line 21).

V. Using Wait-free Methods to Increase Schedulability

As summarized in Section III, MSRP provides data consistency for shared buffers at the price of a blocking time and a negative impact on the schedulability of the system. The blocking time increases with the number and the length of the critical sections on global shared resources. Wait-free methods provide an alternative way to ensure consistent data access to shared resources. They have practically no blocking time, but this comes at the price of additional memory cost. We propose to use a combination of MSRP and wait-free methods, to leverage their complementary characteristics. We develop two algorithms for task partitioning and the selection of protection mechanisms in multicore systems. The first is a wait-free extension of the GS and CASR algorithms. The second is an entirely novel memory-aware partitioning heuristic.

A. Extending Greedy Slacker with Wait-free Methods

Our first algorithm (GS-WF) extends the greedy slacker heuristic by providing an additional recovery option. Once a task fails to be assigned to any core, either in the original GS or after the latest attempt from the $\mathcal{PBL}$ in CASR, GS-WF uses wait-free mechanisms for all the global resources accessed by the task and attempts to assign the task to each of the cores. If the use of wait-free methods makes the task schedulable on more than one core, GS-WF selects the one with the maximum value of the smallest normalized slack. If the task can not be assigned to any core even after using wait-free methods, the algorithm fails.

B. Memory-aware Partitioning Algorithm

Extending the greedy slacker algorithm with wait-free methods enhances schedulability. However, our extension only enhances the assignments that the GS or CASR defined in case of their failure. Given that the allocation decisions are inherently based on minimizing slack and do not consider the memory cost, the end result can be schedulable but quite inefficient in terms of memory usage. Hence, we propose the Memory-aware Partitioning Algorithm (MPA), which consists of two phases:

Phase 1: Finding an initial feasible solution (possibly of good quality, that is, low memory use);

Phase 2: Improving the solution by exploring other possible solutions using a local search.

In the first phase, the algorithm tries to obtain an initial schedulable solution. In order to maximize the probability of finding such a solution, all global resources are initially protected by wait-free methods. Local resources are managed using the SRP policy [3]. The second phase reduces the memory cost resulting from the use of wait-free methods using local search to selectively change the data consistency mechanism to MSRP. We first describe the concept of *assignment urgency*.

1) Assignment Urgency: In the proposed algorithm, tasks are allocated to cores in a sequential order. The order has a significant impact on the schedulability and the overall cost of the task allocation. We propose the concept of **Assignment Urgency (AU)**, an estimate of the penalty (in schedulability or memory) if a task is not the next to be assigned. The assignment urgency AU_i of task τ_i is defined as:

$$AU_i = \begin{cases} M_{max} & \tau_i \text{ schedulable on 1 core} \\ \left| \min_{1 \le j \le m} MC(\tau_i, p_j) - \min_{\substack{1 \le k \le m \\ k \ne j}} MC(\tau_i, p_k) \right| & \tau_i \text{ schedulable on } > 1 \text{ cores} \end{cases} \tag{8}$$

where $MC(\tau_i, p_j)$ is the memory cost of assigning τ_i to p_j using wait-free for all its global resources, and M_{max} is a value higher than the worst case memory cost of assigning any task to any core in the system.

$$M_{max} = \max_{\forall \tau_i \in \mathcal{T}} \max_{\forall p_j \in \mathcal{P}} MC(\tau_i, p_j) + 1 \tag{9}$$

Defining M_{max} in this way ensures that tasks schedulable on only one core will always have higher AU values than tasks with more assignment options. If a task can be scheduled on more than one core, then there is usually enough allocation freedom to defer its assignment. However, this might come with a cost in memory since the task may have to be scheduled on a different core that could not guarantee the lowest memory cost. The possible memory penalty is quantified by the (absolute) difference in memory between the core where it has best memory cost and the one with the second best.

The function `TaskSort()` in Algorithm 2 computes the assignment urgencies and sorts the unassigned tasks in the system by decreasing AU values. $\mathcal{LT}$ contains the list of tasks that have not been assigned yet. The feasible cores for a given unassigned task τ_i are stored in a list of candidate solutions $c\mathcal{P}$. Each entry in $c\mathcal{P}$ is a pair of core and cost *(p, c)*, where *c* is the memory cost of assigning τ_i to the core *p*, as computed by the function MemoryCost(). If there are no feasible solutions for a given task, the algorithm reports failure (line 10). Otherwise, the task assignment urgency is calculated according to Equation (8) (lines 12-17). The function also computes the best candidate core for a given task τ_i as BP_i (line 18). It then sorts the list $\mathcal{LT}$ by decreasing assignment urgency (line 20). In case of a tie in the AU values, the task with the highest density is assigned first.

The main algorithm (Algorithm 3) for task allocation and resource protection selection works in two phases.

2) Phase 1: The first phase is a greedy algorithm that assumes the use of wait-free methods for all global resources and tries to assign each task to the core on which it has the least memory cost. The algorithm uses the function `TaskSort()` to sort the set of unassigned tasks in the list $\mathcal{LT}$ by their AUs, and compute the best core for the allocation of each task (where it causes the least memory cost). The task with the highest assignment urgency is then assigned to its best candidate core (lines 13–14). At any time, if a task has no feasible core, the algorithm resets the task assignments and tries a resource-oblivious any-fit policy which runs in order: worst-fit with decreasing utilization, best-fit with decreasing utilization, first-fit, and next-fit (lines 5–10). If any-fit also

Algorithm 2: Calculating assignment urgencies

1: **Function** `TaskSort`($\mathcal{LT}$)
2: **for all** τ_i in $\mathcal{LT}$ **do**
3: $c\mathcal{P} = \{\}$
4: **for all** p_j in $\mathcal{P}$ **do**
5: **if** isSchedulable(τ_i,p_j) **then**
6: $c\mathcal{P} \leftarrow (p_j$, MemoryCost($\tau_i$,$p_j$))
7: **end if**
8: **end for**
9: **if** $c\mathcal{P} = \varnothing$ **then**
10: **return failure**
11: **end if**
12: **if** size($c\mathcal{P}$) $= 1$ **then**
13: $AU_i = M_{max}$
14: **else**
15: sortByCost($c\mathcal{P}$)
16: $AU_i = c\mathcal{P}$[1].c-$c\mathcal{P}$[0].c
17: **end if**
18: $BP_i = c\mathcal{P}$[0].p
19: **end for**
20: sortByAU($\mathcal{LT}$)
21: **return success**

Algorithm 3: Memory-aware Placement Algorithm

1: **Function** `AllocationAndSynthesis`($\mathcal{T}$)
2: **Phase 1:**
3: $\mathcal{LT} \leftarrow \mathcal{T}$
4: **while** $\mathcal{LT} \neq \varnothing$ **do**
5: **if** `TaskSort`($\mathcal{LT}$) = failure **then**
6: Reset partitioning
7: **if** `anyFit()` = true **then**
8: Goto Phase 2
9: **else**
10: **return** failure
11: **end if**
12: **else**
13: τ_k = ExtractFirst($\mathcal{LT}$)
14: $TA \leftarrow$ Allocate(τ_k, BP_k)
15: **end if**
16: **end while**
17:
18: **Phase 2:**
19: `optimizeResources`(TA)
20: $CurOpt = TA$; $\mathcal{NA}.add(TA)$
21: **while** $\mathcal{NA} \neq \varnothing$ **do**
22: Th = MemoryCost(GetLast($\mathcal{NA}$))
23: $curSys$ = ExtractFirst($\mathcal{NA}$)
24: $\mathcal{LN}$ =`generateNeighbors`(curSys)
25: **for all** AS in $\mathcal{LN}$ **do**
26: `optimizeResources`(AS)
27: **if** IsSchedulable(AS) and MemoryCost(AS) $< Th$ and NotVisited(AS) **then**
28: $\mathcal{NA}.add(AS)$
29: **if** size($\mathcal{NA}$) $> n$ **then** RemoveLast($\mathcal{NA}$)
30: **if** cost(AS) $<$ cost($curOpt$) **then** $curOpt = AS$
31: Th = MemoryCost(GetLast($\mathcal{NA}$))
32: **end if**
33: **end for**
34: **if** NoChange($curOpt$, #iter) or cost($curOpt$) $\leq tgtCost$ **then**
35: **return** $CurOpt$
36: **end if**
37: **end while**

fails, the algorithm fails. At the end of the first phase, all tasks should be assigned to a core in a task allocation scheme TA.

3) Phase 2: In the second phase, the solution obtained in the first phase is improved with respect to its memory cost. This is done in an iterative manner by exploring selected neighbors (with small memory cost) of candidate solutions starting from the initial solution obtained at the end of phase 1. Phase 2 uses two sub-functions:
– `optimizeResources()`, which optimizes data consistency mechanisms for shared resources;
– `generateNeighbors()`, which generates neighboring solutions for a given task assignment.

Algorithm 4: Determining resource protection mechanisms

1: **Function** `optimizeResources`(TA)
2: $\mathcal{GR}$ = FindGlobalResources(TA)
3: **for all** r_i in $\mathcal{GR}$ **do**
4: setProtocol(r_i, `WAITFREE`)
5: **end for**
6: sortByMemoryCost($\mathcal{GR}$)
7: **for all** r_i in $\mathcal{GR}$ **do**
8: setProtocol(r_i, `MSRP`)
9: **if** isSchedulable(TA) = false **then**
10: setProtocol(r_i, `WAITFREE`)
11: **end if**
12: **end for**

Function `optimizeResources()` changes the protection mechanism of global resources from wait-free to MSRP for as many resources as possible, while retaining system schedulability. It is called at the start of phase 2 (line 19, Algorithm 3) and whenever a new candidate solution is found (line 26, Algorithm 3), to optimize its memory cost.

The determination of the optimal resource protection mechanism for each global shared resource would require an exhaustive analysis of all the possible configurations with complexity 2^r, where r is the number of global resources in the system, which could easily be impractical. Therefore, a heuristic procedure (detailed in Algorithm 4) is used. The procedure initializes the protection mechanism for all the global resources to wait-free and places them in a list $\mathcal{GR}$ (lines 2–5). It then sorts the resources in $\mathcal{GR}$ by decreasing memory cost of their wait-free implementation (line 6). The protection mechanism of the first resource in $\mathcal{GR}$ (the resource with highest cost) is changed to MSRP and the system schedulability is checked. If the system becomes unschedulable, the protection mechanism is reversed back to wait-free. The procedure iterates through all the resources in $\mathcal{GR}$, changing the protection mechanism from wait-free to MSRP whenever possible (lines 7–12).

The other function invoked in the main loop of phase 2 is `generateNeighbors()`. It generates neighboring solutions and places them in the list $\mathcal{LN}$, which will be evaluated in the context of the local search for improving the current solution. A *neighbor* of a given task allocation solution can be obtained by a re-assignment of a task τ_i allocated on core P_a to a different core $P_b \neq P_a$ that can accommodate it, either directly (1-move neighbor) or by removing a task τ_j with equal or higher utilization from P_b and assigning τ_j to another core $P_c \neq P_b$ (2-move neighbor).

The **main loop** of phase 2 (lines 21–37 of Algorithm 3) is similar to a branch-and-bound algorithm. It performs a best-first search among candidate solutions. Candidate solutions for exploration are placed in a list $\mathcal{NA}$ sorted by increasing memory cost of the solution. The first solution (one with lowest cost) in $\mathcal{NA}$ is then further explored by branching (generating its neighbors).

The difficulty in exploration arises from the estimate on the quality of the solutions that may be found under a given branch. The algorithm allows the exploration of solutions (neighbors) with both lower and higher costs than the current optimum to avoid getting stuck at a local optimum. However, to avoid infinite searches, the exploration is bounded by a condition on the number of iterations without improvement (line 34, Algorithm 3). It is also bounded by the size of

$\mathcal{NA}$ which is at most equal to the number n of tasks in the system (line 29, Algorithm 3). Essentially, the best n unexplored candidates at any stage are kept in $\mathcal{NA}$. In our experiments, larger sizes for $\mathcal{NA}$ such as $2n$ or $4n$ do not improve the quality of the obtained solution but result in substantially longer execution times. To avoid loops, recently visited neighbors are discarded.

The solution space exploration is depicted in lines 21–37 of Algorithm 3. The first solution in $\mathcal{NA}$ (solution with minimum cost) is removed from the list (line 23) and considered as the new base ($curSys$) for further exploration. Lines 24-33 show the generation and exploration of neighbors. All feasible neighbors of the current base $curSys$ are generated (line 24). However, not all of them are further explored. A threshold value Th is used as an acceptance criterion for new neighbors generated from $curSys$, which is set to be the memory cost of the last solution in $\mathcal{NA}$ (the unexplored solution with the n-th highest cost, as in line 31). Only solutions with lower costs than Th are accepted for further exploration (added to $\mathcal{NA}$, lines 27–28). The cost of any new solution is considered only after resource optimization (line 26), such as when comparing with Th (line 27), or when comparing with the current optimum $curOpt$ (line 30).

If there are no more promising solutions to explore ($\mathcal{NA}$ becomes empty), or the solution is not improving for a certain number of iterations, or a solution with the desired quality ($tgtCost$, typically depending on the available RAM memory) is found, the algorithm terminates (lines 34–35).

VI. An Illustrative Example

We provide a simple example to illustrate the operation of the GS, CASR, GS extended with wait-free (GS-WF), and MPA algorithms. The system consists of seven tasks to be partitioned on two cores, as shown in Table I. All critical sections have a WCET of 1ms, except that task τ_0 has a duration of the critical sections as 0.15ms. Task deadlines are equal to their periods. The task-to-core assignment during the execution of the algorithms is shown in Figure 2.

Task	Period (ms)	WCET (ms)	Readers	Comm. size (Bytes)
τ_0	10	1	1, 3	256
τ_1	100	8	0, 5	128
τ_2	400	117	4	48
τ_3	40	6	1, 6	128
τ_4	20	7	2	48
τ_5	1000	394	6	256
τ_6	20	7	1, 5	128

TABLE I: Task parameters for the example system

GS (Figure 2a) arranges tasks by density and assigns them to maximize the minimum slack. Task τ_5 has the highest density and is assigned first. Without loss of generality, it is allocated to p_1. GS then tries to assign task τ_4 to both cores. On p_1, the smallest normalized slack after assigning task τ_4 to p_1 is 0.389. Alternatively, assigning τ_4 to p_2 achieves a minimum normalized slack of 0.65. Hence, GS chooses to assign τ_4 to p_2. Following the same procedure, τ_6 is assigned to p_1 then τ_2, τ_3, and τ_0 are assigned to p_2. Greedy slacker then fails to assign task τ_1 to any core. At this point, the algorithm fails.

For CASR (Figure 2b), we assume $U_b = 0.858$ (the total task utilization divided by 2). CASR first assigns τ_5 and τ_4 to p_1 and p_2 respectively, similar to GS. Then, CASR allocates

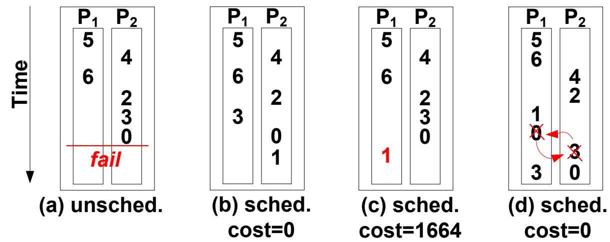

Fig. 2: Task assignment for the example system: (a) Greedy slacker, (b) CASR, (c) GS-WF, (d) MPA

τ_6 to its affine core p_1 and τ_2 to its affine core p_2. Next, CASR assigns τ_3 to p_1 because of the affinity, whereas GS places it on p_2 because of the slack-based rule. At this point, the total task utilization of p_1 exceeds U_b. Thus, when allocating the next task τ_0, p_1 loses its privilege as an affine core. CASR would try both p_1 and p_2, and allocates τ_0 to p_2 to maximize the least normalized slack. Finally, τ_1 is assigned to p_2, and the CASR succeeds in partitioning the task set.

GS-WF (Figure 2c) performs the first six task assignment (5, 4, 6, 2, 3, 0) in the same way as GS. The algorithm then attempts and fails to assign τ_1 using MSRP. At this point, a second attempt is made to assign τ_1 using wait-free for all the global resources used by τ_1. This attempts succeeds with a memory penalty of 1664 bytes. The memory cost is calculated according to Equation (7), leveraging information about task parameters to reduce the buffer sizes.

The MPA algorithm (Figure 2d) starts by calculating the initial assignment urgencies according to Equation (8). Tasks are then sorted by their assignment urgency values. Since no task has been assigned, all assignment urgencies will be equal to zero. The task density is used as a tie-breaker and τ_5 is selected as the first task to be assigned. Once τ_5 is assigned, the remaining tasks are all schedulable on both cores, therefore their assignment urgencies depend solely on the memory costs resulting from sharing resources with τ_5. Assigning any task to core p_1 has no memory cost since all shared resources defined until this stage will be local and managed using SRP. However, the costs of assigning tasks to p_2 will differ among tasks. The highest memory cost comes from τ_6 (cost is 1152) and hence it has the highest assignment urgency of 1152 (by Equation (8)). τ_6 will then be assigned to the core where it causes the least memory cost (p_1).

Next, the check on schedulability, performed by `TaskSort()`, reveals that tasks τ_4 and τ_2 are now unschedulable on p_1. Since these tasks can only be assigned to one core, their assignment urgencies becomes M_{max} (according to Equation (8)). All other tasks are schedulable on both cores and hence have lower AU values. Task τ_4 is now at the top of the list $\mathcal{LT}$ (using its density to break the tie with task τ_2) and is assigned immediately to core p_2 followed by task τ_2. Phase 1 of the algorithm continues in a similar way, until the last task τ_3 is assigned. All global resources use wait-free methods as the data consistency mechanism, and the cost of this initial solution is 1280 bytes.

Phase 2 of MPA starts by running the function `optimizeResources()`, which reduces the memory cost to 768 bytes by changing the protection mechanism for the resource written by τ_3 to MSRP. This solution is the current optimal solution ($CurOpt$ in line 20 of Algorithm 3). Phase

2 then generates and examines neighbors to this solution. One of the feasible neighbors can be obtained by swapping tasks 0 and 3. This proves to be an effective move as `optimizeResources()` is able to reduce the memory cost to 0. Since this is an optimal solution in terms of memory, the algorithm terminates.

VII. Experimental Results

In this section, the proposed algorithms are evaluated in terms of schedulability and memory cost (if applicable). In Section VII-A, we first compare the schedulability analysis from [17] (Equations (2)–(6)) and the ILP-based approach [28] using relatively small task sets. The ILP-based analysis [28] achieves better results, but is significantly slower and does not scale to large systems. We then provide an overall evaluation in Section VII-B comparing the GS with the CASR, GS-WF and MPA in terms of schedulability. We also compare the memory costs of GS-WF and MPA. In Section VII-C, we study the effect of other system parameters. Finally, in Section VII-D, we compare MPA with an exhaustive search to evaluate its ability to find solutions with reduced memory cost.

We adopt a task generation scheme similar to [27]. We consider systems with 3, 4 or 8 cores. The task periods are generated according to a log-uniform distribution from one of two different ranges [10, 100] ms and [3-33] ms. The task utilization is selected from the set {0.05, 0.1, 0.12, 0.2, 0.3}. The worst case execution time is then derived from the period and utilization. The critical section lengths are randomly generated in either [0.001, 0.1] ms or [0.001, 0.015] ms.

The tasks in the system share a number of resources between 1 and 40. The *resource sharing factor* represents the portion of tasks in the system sharing a given resource. A resource sharing factor of 0.1 means that each resource is shared by 10% of the tasks in the system. For each experiment, a resource sharing factor from the set {0.1, 0.25, 0.5, 0.75} is selected. The tasks that share a given resource are randomly selected and are independently generated from other resources. For each parameter configuration, 100 systems are randomly generated. The size of communication data was chosen among a set of possible values (in bytes): 1 (with probability p =10%), 4 (p =20%), 24 (p =20%), 48 (p =10%),128 (p =20%), 256 (p =10%), and 512 (p =10%).

For the CASR, we evaluate two approaches: 1) a single run of CASR with the utilization bound $U_b = U_{\mathcal{T}}/m$ (single Ub) and 2) the best solution from multiple (five) runs of CASR with a set of U_b values $\{0, 25\%, 50\%, 75\%, 100\%\}$ (multiple Ub). For MPA, the number of iterations is set to be $10n$ where n is the number of tasks in the system.

A. Schedulability Analysis

We first evaluate the task partitioning algorithm using the approximate schedulability analysis in [17], and the ILP-analysis in [28]. The average core utilization is selected to be in the range 68%-96%, by fixing the average task utilization at 12% and varying the number of tasks to be scheduled on 3 cores in the range [17, 24]. Tasks share 20 resources with a resource sharing factor of 0.25. The periods of the tasks are randomly generated in [10, 100] ms and the critical section lengths are randomly generated in [0.001, 0.1] ms. Table II reports the percentage of schedulable solutions obtained by each algorithm and the memory costs (for MPA and GS-WF). Table III reports the average runtimes of the algorithms. Each table is divided into two parts where the upper half shows the results using the approximate analysis, and the lower half (with the prefix i) shows the ILP-based analysis results.

	U = 68%	U = 80%	U = 88%	U = 92%	U = 96%
GS	100%	46%	0%	0%	0%
CASR (single Ub)	100%	69%	0%	0%	0%
CASR (multiple Ub)	100%	91%	1%	0%	0%
GS-WF	100%/0	100%/1687	26%/6963	0%/–	0%/–
MPA	100%/0	100%/0	96%/231	37%/3315	0%/–
iGS	100%	99%	70%	9%	0%
iCASR (single Ub)	100%	100%	85%	21%	0%
iCASR (multiple Ub)	100%	100%	98%	37%	0%
iGS-WF	100%/0	100%/31	81%/347	13%/1395	0%/–
iMPA	100%/0	100%/0	96%/0	37%/79	0%/–

TABLE II: Schedulability/average memory cost (GS-WF and MPA only, in bytes) for the algorithms

	U = 68%	U = 80%	U = 88%	U = 92%	U = 96%
GS	0.022	0.026	0.011	0.007	0.007
CASR (single Ub)	0.016	0.044	0.035	0.002	0.001
CASR (multiple Ub)	0.019	0.073	0.159	0.009	0.007
GS-WF	0.020	0.032	0.021	0.009	0.008
MPA	0.077	0.183	13.2	3.7	0.022
iGS	65.31	86.06	101.99	49.97	20.16
iCASR (single Ub)	28.49	70.53	122.91	132.74	51.7
iCASR (multiple Ub)	41.54	86.6	164.29	398.2	253.22
iGS-WF	47.31	88.67	104.65	57.83	26.13
iMPA	198.62	247.73	133.43	52.96	17.35

TABLE III: Average runtimes (in seconds) for the algorithms

The general trend is that CASR with multiple utilization bounds and MPA outperform the other algorithms. The performance of all partitioning algorithms improves in terms of both schedulability and memory cost when using the ILP test. However, this comes at the price of a larger runtime. The runtimes for relatively small systems increase by 150 times on average when using the ILP analysis, and even more for larger systems. In general, the relative comparison among the partitioning schemes does not appear to be sensitive to the choice of the analysis method. In the rest of the experiments, we use the analysis in [17] for larger systems.

B. General Evaluation

The second set of experiments is performed on systems with a (higher) number of tasks n in the range [40, 76], scheduled on 4 and 8 cores. The average utilization of each task is 0.1. The periods are generated in the range [10, 100] ms, and critical section lengths are selected from the range [0.001, 0.1] ms. The number of resources shared among tasks is fixed at 4. Each resource is shared by a quarter of the tasks in the system.

The results are shown in Table IV and Figures 3 and 4 for the case with 8 cores. In our experiments, CASR performs better with respect to schedulability. Table IV shows the effectiveness of the recovery strategy. For each U_b value, the table shows the results of CASR for all the generated systems, as the percentage of systems that are found schedulable after the first and the second recovery stage (over all the schedulable ones). In each table entry, the first number is the percentage of systems successfully partitioned (schedulable) only after

#tasks	$U_b = 0$	$U_b = 25\%$	$U_b = 50\%$	$U_b = 75\%$	$U_b = 100\%$
40	$(0, 0)$	$(0, 0)$	$(0, 0)$	$(3, 2)$	$(27, 3)$
43	$(0, 0)$	$(0, 0)$	$(0, 0)$	$(3, 2)$	$(24, 5)$
46	$(0, 0)$	$(0, 0)$	$(0, 1)$	$(7, 10)$	$(30, 4)$
50	$(0, 0)$	$(0, 0)$	$(0, 0)$	$(9, 5)$	$(35, 6)$
53	$(0, 1)$	$(0, 1)$	$(0, 1)$	$(7, 12)$	$(36, 7)$
56	$(0, 4)$	$(0, 5)$	$(1, 5)$	$(17, 8)$	$(40, 7)$
60	$(1, 11)$	$(3, 13)$	$(0, 19)$	$(19, 15)$	$(37, 11)$
63	$(10, 31)$	$(8, 30)$	$(4, 30)$	$(38, 21)$	$(53, 17)$
66	$(0, 0)$	$(0, 0)$	$(25, 25)$	$(30, 20)$	$(75, 25)$

TABLE IV: Percentage of systems recovered by $\mathcal{PBL}$ and $\mathcal{BL}$ in CASR among the schedulable ones (8 cores, 4 resources)

using the post-black-list $\mathcal{PBL}$ and the second number is the percentage of configurations found only after using the $\mathcal{BL}$ list and with empty $\mathcal{PBL}$. For systems with more than 68 tasks, CASR (and the other policies) do not find any feasible solution. The recovery strategy is more effective when the utilization bound is larger or the system utilization is higher.

Figure 3 also shows the improvement obtained using wait-free methods. The GS-WF algorithm, as expected, contributes to a significant increase in the schedulability of GS. MPA performs even better, scheduling more systems at high utilizations. Figure 4 compares the additional memory cost needed by wait-free methods for both GS-WF and MPA. It shows that for systems with a small number of tasks, GS-WF performs slightly better. However, as the number of tasks (and hence utilization) increases, MPA outperforms GS-WF. For systems with 60 tasks (or an average core utilization of about 0.75), these algorithms perform approximately equally; at higher utilization, MPA tends to have a lower memory cost.

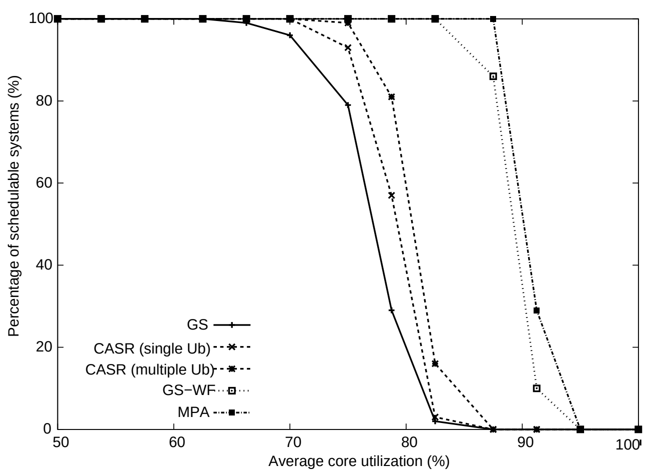

Fig. 3: Comparison of schedulability (8 cores, 4 resources)

C. Effect of Other Parameters

To evaluate systems in which tasks share a large number of resources we generated system configurations with 4 cores and 20 resources and a number of tasks in the range [20, 40]. All other parameter settings are similar to those in Section VII-B. Figures 5 and 6 plot the results of this experiment. The general trend in schedulability remains the same, with CASR performing better than GS while MPA significantly improving upon GS-WF in both schedulability and memory cost.

Next, we fix the number of tasks at 28 and try a variable number of resources in the range [1, 40]. The schedulability

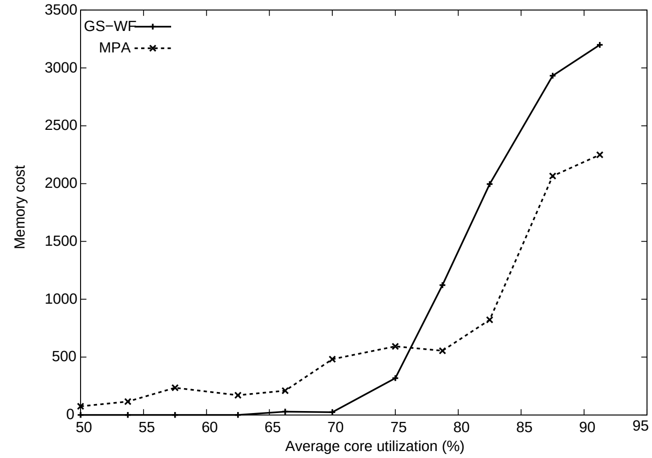

Fig. 4: Comparison of memory cost (8 cores, 4 resources)

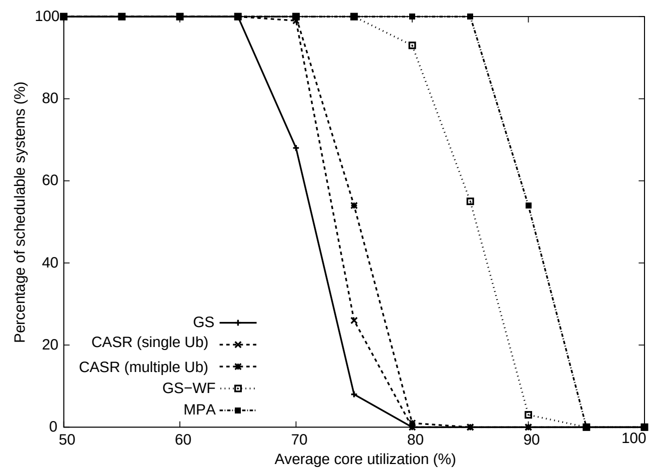

Fig. 5: Comparison of schedulability (4 cores, 20 resources)

results are shown in Figure 7. Both GS-WF and MPA always schedule all tasks and are thus omitted from the figure.

To evaluate the effect of changing the resource sharing factor, another experiment is performed using the same parameter settings while keeping the number of resources at 20 and changing the resource sharing factor (rsf). Table V

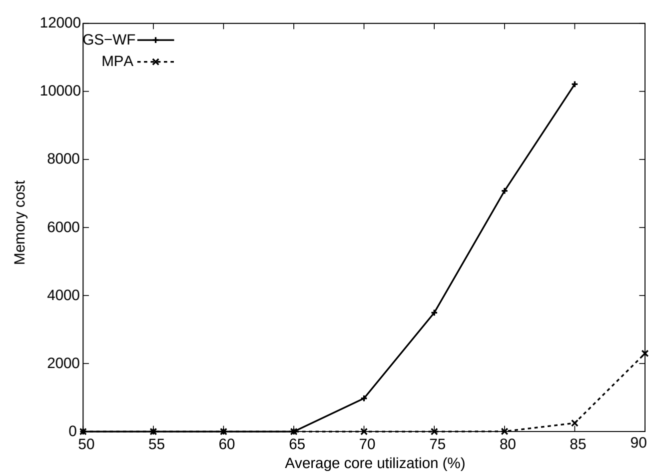

Fig. 6: Comparison of memory cost (4 cores, 20 resources)

	rsf=0.1	rsf=0.25	rsf=0.5	rsf=0.75
GS	100%	68%	0%	0%
CASR (single Ub)	100%	94%	0%	0%
CASR (multiple Ub)	100%	100%	0%	0%
GS-WF	100%/0	100%/1136	98%/9628	67%/11493
MPA	100%/0	100%/0	100%/104	100%/841

TABLE V: Schedulability and average memory cost (GS-WF and MPA only, in bytes) for different resource uses

presents the result of this experiment. It shows that CASR schedules more tasks than GS. As communication increases, CASR outperforms GS. When the use of wait-free resources is possible, the MPA algorithm performs significantly better than both GS and GS-WF as tasks communicate more often. MPA also succeeds in keeping the memory usage relatively small, requiring just 841 bytes of memory at rsf=0.75 compared to 11934 bytes for GS-WF.

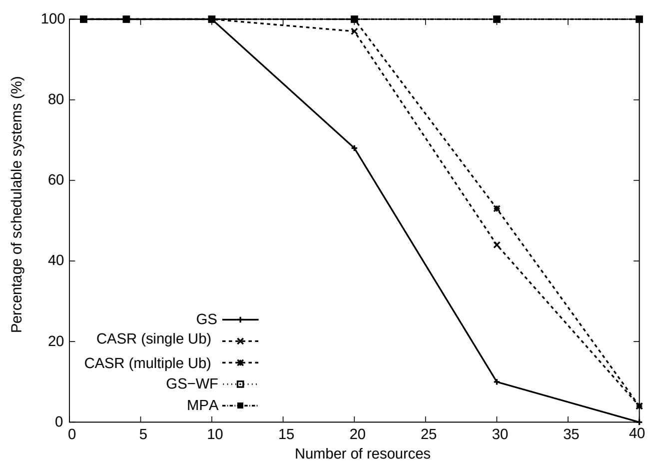

Fig. 7: Comparison of schedulability with a variable number of resources

D. MPA vs. Exhaustive Search

Finally, we tried an exhaustive search to validate the quality of the proposed MPA heuristic. Exhaustive search can be time consuming, as task partitioning is an NP-hard bin packing problem. Therefore, we restrict our comparison to small systems with 12–16 tasks and 3 cores. To compare the effectiveness of the MPA algorithm in terms of memory cost, we computed the results of exhaustive search for partitioning 100 systems at a utilization of 80%. The average memory cost was 19% worse for MPA. In terms of schedulability, MPA successfully scheduled all 100 systems. However, MPA runs significantly faster than the exhaustive search: the average runtime for MPA is about 6.3 seconds, while the exhaustive search takes about 2988 seconds on average.

VIII. Conclusion

In this paper, we provide two algorithms that improve on existing task placement algorithms for systems using MSRP to protect shared resources. Furthermore, we leverage an additional opportunity provided by wait-free methods as an alternative mechanism to protect data consistency for shared buffers. The selective use of wait-free methods significantly extends the range of schedulable systems at the cost of memory, as shown by experiments on random task sets.

Acknowledgements

We would like to thank Alexander Wieder and Björn Brandenburg for providing their code on Greedy Slacker.

References

[1] The AUTOSAR consortium. *The AUTOSAR Standard, specification version 4.1*. [Online] http://www.autosar.org.
[2] N. Audsley. "On priority assignment in fixed priority scheduling." *Information Processing Letters*, 79(1):39–44, 2001.
[3] T. Baker. "A stack-based resource allocation policy for realtime processes." In *Proc. 11th IEEE Real-Time Systems Symposium*, 1990.
[4] S. Baruah. "The partitioned EDF scheduling of sporadic task systems." In *Proc. IEEE Real-Time Systems Symposium*, 2011.
[5] S. Baruah and E. Bini. "Partitioned scheduling of sporadic task systems: An ILP based approach." In *Proc. Conf. Design and Architectures for Signal and Image Processing*, 2008.
[6] S. Baruah and N. Fisher. "The partitioned multiprocessor scheduling of sporadic task systems." In *Proc. 26th IEEE Real-Time Systems Symposium*, 2005.
[7] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" In *Real-Time & Embedded Technology & Applications Symposium*, 2008.
[8] B. Brandenburg. "A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications." In *Proc. 25th Euromicro Conference on Real-Time Systems*, 2013.
[9] B. Brandenburg. "The FMLP+: An Asymptotically Optimal Real-Time Locking Protocol for Suspension-Aware Analysis." In *Proc. 26th Euromicro Conference on Real-Time Systems*, 2014.
[10] E. Bini and G. Buttazzo. "Measuring the Performance of Schedulability Tests." *Real-Time Systems*, 30(1-2):129–154, May 2005.
[11] B. Chattopadhyay and S. Baruah. "A lookup-table driven approach to partitioned scheduling." In *Proc. 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.
[12] 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.
[13] J. Chen and A. Burns. "Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties." In *Proc. 6th Int. Conference on Real-Time Computing Systems and Applications*, 1999.
[14] R. Davis and A. Burns. "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems." *ACM Computing Surveys*, 43(4), Oct. 2011.
[15] S. Dhall and C. Liu. "On a real-time scheduling problem." *Operations Research*, 26(1):127–140, 1978.
[16] N. Fisher and S. Baruah. "The Partitioned Scheduling of Sporadic Tasks According to Static Priorities." In *Proc. 18th Euromicro Conf. Real-Time Systems*, 2006.
[17] P. Gai, G. Lipari, and M. D. 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.
[18] G. Han, H. Zeng, M. Di Natale, X. Liu, and W. Dou. "Experimental Evaluation and Selection of Data Communication Mechanisms in Multicore Platforms." *IEEE Transactions on Industrial Informatics*, 10(2):903–918, May 2014.
[19] M. Herlihy. A methodology for implementing highly concurrent structures. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, 1990.
[20] H. Huang, P. Pillai, and K. G. Shin. "Improving wait-free algorithms for interprocess communication in embedded real-time systems." In *Proc. USENIX Annual Technical Conference*, 2002.
[21] D. Johnson. "Near-optimal bin packing algorithms." *Ph.D. dissertation, Massachusetts Institute of Technology*, 1973.
[22] 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.
[23] F. Nemati, T. Nolte, and M. Behnam. "Partitioning real-time systems on multiprocessors with shared resources." In *Proc. Conference on Principles of distributed systems*, 2010.
[24] R. Rajkumar. "Real-time synchronization protocols for shared memory multiprocessors." In *Proc. 10th International Conf. Distributed Computing Systems*, 1990.
[25] C. Sofronis, S. Tripakis, and P. Caspi. "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling." In *Proc. International conference on Embedded software*, 2006.
[26] 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.
[27] A. Wieder and B. Brandenburg. "Efficient Partitioning of Sporadic Real-Time Tasks with Shared Resources and Spin Locks." In *Proc. 8th IEEE International Symposium on Industrial Embedded Systems*, 2013.
[28] A. Wieder and B. Brandenburg. "On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks." In *Proc. 34th IEEE Real-Time Systems Symposium*, 2013.
[29] H. Zeng and M. Di Natale. "Mechanisms for Guaranteeing Data Consistency and Time Determinism in AUTOSAR Software on Multi-core Platforms." In *6th IEEE Symposium on Industrial Embedded Systems*, 2011.