Leibniz Transactions on Embedded Systems

Volume 4 | Issue 2 | January 2018

Vol. 4, Issue 2 ISSN 2199-2002 http://www.dagstuhl.de/lites

ISSN 2199-2002

Published online and open access by
the European Design and Automation Association (EDAA) / EMbedded Systems Special Interest Group (EMSIG) and Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.
Online available at
http://www.dagstuhl.de/dagpub/2199-2002.

Publication date
January 2018

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier
10.4230/LITES-v004-i002

Aims and Scope
LITES aims at the publication of high-quality scholarly articles, ensuring efficient submission, reviewing, and publishing procedures. All articles are published open access, i.e., accessible online without any costs. The rights are retained by the author(s).

LITES publishes original articles on all aspects of embedded computer systems, in particular: the design, the implementation, the verification, and the testing of embedded hardware and software systems; the theoretical foundations; single-core, multi-processor, and networked architectures and their energy consumption and predictability properties; reliability and fault tolerance; security properties; and on applications in the avionics, the automotive, the telecommunication, the medical, and the production domains.

Contact
Schloss Dagstuhl – Leibniz-Zentrum für Informatik
LITES, Editorial Office
Oktavie-Allee, 66687 Wadern, Germany
lites@dagstuhl.de
http://www.dagstuhl.de/lites

Contents

Regular Papers

Dynamic and Static Task Allocation for Hard Real-Time Video Stream Decoding on NoCs*

Hashan R. Mendis[1], Neil C. Audsley[2], and Leandro Soares Indrusiak[3]

1 **Real-Time Systems Group, Department of Computer Science, University of York, UK**
`hrm506@york.ac.uk`
2 **Real-Time Systems Group, Department of Computer Science, University of York, UK**
`neil.audsley@york.ac.uk`
3 **Real-Time Systems Group, Department of Computer Science, University of York, UK**
`http://orcid.org/0000-0002-9938-2920`
`leandro.indrusiak@york.ac.uk`

Abstract

Hard real-time (HRT) video systems require admission control decisions that rely on two factors. Firstly, schedulability analysis of the data-dependent, communicating tasks within the application need to be carried out in order to guarantee timing and predictability. Secondly, the allocation of the tasks to multi-core processing elements would generate different results in the schedulability analysis. Due to the conservative nature of the state-of-the-art schedulability analysis of tasks and message flows, and the unpredictability in the application, the system resources are often under-utilised. In this paper we propose two blocking-aware dynamic task allocation techniques that exploit application and platform characteristics, in order to increase the number of simultaneous, fully schedulable, video streams handled by the system. A novel, worst-case response time aware, search-based, static hard real-time task mapper is introduced to act as an upper-baseline to the proposed techniques. Further evaluations are carried out against existing heuristic-based dynamic mappers. Improvements to the admission rates and the system utilisation under a range of different workloads and platform sizes are explored.

2012 ACM Subject Classification On-chip resource management
Keywords and Phrases real-time multimedia, task mapping, network-on-chip
Digital Object Identifier 10.4230/LITES-v004-i002-a001
Received 2016-06-13 **Accepted** 2017-04-03 **Published** 2017-07-07

1 Introduction

Current multiprocessor System-on-Chip (MPSoC) platforms (many-cores) have tens or hundreds of processing elements (PEs) and often need to support an increased number of applications simultaneously. Network-on-chip (NoC), interconnects have emerged as the promising solution for communication infrastructure of such systems, due to its efficiency and scalability [7]. Future technologies with streaming multimedia applications form a large portion of the application space that exploit these highly distributed on-chip architectures [33]. The processing load imposed by computation-intensive applications such as video decoding can be partitioned into tasks

* We would like to thank the LSCITS program (EP/F501374/1) and DreamCloud project (EU FP7-611411), for funding this research and RheonMedia Ltd. for providing industrial case studies.

and distributed among multiple PEs on the many-core platform, to improve metrics such as performance, utilisation, energy and to meet timing constraints.

This work looks at the resource-aware embedded systems design problem from the software perspective. Modern MPSoCs have many on-chip resources, such as PEs, communication channels and main memory controllers, which have to be allocated to applications to optimise on high-level metrics such as latency, utilisation and power consumption. Efficient task allocations can reduce NoC usage, and network contention; thereby reducing the response time of the task set and communication energy consumption. These allocations are generally performed at the software level, by a resource manager. On the other hand, from a hardware viewpoint, the NoC communication bandwidth can be reduced (e.g. reducing the link width or frequency) to maximise the NoC utilisation and save area and power consumption. However, this work focuses solely on software-based runtime task mapping optimisation, as it assumes the hardware implementation platform is fixed and it also offers a less costly and more flexible solution.

The nature of allocating and scheduling application tasks to PEs is considered an NP-hard problem. *Design-time (offline/static) mapping* techniques that have a global view of the system and workload, can be used to optimize the task mapping process at system design-time, such that system resources are efficiently utilised. The complexity and workload characteristics of video decoding applications depend greatly on the temporal and spatial variations in the video stream; hence, the workload is highly dynamic and unpredictable. When dealing with live video processing applications, certain critical application properties are unknown at design-time. Therefore, these highly varying workloads require *runtime (online/dynamic) mapping* techniques, based on light-weight heuristics to allocate and schedule the tasks to PEs whilst the system is operational. Live video decoding systems have hard timing constraints that need to be guaranteed before admitting into the system for decoding. These systems often use deterministic video admission control strategies, which use worst-case timing behaviour of the tasks, resulting in under-utilised systems [24]. In [23], the authors show that by employing efficient dynamic, application and platform aware task to PE mapping strategies, the utilisation levels of hard-real time video decoding systems could be improved. In this work, the mapping approaches in [23] are further evaluated by comparing against an offline mapper and under different platform and workload conditions.

Violation of timing constraints in video processing systems can lead to degraded quality, but the system will continue to operate; hence multimedia systems are generally considered soft real-time (SRT). However, there exists a range of systems that depend on video streams that need to be processed with hard real-time (HRT) guarantees. For example, in vision-based robot control systems, accuracy and functionality of the feedback control systems depend on processing video frames with tight timing restrictions. Another example is in the telesurgery/teleoperation industry [14], where a doctor performs surgery on a patient without physically being in the same location. These safety-critical systems require responsive and reliable communication technology as well as hard real-time guarantees from the video processing systems to function safely. Furthermore, next generation automated video surveillance systems, will require processing and tracking objects in hundreds of video streams in real-time; missing deadlines in these systems would lead to reduced security and delayed response to threats.

Contributions: In order to address the aforementioned issues, we present the following novel contributions:

- A more *precise* definition of the application model [24, 23] used to represent decoding of multiple MPEG-2 video streams is presented, including its *execution and communication characteristics* (Section 3.1). Very few other work consider data parallel video decoding using a scalable, distributed memory, message-passing based communication model.

- A response-time analysis (RTA) based, application-aware, *deterministic admission controller* (D-AC) is presented in [24]. This work (in Section 4) describes, the underlying *D-AC process algorithmically*. Formal definitions of taking into account task graph precedence constraints within the RTA is also presented.
- We present *new evaluations* (Section 7) of the two application-specific, blocking-aware heuristic based runtime mapping approaches introduced in [23]:
  - We present an *upper-baseline* for evaluation – a genetic algorithm (GA) based static/design-time task mapping optimisation approach (Section 6), with a *novel fitness function* that considers video stream schedulability. This design-time mapper is compared against our proposed dynamic mappers and other existing runtime mappers.
  - We present new experimental treatments: varying both the *platform size* and workload *computation-to-communication ratio*, which gives new insight to the strengths and weaknesses of each evaluated mapping approach.

The rest of this paper is organized as follows. Section 2 presents related work in task mapping and scheduling. Section 3 introduces the system models. Section 4 presents the deterministic admission controller. The proposed heuristic based runtime task mapping algorithms are described in Section 5, followed by the design-time static mapper in Section 6. Section 7 presents the experimental design and discusses the results. Section 8 concludes this paper.

2 Related Work

Mapping of tasks onto PEs broadly falls under two categories: *static* and *dynamic* mapping. Static mappers (executed at design time) have a complete view of the application, workload and platform and attempt to find a suitable task to processor placement to optimise for different metrics such as execution time, throughput, resource utilisation and energy consumption [29]. For example, Butazzo et al. [6] proposes a static mapper that uses a branch-and-bound algorithm to partition and map a taskset with precedence constraints, to reduce the computational resources. However, they assume negligible communication cost between the tasks. Simulated-annealing based, offline, task and memory mapping for mixed-criticality NoCs have been introduced by Giannopoulou et al. [13]. Their optimisation technique accounts for the interferences on the shared memory and the NoC, however they assume a time-triggered NoC with static routes regulated at the source, which is contrary to our priority-based wormhole switching NoC architecture. The static mapper presented in this work uses a RTA and points-based fitness function which evaluates the schedulability of a video stream.

Dynamic mappers (executed at runtime) use heuristics to optimise certain metrics such as application execution time, energy consumption, temperature, reliability and resource utilisation [29]. Carvalho et al. [8] exploit the hop-distance and path load between cores, as a dynamic mapping heuristic to reduce communication packet latency, energy and channel occupation. This work was later extended by Singh et al. [30] to include PEs that can accommodate multiple tasks. Kaushik et al. [21] adapts these heuristics to balance both computation and communication, by using a pre-processing stage to achieve a balanced and reduced task-graph. There also exist hybrid mapping approaches (e.g. [27]), where design-time computed mapping templates are merged with runtime heuristic based decisions to reduce average power dissipation. This work focuses on dynamic mapping strategies that exploit both the application and architecture characteristics.

Ditze et al. [9] presents an extension to the least-laxity-first scheduling algorithm to schedule the MPEG decoding tasks and an admission controller that enforces QoS constraints of parallel video streams. However, their feedback based admission controller does not fully guarantee the schedulability of admitted video streams, but attempts to reduce over-reservation of system

resources. Bamakhrama et al. [4] presents an analytical framework to determine the minimum number of processors required to schedule a set of streaming applications with given I/O rates, while guaranteeing the maximum achievable throughput. Similarly, in [1] the critical-paths of task-graphs are mapped onto the PEs first, to reduce the average end-to-end worst-case execution time of a set of streaming applications, such as MP3 and H.263 decoders. They use utilisation tests to determine mapping feasibility and consider the inter-core communication cost as a constant cost per hop on the 2D mesh interconnect. Contrarily, we model the interference patterns of the task communication message flows on a predictable, pre-emptive interconnect. Utilisation based schedulability tests, have been used to guarantee timing properties of streaming applications, when using a contention-free TDMA based communication interconnect [15]. However, inefficient resource reservation in TDMA interconnects lead to unused bandwidth and connection setup overheads limits scalability [12]. In recent work by Dziurzanski et al. [10], RTA based schedulability tests have been used to determine if a taskset is schedulable on a processor. RTA tests take into account the interference caused by higher priority tasks and flows. However, using RTA based tests online, is expensive and in [10], they perform approximate tests to reduce the overhead and use a feedback-based control-theoretic approach to reclaim slack in order to improve admission rates. Similarly, in [25], tasks have been mapped to PEs that have the highest amount of average slack, where the PE slack values are periodically monitored and sent to a global manager. They state that a trade-off has to be made between NoC communication load imposed by slack monitoring and the monitoring frequency which can affect mapping results.

Contrary to many of the discussed related work, in our study we model a homogeneous multicore platform connected via a scalable, network interconnect with priority-based arbitration (similar to QNoC [5]); which makes it easier to predict worst-case network contention scenarios. Furthermore, unlike in existing approaches we do not use any monitoring feedback at runtime, which results in no communication overhead in the resource management technique. To the best of our knowledge, the runtime NoC resource management techniques (i.e. runtime task mapping and admission control), proposed in this work are the only ones aimed at HRT video decoding, to consider both the interference caused by task blocking and to exploit known video stream properties.

3 System Model and Problem Formulation

3.1 Application model

This section outlines the multi-stream video decoding application model, with focus on how the abstract workload is generated. Frames in each video stream, can be of type: I (Intra), P (Predictive) or B (Bi-directional) encoded; i.e. *frame type* denoted $f_t = \{I, P, B\}$. We assume parallel decoding of *multiple* MPEG-2 decoded video streams, with a fixed, independent, group-of-pictures (GoP) structure of *IPBBPBBPBBBB* (decoding order). According to the MPEG-2 specification this 12 frame GoP structure is recommended to balance compression, facilitate reasonable random-access points in the stream and to manage error propagation [11]. Decoding an MPEG stream can be parallelised at different levels of granularity (GoP/frame/slice/macroblock-level) [22]. In our application model, we assume frame-level data parallel decoding, which does not involve stream instrumentation.

As shown in the system overview diagram in Figure 1a, the application model has a hierarchical structure. At the top most level are stream based workflows (W_i), each containing video streams VS_i with arbitrary number of N independent jobs (denoted by J_i). Each of the video streams will have varying resolutions ($res(VS_i)$ = frame height×frame width). A job, represents an MPEG GoP and are modelled as a directed acyclic graph (DAG) with a fixed dependency structure, as

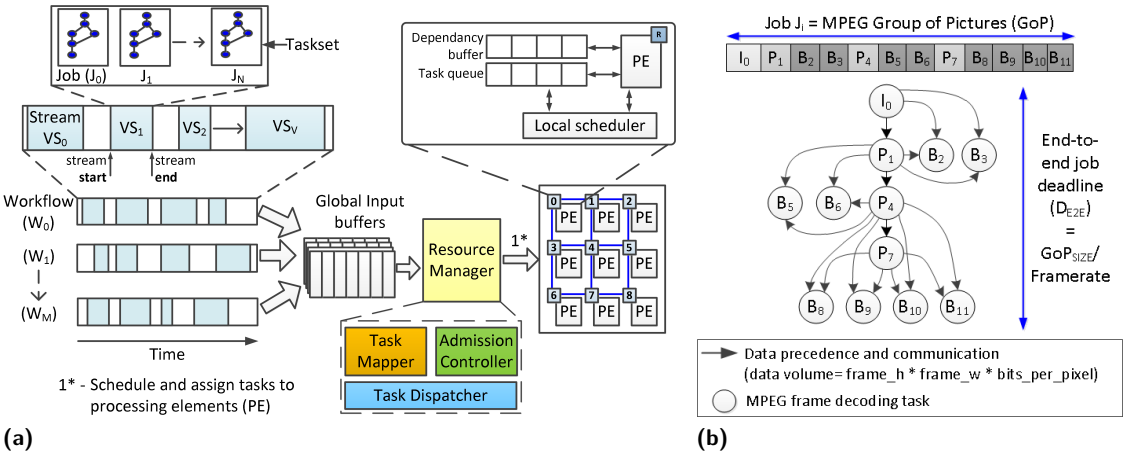

Figure 1 (a) System overview diagram; (b) MPEG GoP data precedence and task communication graph (Communication traffic between tasks and main-memory not illustrated).

depicted in Figure 1b. Each node in the task graph (TG) represents a real-time frame decoding task τ_i and edges represent traffic-flows (*flows* for short), denoted as Msg_i, which are reference data that needs to be sent to one or more dependent tasks (also referred to as *child* tasks). A task's execution can only start *iff* its predecessor(s) (also referred to as *parent* tasks) have completed execution and their output data is available. As shown in Figure 1b, certain tasks in the TG can be executed in parallel (e.g. P_4, B_2, B_3) if all the precedence constraints are met.

A task τ_i is characterised by the following tuple: (p_i, t_i, x_i, c_i, a_i); where p_i is the fixed priority, t_i is the period, x_i is the actual computation cost in terms of execution time, c_i is the worst-case computation cost and a_i is the arrival time of the task τ_i. Tasks are sporadic, preemptive and have fixed priority. Individual task deadlines are unknown; however, each job is considered schedulable if it completes execution on/before its end-to-end deadline ($D_{e2e} = |J_i|/fps$). fps denotes the frame rate of the video stream, which we assume is fixed at 25fps for all video streams. A task upon completing its execution sends its output (i.e. the decoded frame data) as a message flow to the PEs executing its child tasks (dependent task). A message flow, denoted by Msg_i is characterised by the following tuple: (P_i, T_i, PL_i, C_i); where P_i is the priority, T_i is the period, PL_i is the payload and C_i is the basic latency of the message flow Msg_i. It is important to note that, if one or more of a task's children are assigned to a single PE, then only one data flow is sent to the PE in order to avoid flow redundancy. Each task also has memory read and write flows which are not illustrated in Figure 1b. Flows inherit the t_i and p_i of the sender tasks and the PL_i of transmitting the reference frame data is $(res(VS_i) \times bits\ per\ pixel)$.

3.1.1 Deriving the Task Execution Cost

The computation cost of decoding a video frame and the payload of reference data message flow, will greatly depend on the temporal and spatial variations of video streams. MPEG decoding consists of operations such as parse, decode, motion compensation (MC), inverse quantisation (IQ) and inverse discrete cosine transform (IDCT) etc. At the lowest granularity, MPEG contains *blocks* and based on how they are encoded they can be categorised into 9 types as explained by Tan et al. [31]. Depending on the type of frame and decoding steps performed on the frame, the frame type to block type relationship may vary, as shown in Table 1. Based on this information, we can model the frame execution cost as given in Equation (2). Here, the w_j term denotes the weight of

Table 1 Relationship between MPEG block types and frame types (adapted from [31]).

Frame type (f_t)	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9
I-frame	X								X
P-frame	X	X	X						X
B-frame	X			X	X	X	X	X	X

a type j block and the w_0 denotes the constant term in the regression model [31]. As per [31], the regression coefficients w_j are fixed for a given decoder regardless of the type/resolution of the video. We model the number of type j blocks (denoted M_j) as a uniform random variable between $\{0, \max M_j\}$, where $\max M_j$ as defined in Equation (1), is the maximum amount of blocks for a given video resolution.

Figure 2 shows frame decoding time distributions of 200 jobs (GoPs) which were synthetically generated, as per the execution model described before. The distributions show that P/B-frames have a larger range than I-frames due to the lower amount of coding options used when decoding. Furthermore, I-frames on average take longer to decode because I-frames have a high number IDCT only blocks which have a higher weighting. These distributions correlate well with previous MPEG real video stream decoding analysis seen in [19]. As defined in Equation (3), the WCET c_i of a task of type f_t in a video stream is the *maximum* of all f_t type task's execution costs x_i; therefore, different frame types would have a different c_i. For example, in Figure 2, the I-frame decoding task WCET is $\approx 0.08s$, the P-frame decoding WCET is $\approx 0.07s$ and the B-frame decoding WCET is $\approx 0.06s$. We assume the actual execution cost x_i is unknown to the dynamic task mapping algorithm at runtime.

We emphasise that the task WCET that this work is considering assumes all data a task needs is available in the local memory at the start of its execution. The data required is provided by the reference data transfers from parent tasks and reading encoded data from main memory, both of which occur before the task execution. The communication latencies related to these data transfers are dealt with separately as part of the end-to-end response time calculation presented in the preceding sections.

$$\max M_j = \frac{res(VS_i)}{block\_size}, \text{ (0 if block } M_j \text{ not enabled in frame type)} \tag{1}$$

$$x_i = w_0 + \sum_{1 \le j \le 9} w_j \times rand(0, \max M_j) \tag{2}$$

$$c_i(f_t) = \max\{x_i \text{ of } \tau_i \in VS_i \mid \text{frame type} = f_t\} \tag{3}$$

3.1.2 Task Priority Assignment

The task and flow priority assignment between different jobs of the same video stream do not change. The resource manager assigns a priority to each task in the first job (J_0) of the video stream upon admission. These priories are then fixed and the exact same priority values are used for tasks in subsequent jobs of that stream. This is supported by the fact that each job in the stream has the same number of tasks and dependency structure.

Isovic et al. [18], introduces a quality and dependency-aware frame priority assignment which we have adapted to fit our 12-frame GoP sequence as given in Equation (4). For a given video stream, task priorities are assigned according to Equation (5), where tasks of low resolution video streams are given higher priority over high-resolution video streams. Here f_{ix} denotes the

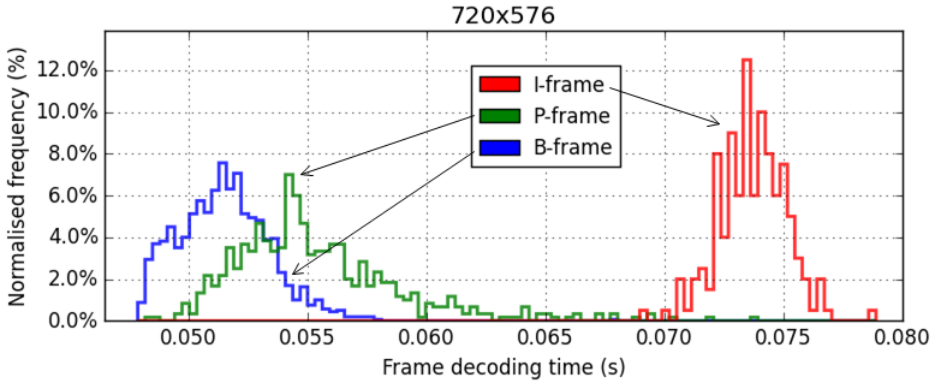

Figure 2 Frame decoding time distribution of synthetically generated workload (720 × 576 resolution video, 200 jobs).

frame index within the GoP. This assignment ensures low-resolution video streams will have a lower chance of blocking resulting in lower response-times than high-resolution streams. The ($t_c \times$ offset) component ensures that unique job-level priorities and earliest arrival time is used to select between equal resolution video streams (t_c denotes current time). Flows inherit the priority of their source tasks. A higher p_i value denotes a higher priority.

$$GoP^{pr}_{fr} = \{12, 11, 4, 7, 10, 3, 5, 9, 2, 6, 1, 8\} \tag{4}$$

$$p_i = (res(VS_i) - GoP^{pr}_{fr}[f_{ix}]) + (t_c \times \text{offset}) \tag{5}$$

It is important to note that this work is not trying to solve the priority assignment problem or propose a new assignment scheme, for dynamic workloads. This work purely attempts to optimise on the task allocation, for a given priority assignment. We have selected a sensible assignment policy that reflects common practice in the video streaming industry. The proposed mapping heuristics can work around any other priority assignment approach.

3.1.3 Job Arrival Rate

Video stream start/end times are arbitrary. We assume the video streams have a variable bit rate (VBR), which means the video stream data will be arriving at the input of the system at a variable rate. In reality, the input could be more bursty in nature due to the variability in the transmission medium (e.g. the Internet). However, we assume that the VBR encoding has a user-specified upper bound related to the stream frame-rate; therefore the job arrival rate can be modelled as sporadic with a minimum inter-arrival time. The arrival rate of a job is therefore modelled as per Equation (6) where J^{rate}_{min} and J^{rate}_{max} are workload parameters, usually set to 1.0 and 1.3 respectively. Decreasing J^{rate}_{min}, would increase the chance of new jobs arriving before the deadline of the previous job has passed, and increasing J^{rate}_{max} too much can make the system more idle. We assume all tasks of a job J_i arrive at the same time instant and hence, the period of all tasks and flows (i.e. t_i and T_i) within a job are equal and set to the minimum inter-job arrival time (i.e. 12/25=0.48s, for 25fps and 12 tasks per job).

$$\text{Arrival rate}(J_i) = rand(J^{rate}_{min} \times D_{e2e}, J^{rate}_{max} \times D_{e2e}) \tag{6}$$

3.2 Platform Model

The multi-core platform is composed of P homogeneous PEs connected by a NoC. The NoC platform model uses wormhole packet switching, fixed priority preemptive arbitration, has a

2D mesh topology and uses the XY deterministic algorithm for routing such as in [5]. XY deterministic routing assures that the message flows will always use the same resources of the NoC (i.e. same path) to deliver data from a given specific source and destination. This property is a requirement of the flow response time analysis used by our deterministic admission controller. Therefore, any deterministic routing technique that guarantees a network path is compatible with the presented analysis. The NoC link arbiters are priority-preemptive thus making it easier to predict the outcome of network contention for specific scenarios. All inter-PE and PE-to-memory communication occurs via the NoC by passing messages. Each PE contains a local memory, a priority-preemptive local scheduler, task queue and a dependency buffer. The PE upon completing a task's execution, transmits its output to the appropriate PEs dependency buffer. Tasks can begin execution on a PE *iff* all its data dependencies have arrived at the dependency buffer. If two tasks are mapped on the same PE it is assumed they do not need to communicate through channels of the NoC; hence the output of the source task will immediately be available at the dependency buffer of its PE. Higher priority tasks that have all dependencies fulfilled can interrupt already running lower priority tasks. Once a task finishes its execution the local scheduler picks the next highest priority task with dependencies fulfilled, to be executed. Similarly, higher priority flows can interfere lower priority flows that share the same network link.

The global input buffers are located in the main memory, and the task data (i.e. the encoded MPEG frame data) must be transmitted from the main memory via the NoC to the respective PE before execution (referred to as *memory read*). Additionally, after the MPEG decoding task has completed, its output (i.e. decoded MPEG frame data) is transmitted to the frame-buffer located in the main memory (referred to as *memory write*). The model assumes an encoded MPEG-2 I-frame (with $\approx 40\%$ frame compression), is twice as big as a P-frame and 4 times as big as a B-frame [19]. The encoded frame byte size and the decoded frame byte size represents the memory read and write traffic payloads respectively. We assume memory read flows have higher priority over data and memory write flows. The platform model assumes 4 main memory controllers (MMC), placed on the four sides of the NoC. A task communicates with the MMC closest to it.

3.3 Open-Loop Runtime Resource Manager

The resource manager (RM) of the system (Figure 1a), performs task mapping, priority assignment, admission control (AC) and task dispatching to the PEs. In our platform model we assume the RM is a separate entity/component, however it could also reside in one of the PEs. Task mapping and priority assignment is performed only once for a video stream at the admission of the first job J_0; all subsequent jobs of the video stream follow the initial mapping and priority assignment. The RM also maintains a *runtime mapping table* (TMT) of the jobs of every active video stream in the system. This mapping table contains the following task information:

- Real-time properties: $\{c_i, t_i, p_i\}$,
- Non-real-time properties: $\{f_t, f_{ix}, \}$,
- Task mapping: indicates which PE a specific task τ_i is mapped to.

The TMT is populated with each task of the first job J_0 of an admitted video stream. Once the video stream has stopped/finished, the task entries related to the stream are removed. The RM also has knowledge of the fixed TG dependency structure used by all video streams. The above TMT information is looked-up during the deterministic admission control and runtime task mapping operations. For example, the task mapping techniques proposed in this work, make use of the information in the TMT (e.g. task WCET, priorities and mapped PE) to determine the worst-case interference for a mapped task. Therefore, unlike in previous work [30, 15], our proposed resource management technique is open-loop and does not require any feedback/monitoring mechanism.

3.4 Problem Statement

A deterministic admission controller (D-AC) decides whether to reject or admit a video stream, based on the schedulability of the new and existing video streams. This enables the system to give a hard real-time video stream decoding guarantee to the end-user. However, the D-AC tests result in under-utilised system resources, due to the pessimistic nature of the RTA [23, 24]. With proper task to PE mapping approaches, admission rates and system utilisation can be improved. This is challenging as certain workload characteristics such as execution time, arrival patterns and task and flow interferences are unknown a priori. We present heuristic based runtime mapping approaches that consider the current state of the PEs and the task and flow blocking behaviour in order to minimise communication and computation load of the processing elements. The goal is to develop task mapping heuristics that will lower the worst-case response-time (WCRT) of the video stream such that the D-AC will increase its admission rate, leading to better utilised systems.

4 Deterministic Admission Control

The deterministic admission controller (D-AC) is invoked when a new video stream request is received. It performs RTA to determine if any of the new or existing/active video streams would miss their end-to-end deadline, by admitting the new video stream. Algorithm 1 shows the steps involved in D-AC decision process. Firstly, upon arrival of a new video stream, the online task mapping heuristic is initiated to assign tasks and flows, processor and priority allocations (line 1). The task mapping details are then added temporarily to the TMT (line 2), to account for the additional resource contention incurred by potentially admitting a new stream. If the video is rejected, then the entries are removed (line 22). After the task mapping, the flows (and their real-time properties) resulting from the mapping can be generated (line 3).

With the information above, the calculation of the WCRT of tasks and flows of all video streams in the system can be initiated (lines 4–11 of Algorithm 1). Higher priority tasks and flows of one stream can block lower priority tasks and flows of other active video streams. A flow Msg_i can have two types of interference sources – *direct* and *indirect* interference. The direct interference flow set (denoted S_{id}) are higher priority flows that have at least one physical link in common with the observed traffic-flow. The indirect flow set (denoted S_{ii}) are higher-priority flows with no shared links with the observed flow but share at least one link with a flow in S_{id}. Equation (9) given in [28, 16] is used to calculate the upper bound for the worst-case network latency R_i of each traffic flow (Msg_i) in wormhole switching, fixed priority preemptive NoCs. The task WCRT (r_i in Equation (7), originally introduced in [3]) is used as release jitter J_i^I of message flows. In Equation (7), $hp(\tau_i)$ is the set of higher priority tasks mapped on the same PE and we have excluded the non-interfering task set $ni(\tau_i)$ due to the known precedence constraints. The basic latency C_i of a message flow, calculated using Equation (8), is the flow transmission latency when no flow contention is present. In Equation (8), $Hops$ is the number of hops between source and destination, RL is the link traversal time, $numFlits$ is the payload size and HL denotes the time needed to route a packet header. In Equation (9), we adjust S_{id}, such that non-interfering flows $ni(Msg_i)$ of flow Msg_i are excluded due to task precedence constraints. The terms t_j and T_j in Equation (7) and Equation (9) denote the task and flow periods respectively. The calculated WCRT of tasks and flows are saved in TMT to be used to calculate the WCRT of the video stream job.

Algorithm 1 Deterministic admission control pseudo-code.

Input: TMT : Runtime task mapping table;
Real-time task properties of new video stream request VS_k
Output: Boolean AC-decision : admit/reject
// Perform mapping and derive resulting flow set
1: Map all tasks $\tau_q \in VS_k$ to NoC PEs and assign priorities.
2: Temporarily insert VS_k task mapping details to TMT
3: Derive all valid periodic flows in system (for above mapping configuration) : FT_{temp}
// Calculate WCRT of each task and flow in TMT
4: **for all** $\tau_i \in TMT$ **do**
5: Find interference set $hp(\tau_i)$ (exclude $ni(\tau_i)$)
6: Calculate task WCRT – Equation (7), save value in TMT
7: **end for**
8: **for all** $Msg_i \in FT_{temp}$ **do**
9: Find interference sets S_{id}, S_{ii} (exclude $ni(Msg_i)$)
10: Calculate task & flow WCRT – Equation (9) , save value in FT_{temp}
11: **end for**
// Calculate $WCRT(J_i^{CP})$ of all video streams
12: **for all** $VS_i \in TMT$ **do**
13: vs_p : init. structure for all simple paths of VS_i job
14: **for all** $paths \in job(VS_i)$ **do**
15: Calc. $path$ response-time $= \sum_{Msg_q \in path} R_q$; save to vs_p
16: **end for**
17: Critical path of VS_i job : $J_i^{CP} = \max\{vs_p\}$
// Check video stream schedulability
18: **if** $WCRT(J_i^{CP}) \leq D_{e2e}$ **then**
19: $ac\_decision = TRUE$
20: **else**
21: $ac\_decision = FALSE$
22: Remove VS_k details from TMT
23: **return** $ac\_decision$
24: **end if**
25: **end for**
26: **return** $ac\_decision$

$$r_i^{n+1} = c_i + \sum_{\forall \tau_j \in \{hp(\tau_i) \setminus ni(\tau_i)\}} \left\lceil \frac{r_i^n}{t_j} \right\rceil c_j \tag{7}$$

$$C_i = (HL \times Hops) + (RL \times (Hops - 1)) + (HL \times numFlits) \tag{8}$$

$$R_i^{n+1} = C_i + \sum_{\forall Msg_j \in \{S_{id} \setminus ni(Msg_i)\}} \left\lceil \frac{R_i^n + r_j + J_j^I}{T_j} \right\rceil C_j \tag{9}$$

In lines 12–24 of Algorithm 1, the WCRT of the critical path of the job $WCRT(J_i^{CP})$ is calculated. Recall that the video streams use a fixed job structure, hence there are fixed number of simple paths of the TG known a priori. For each path of a video stream job $job(VS_i)$ the summation of the WCRT of all nodes and edges is calculated (line 15); note that the WCRT of the source task is included in R_i as release-jitter r_j. The job critical path J_i^{CP} is the path with the maximum accumulated cost (line 17). A video stream is granted admission, *iff* the expression given in Equation (10) is true for the new and all active video streams in the system (lines 18–24

in Algorithm 1); this guarantees that the worst-case timing requirements of all existing and new video streams will be successfully met. It is important to note that the J_i^{CP} and $WCRT(J_i^{CP})$ are properties of a given task-to-PE assignment, hence task mapping is integral to the D-AC decision.

$$WCRT(J_i^{CP}) \leq D_{e2e} \tag{10}$$

4.1 Exclusion of Non-Interfering Tasks and Flows

Precedence constraints of the tasks are taken in to account when calculating the task and flow interference. For this analysis, it is assumed there is no overlap between consecutive jobs within the same video stream. The end-to-end RTA given in [16] assumes a synchronous pipeline execution mode, where multiple instances of the TG or portions of the TG (i.e. from a prior job in the same video stream) can be simultaneously executing in the system. In this work, we assume there is no overlap in execution between consecutive jobs of the same video stream. Hence, when deriving the $hp(\tau_i)$ of a task, we can exclude dependent/successor tasks. Likewise, when calculating the direct (S_{id}) and indirect (S_{ii}) flow interference sets the task precedences are taken into account to determine non-interfering flows.

We now formally present the non-interference set of tasks and flows with respect to precedence constraints. Within the application TG, there exists different simple paths (also referred to as *paths*). A simple path is a topologically ordered set of nodes and edges which does not have repeating vertices and are a subset of the TG. For example, the simple path ($P_1 \Rightarrow B_9$) consists of the frame decoding tasks $P_1, P4, P7, B9$ and the flows between them. We define two distinct simple path types *to* and *from* a node in the TG. The *ancestral simple path* (expressed as ($\tau_0 \Rightarrow \tau_i$)) consists of path from root node (τ_0) to the target node τ_i; the *descendant simple path* (expressed as ($\tau_i \Rightarrow \tau_{-1}$)) consists of the path from target node τ_i to any leaf node (τ_{-1}) in the TG. For example in the TG (Figure 1b), if we consider $\tau_i = P4$, then the nodes I_0 and P_1 will lie on the ancestral simple path, and the nodes P_7 and one leaf node $B_{8/9/10/11}$ will lie on the descendant simple path. Hence, the non-interference set of a task τ_i can be defined as per Equation (11). Similarly flows in ancestral and descendant simple paths will not interfere with the target flow as given by Equation (12). Here, the ($Msg_i \colon \tau_s \rightarrow \tau_d$) component denotes the target flow Msg_i and its source and destination tasks τ_s and τ_d respectively.

$$ni(\tau_i) = \tau_k \in \{\tau_0 \Rightarrow \tau_i \cup \tau_i \Rightarrow \tau_{-1}\} \tag{11}$$

$$ni(Msg_i \colon \tau_s \rightarrow \tau_d) = Msg_k \in \{\tau_0 \Rightarrow \tau_s \cup \tau_d \Rightarrow \tau_{-1}\} \tag{12}$$

5 Proposed Runtime Mapping Approaches

5.1 Least Worst-Case Remaining Slack (LWCRS)

The difference between the task deadline and worst-case computation cost (i.e. task slack), is used as a primary metric when determining the task-to-PE mapping. The heuristic (Algorithm 2) takes into account the worst-case blocking factor introduced by $hp(\tau_i)$ to determine the PE that provides the least worst-case remaining slack (LWCRS) for a target task τ_i. The mappers make use of the information stored in the runtime TMT (as described in Section 3.3), to determine the worst-case blocking for a task. Algorithm 2, iterates through the provided *PE_list* and calculates the following for each task-to-PE mapping: $RemSlack_t$ – the worst-case *remaining* slack (WCRS) of τ_i – taking into account blocking induced by $hp(\tau_i)$ (line 7); $RemSlack_{lp}$ – the WCRS for each of the lower-priority tasks already mapped on the PE ($lp(\tau_i)$), taking into account the (c_i) of τ_i (line 10-11). A weight ($w = RemSlack_t + RemSlack_{lp}$) is then assigned to all PE in *PE_list*(line 14); the PE with the lowest w provides the LWCRS. The heuristic attempts to find the PE

Algorithm 2 *_get_PE_least_slack* pseudo-code – Find PE with the least worst-case remaining slack.

Input: τ_i : target task; TMT : copy of the runtime task mapping table; *PE_list* : list of PEs to search
Output: tuple : (result PE, search result (boolean))

$PE\_packing = \{\}$
for all $PE_i \in PE_{list}$ **do**
  // obtain following from TMT
  Get $MPT(PE_i)$: tasks already mapped on PE_i
  Get $hp(\tau_i), lp(\tau_i)$: high/low priority tasks in $MPT(PE_i)$
  // get worst-case remaining slack (WCRS) to target task
  τ_i^{slack} : task slack w.r.t estimated sub-task deadline
  $RemSlack_t = \tau_i^{slack} - \sum_{\forall \tau_j \in hp(\tau_i)} c_j$
  // get WCRS on low-pri mapped tasks
  $RemSlack_{lp} = \{\}$
  for all $\tau_j \in lp(\tau_i)$ **do**
    $RemSlack_j = \tau_j^{slack} - \sum_{\forall \tau_k \in hp(\tau_j)} c_k$
    Insert $RemSlack_j$ to $RemSlack_{lp}$
  end for
  // populate local data structure
  if $RemSlack_t > 0$ and $\forall x \in RemSlack_{lp} | x > 0$ **then**
    $PE\_packing\,[PE_i] = RemSlack_t + \sum RemSlack_{lp}$
  end if
end for
// if none of the PEs in the list will provide a positive WCRS, then choose the PE with min. utilisation
if $\forall x \in PE\_packing | x > 0$ **then**
  PE_j = index of MIN(PE_packing)
  return (PE_j, FALSE)
else
  PE_j = *_get_PE_min_util*(TMT, *PE_list*)
  return (PE_j, FALSE)
end if

mapping that will result in a tight temporal-fit, without missing the deadlines of τ_i nor any of the already mapped tasks. Individual task deadlines are unknown, hence they are estimated via the technique proposed in [20], where the cumulative remaining taskset slack is divided proportional to c_i. If a suitable PE with positive slack is not found, the algorithm will return the PE with minimum utilisation.

5.2 LWCRS-Aware Mapping

The LWCRS-aware mapping approach (Algorithm 3), makes use of the LWCRS utility function explained in Section 5.1 as well as tries to minimise distance between communicating tasks. The primary objective of the algorithm is to tightly pack (i.e in the temporal domain) each task of the job, into the PEs, in order to leave room for tasks of future video streams. The LWCRS mapper will ensure that initial PEs in the NoC will be heavily utilised before selecting the next available PE, whilst not violating the subtask deadlines. This increases the number of simultaneous video streams that the system can handle without missing any deadlines. LWCRS-aware mapping is a general purpose technique, that can be applied to map other types of applications that have dependency/communication patterns.

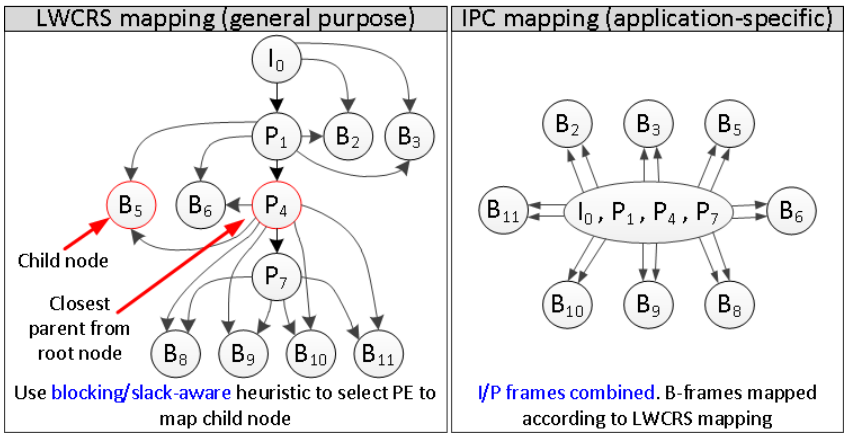

Figure 3 Illustration of LWCRS mapping closest parent selection *(left)* and IPC mapping task grouping *(right)*.

The algorithm of LWCRS mapping is given in Algorithm 3. The algorithm uses a copy of the runtime task mapping table TMT (maintained by the RM). The existing task-to-PE mappings and mapped task properties are stored in the TMT. Algorithm 2 is used within Algorithm 3, to find a PE which gives the LWCRS (lines 3 and 12). Firstly, the PE which gives the lowest slack is selected to map the root node of the TG (line 3). For all other nodes in the TG, the algorithm maps each node with an increasing hop distances distance from its *closest parent* (lines 9–17). A nodes' closest parent is defined as the node with the longest path from the root node. For example in Figure 3, B5 has 2 parents – P4 and P1; however P4 is the closest parent due to the longer path from the root node (therefore $\tau_{B5}^{PARENT} = P4$). If no suitable PE with remaining slack is found, the algorithm maps the target node to the PE with minimum utilisation (lines 17–20). The algorithm attempts to reduce long-communication routes between communicating tasks in order to reduce network congestion and communication costs. Each node in the TG is mapped onto a PE that gives the LWCRS as well as close proximity to τ_i^{PARENT} (lines 6–15). TMT is updated in each iteration.

5.3 I and P Frames Combined Mapping (IPC)

Unlike the LWCRS-aware mapper, IPC exploits known application-specific communication and dependency patterns. By inspecting the video job TG (Figure 3), we can see that the I and P frames lie on the longest-path in the TG. We define longest-path as the path in the TG with most number of non-repeating nodes (assuming edge weights are equal). Furthermore, the path $I_0 \rightarrow P_1 \rightarrow P_4 \rightarrow P_7$ is most often the critical path (CP) of a job, assuming B-frame decoding tasks do not experience severe blocking. Figure 3 (right) illustrates the TG after the grouping of I and P frames. Combining the I/P frames together has two distinct advantages : (a) it reduces the NoC congestion/interference as fewer flows need to be injected into the NoC; (b) it reduces the end-to-end response time of a job since the TG's potential CP is executed as soon as possible, without waiting for message flows. The IPC mapping technique works as follows. Firstly, the I and P frame decoding tasks of the job are grouped and mapped to the lowest-utilised PE on the platform. The B-frame decoding tasks are mapped as close to their parent tasks with a 2 hop distance constraint. The LWCRS heuristic (Algorithm 2 described in Section 5.1) is used to select a PE within the 2 hop distance region. B-frames have no inter-dependencies, hence they can be processed in parallel. The B-frame decoding computation cost is lower than I/P frames, hence they can occupy smaller temporal gaps in the PE task queues.

Algorithm 3 LWCRS-aware mapping heuristic algorithm pseudo-code.

Input: all tasks in the job (J_0), TMT : copy of the runtime task mapping table
Output: task to processing element mapping : MPG ($\tau_i \rightarrow PE_i$)

for all unmapped tasks : $\tau_i \in J_0$ **do**
  if τ_i is *root_node* **then**
    (PE_i, $found$)=***__get__PE__least__slack***(τ_i, TMT, PEs)
    Map ($\tau_i \rightarrow PE_i$); Update TMT;
  else
    // obtain following from TMT
    Get τ_i^{PARENT} : parent task with max route cost.
    Get PE_i^P : PE that τ_i^{PARENT} was mapped onto
    // get neighbours of increasing hop_counts
    for $hc = 1$ to MAX_HOPS **do**
      $N\_PE_i^P = \_get\_neighbours(PE_i^P, hc)$
      Append PE_i^P into $N\_PE_i^P$
      (PE_i, $found$) = ***__get__PE__least__slack***(τ_i, TMT, $N\_PE_i^P$)
      if $found$ is $TRUE$ **then**
        Map ($\tau_i \rightarrow PE_i$); Update TMT;
        break loop; Go to step 1;
      end if
    end for
    // if no suitable PE is found, select closest min. util. PE
    if a suitable PE is NOT found **then**
      $N\_PE_i^P = \_get\_neighbours(PE_i^P, hop\_count = 1)$
      $PE_i = \_get\_PE\_min\_util$(TMT, $N\_PE_i^P$)
      Map ($\tau_i \rightarrow PE_i$); Update TMT
    end if
  end if
end for
return TMT

6 Static Hard Real-Time Mapper

Static-mapping algorithms are used for multiprocessor systems when the application characteristics and workloads are known at design-time. To evaluate our heuristic based dynamic mapping techniques, we compare against a static search-based hard real-time (HRT) mapper as introduced by Sayuti et al. [26]. We adapt and modify this static mapper to suit our application and platform model. In this particular technique, the authors use a genetic algorithm (GA) based optimisation strategy to optimise the task to PE mapping approach. GAs have been used in the past to optimise task allocation for multi/many-core systems [2, 32]; however the work done in [26] was the first to explore task mapping of hard real-time tasks whilst optimising multiple objectives. GAs start with a random initial population of candidate solutions and gradually evolves the populations towards the global optimum using a given fitness statistic. In Sayuti et al. [26] the GA individual is represented by an integer-based chromosome structure indicating the PE mapping for each task. As illustrated in Figure 4, their algorithm uses simple evolutionary GA pipeline constructs such as single-point crossover, bit-flip mutation and binary tournament selection to generate a new population for each generation. They use elitism to ensure the best individual of each generation is advanced to the next generation. A multiple objective fitness function (application schedulability – Equation (9) and power dissipation for every individual mapping solution) is used to guide a random search towards solutions of increasing fitness. It is important to note however that due to the random nature of solution development, GAs do not guarantee optimality even when it may be reached.

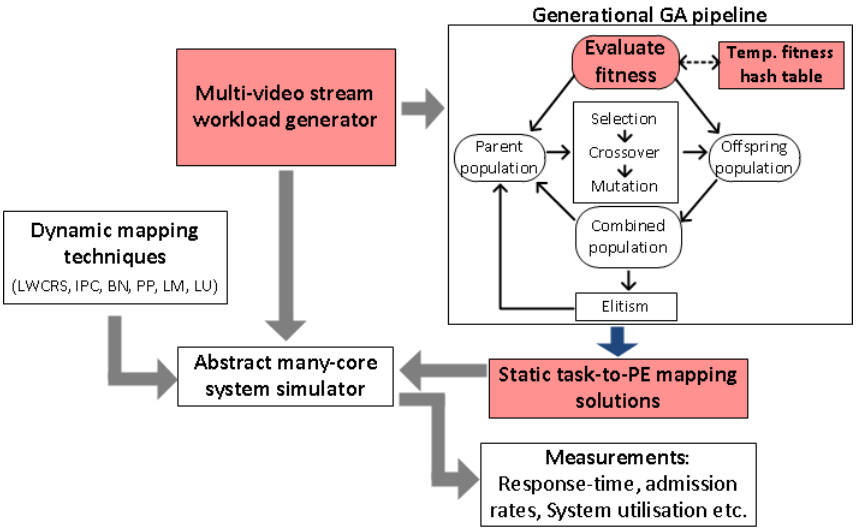

Figure 4 The GA pipeline from [26] adapted and integrated to the experimental design flow.

This static HRT mapper is purely used as an upper baseline, to evaluate the proposed heuristic based mapping techniques. If workload characteristics are known at design time, we can use a static HRT mapper to find a mapping configuration that will optimise our chosen metrics. This obtained experimental upper bound is valuable to assess the performance of the dynamic mappers results for a given workload. Dynamic mappers rely on fast heuristics and do not have complete knowledge of the workload at runtime; hence an indication of a good heuristic is one which shows results as close as possible to the upper-bound. However, recall that static mappers not only rely on full knowledge of the workload, but also incur a considerable runtime execution overhead. In this work we show via experimentation that even for small NoCs, under low workload conditions the static mapper will take up to several hours to find a reasonably optimised mapping configuration. The computation complexity of the fitness function will increase exponentially as the workload increases. For these reasons, GA-based static HRT mappers are not suitable for use in runtime mapping.

The GA-based HRT mapping optimisation given in [26] has been significantly, adapted to be able to integrate it with our application model and metrics. The red shaded components in Figure 4 indicate the changes made to the GA and the design flow with respect to the GA in [26]. In the experimental design flow, the same workload is first generated and input into the GA pipeline to obtain mapping solutions. The final GA mapping solutions are then taken and used in the system simulator to obtain performance measurements and compare against the mapping results provided by the dynamic mappers. The following sub-sections outline the adaptations done to the GA design in [26].

6.1 Points-Based GA Fitness Function

Unlike in [26] we are concerned with the end-to-end (E2E) schedulability of an entire video stream rather than individual tasks/flows. We use a novel points-based single-objective fitness function as described in Algorithm 4. The loop in lines 3–12 evaluates and accumulates the points for each VS_i in the workload (WL). Video streams that have a higher $WCRT(J_i^{CP})$ than their deadline are given a positive point based on the amount of which they have missed their deadline (i.e. $WCRT(J_i^{CP}) - D_{e2e}$). On the other hand video streams that are fully schedulable are awarded

Algorithm 4 Points-based GA fitness function.

1: $points = 0$ //Calculate points for all videos in workload (WL)
2: **for all** $VS_i \in WL$ **do**
3: Calculate WCRT of all sporadic tasks and flows of VS_i
4: Find J_i^{CP} of VS_i
5: **if** $J_i^{CP} \geq D_{e2e}$ **then**
6: // unschedulable
7: $points \mathrel{+}= 1 \times \frac{WCRT(J_i^{CP}) - D_{e2e}}{D_{e2e}}$
8: **else**
9: // fully schedulable
10: $points \mathrel{+}= -1 \times \frac{D_{e2e} - WCRT(J_i^{CP})}{D_{e2e}}$
11: **end if**
12: **end for**
13: **return** *points*

a negative point relative to their slack (i.e. $D_{e2e} - WCRT(J_i^{CP})$). The ratios in line 7 and 10 of Algorithm 4 indicates the extent to which a video stream is unschedulable/schedulable. Individuals with negative points have a higher fitness score than those with positive points. This points-based fitness scoring system would enable the GA to pick individuals with task mappings that have lower distributions of $WCRT(J_i^{CP})$. For example consider the following scenario. Two GA individuals, A and B have equal number of fully schedulable streams and one unschedulable stream each, but they have fitness scores -1.65 and -1.15 respectively. This indicates that A's unschedulable video missed its deadline by a lower margin than B's unschedulable video. This differentiation can not be made if an integer based fitness score is used, where both these individuals would be ranked equally.

6.2 Application-Specific Adaptations

In addition to the novel fitness function, we introduce certain extensions to the application model in [26] to accommodate our application specific task-model. For example, precedence constraints were taken into account when calculating the task/flow interference sets (Section 4.1). Memory read/write traffic has been incorporated such that for each mapping configuration (i.e. each chromosome) tasks-to-MMC selection (Section 3.2). Redundant flows are removed for each chromosome when multiple children are mapped to the same PE (Section 3.1).

6.3 GA Design Optimisations

Due to the extensions above, the fitness evaluation of the GA becomes more complex. To alleviate this issue, we limit the recursion depth of the WCRT analysis in Equation (9). Furthermore, a hash table of task mapping solutions and corresponding fitness scores are maintained and looked-up to avoid having to evaluate the same gene more than once. Subsequently, the GA evolution cycle terminates immediately if it encounters an individual with an acceptable mapping solution. Such a mapping solution will result in all the admitted video streams to be schedulable; in other words, the maximum $WCRT(J_i^{CP})$ of all the video stream is less than the E2E deadline: $\max_{VS_i \in WL}(WCRT(J_i^{CP})) < D_{e2e}$

7 Evaluation

7.1 Experiment Design

We wish to evaluate the performance of the proposed task mapping schemes in terms of admission rates and PE utilisation. Experimental evaluation is performed through a discrete-event, abstract simulation of a 3x3 NoC platform with the characteristics described in Section 3. The PEs are assumed to be operating at 200MHz and the NoC frequency is set to 10MHz with 7 clock-cycle routing latency and 16 byte link width. A lower NoC frequency (i.e. low bandwidth) is assumed, in order to induce an experimental condition with a reasonable amount of network utilisation/congestion. The NoC and the PEs use priority-preemptive arbitration and scheduling respectively. The light-weight NoC simulation component described in [17] is used to model the NoC communication traffic and interference patterns. The D-AC in Section 4 is used for all the simulation runs of this experiment.

Synthetic abstract video streams are used as workloads for all experiments as described by the application model in Section 3.1. The task execution costs are calculated using the block-level frame decoding cost model described in Section 3.1.1. All synthetic streams have a fixed frame rate of 25fps and 12 frame GoPs. The inter-arrival time of the video stream jobs are uniformly distributed between $1.0 \times D_{e2e}$ and $1.3 \times D_{e2e}$.

7.1.1 Varying Workload

The total workload introduced to the system can be defined as a summation of the resolutions of all the video streams admitted and active in the system, as shown in Equation (13). Mapping approaches for a range of workload values are evaluated; starting from a single video stream with 230×180 resolution to 9 parallel video streams with 720×576 resolution. Each experiment contains 30 simulation runs with different random seeds, which results in varying video stream arrival patterns and task execution costs.

$$\text{Total workload value} = \sum_{\forall VS_i \in WL} [frame\_h(v_i) \times frame\_w(v_i)] \tag{13}$$

7.1.2 Varying Communication-to-Computation Ratio and NoC Size

The mapping techniques explored in this paper use runtime heuristics such as communication path load, hop-distance and PE utilisation. Hence, it is interesting to explore the performance of these techniques, when the application communication-to-computation ratio (CCR) varies. For example, if an application is computation-bound (low CCR) then standard load-balancing heuristics may be sufficient. On the other hand if the application is communication-bound (high CCR) then communication-aware heuristics may perform better. Several previous work in the state-of-the-art in dynamic task mapping [21, 1] does not consider the influence of varying CCR in their results, which may lead to biased results.

A single video stream can be represented by a sporadic TG as shown in Figure 1b. Thus, the CCR of a single video stream can be calculated as the ratio between the total cost of the communication edges over the total task cost in the TG, as shown in Equation (14). Here, the communication basic latency and the task WCET are the communication and task costs respectively. The CCR of a workload can then be defined as the mean CCR of all the parallel video streams included in the workload (Equation (15)). To change the CCR, we keep the task computation cost constant and gradually vary the NoC frequency. To evaluate the scalability of the proposed mappers we perform experiments under different NoC sizes and CCR combinations.

Data from 30 uniquely seeded simulation runs are obtained. Each run consists of a fixed number of simultaneous video streams but with different arrival patterns and varying task execution costs.

$$\underbrace{CCR(VS_i)}_{\substack{\text{CCR of a single} \\ \text{video stream}}} = \frac{\text{Total edges cost}}{\text{Total nodes cost}} = \frac{\sum\limits_{\forall e \in edges} C_i}{\sum\limits_{\forall \tau_i \in J_i} c_i} \tag{14}$$

$$CCR_{WL} = \frac{\sum\limits_{\forall VS_i \in WL} CCR(VS_i)}{|VS|} \tag{15}$$

7.1.3 Metrics

For each simulation run we measure the video *admission-rates* and *PE and NoC busy times*. The admission rate is calculated as a ratio between the admitted video streams over the total video stream decoding requests. A D-AC ensures all admitted videos will be fully schedulable. The percentage PE busy time (also referred to as PE utilisation) is measured as the ratio between the total active (busy) time of all PEs in the system over the total simulation time. The percentage NoC busy time (also referred to as NoC utilisation) is the ratio between the total active time of the NoC links over the total simulation time.

The objective is to increase the admission rate of the system and thereby decrease the PE idle-time. Higher admission rates using lower NoC usage is advantageous because it can potentially lead to lower power consumption.

7.2 Baseline Mapping Heuristics

7.2.1 Dynamic Mapping Heuristics

The path-load based best-neighbour (BN) heuristic defined in [8], and the *pre-processing (PP)* based algorithm defined in [21] are used as baselines. The original BN algorithm was adapted to support multiple tasks and have used PE utilisation to determine available PEs, while maintaining path-load as the main heuristic. Since the dependency pattern of the tasks are assumed to be known beforehand, the pre-processing stage of the PP algorithm is performed at design-time. While PP takes into account both the communication and the computation properties of the tasks the BN heuristic focuses mainly on communication link congestion. Evaluation is also performed against two load-balancing task allocation heuristics which attempt to evenly distribute the load of the application across available PEs. The *lowest utilised (LU)* heuristic iterates through all tasks in the job and maps each task to the analytical lowest utilised PE. The worst-case utilisation of a PE is measured as given in Equation (16), where $MPT(PE_i)$ denotes all tasks mapped on PE_i. Since we do not know the actual execution cost of the tasks, we use the worst-case computation cost (c_i). At the end of each iteration the respective local mapping table is updated with the new task-to-PE mapping. Finally, the *least mapped (LM)* heuristic, selects the PE with the minimum number of mapped tasks according to the runtime task-mapping table.

$$U = \sum_{\forall \tau_i \in MPT(PE_i)} \left\lceil \frac{c_i}{t_i} \right\rceil \tag{16}$$

7.2.2 Static GA-Based HRT Mapper

The GA-based HRT mapper (GA-MP) described in Section 6 is used to act as an upper-baseline for our proposed task mapping techniques. The workload consisting of multiple video decoding

streams are generated and fed into both the GA-MP and the abstract simulator (Figure 4). Task-to-PE mappings solutions obtained via the GA-MP are then evaluated in the discrete-event abstract simulator. The performance of the static mappings obtained from the GA-MP are then compared with the dynamic mapping strategies. The crossover and mutation probability rates are important parameters in the GA-pipeline and for all our experiments we use (0.5, 0.01) respectively. Higher workloads will be executed with a larger number of GA-evaluations to suit the increasing complexity of the search problem.

7.3 Discussion of Experimental Results

7.3.1 Dynamic Mapper Performance Under Workload Variation

Figure 5 shows the performance of the different task-mapping heuristics in terms of mean admission rate and mean PE and NoC busy time, as the level of the workload is increased. Results obtained by using the dynamic task mappers (i.e. IPC, LWCRS, PP, BN, LM and LU) are first discussed. The admission rates for all mapping types decrease as the workload is increased, because for high workloads the AC cannot guarantee the timing requirements will be met, and hence rejections will be made. Using the IPC and LWCRS task mapping heuristics the WCRT of the jobs can be reduced, thus allowing the D-AC to admit a higher number of video streams than the baseline mappers. When comparing with the baseline heuristics, the proposed IPC and LWCRS dynamic mapping heuristics provide a 10%–20% improvement in low workloads, and an average of 5% improvement in high workloads, for admission rates. It is important to note that even though IPC outperforms LWCRS, IPC is an application specific heuristic which makes use of known characteristics of the video stream.

The PP heuristic outperforms the other baselines (BN, LM, LU) because it attempts to balance computation and communication by grouping the tasks in a job. The admission rates when using BN drops lower than LU and LM in the highest workload levels because the path-load heuristic alone is not sufficient. Higher admission rates result in more tasks being processed by the system, leading to increased PE utilisation as depicted in Figure 5(middle). However, the PE utilisation is a function of both the number of video streams admitted and their spatial resolution (e.g. 4 high resolution streams will yield a higher utilisation than 5 lower resolutions streams). With respect to the dynamic mapping heuristics, we can see that as the workload increases, the PE utilisation also increases; however in workload level 225.4×10^4 we see a decline because the admission rates are low as well as the admitted video streams are of a lower-resolution. The proposed IPC and LWCRS heuristics outperform the baseline dynamic mappers, except in the case where PP performs better than LWCRS at workload level 189.8×10^4, where the admitted video streams by PP is of higher resolution than LWCRS. Across the different workloads, the proposed dynamic mapping methods show an improvement of 5%–15% in PE busy time for workloads over 83.1×10^4

The NoC busy time results shown in Figure 5(bottom) complement the PE busy time results and offers further insight into the mapping behaviour. Lower PE busy times indicate the PEs are busy waiting for data to arrive at the local buffers, thus increasing the NoC usage. Out of the proposed dynamic mappers, IPC utilises the NoC more than LWCRS. LWCRS produces a tighter grouping of tasks than IPC resulting in lower number of PEs being used. Tightly grouping tasks reduces the number of data flows but does not reduce memory flows; therefore now memory traffic becomes a bottleneck, primarily congesting the local-link. Hence, LWCRS could still have a higher $WCRT(J_i^{CP})$, leading to reduced admission rates compared with IPC. In IPC, because 4 tasks in the job (i.e. I_0, P_1, P_4, P_7) are always mapped together, the algorithm will try to find other PEs to map the B-frame tasks. This leads to more PEs being used than LWCRS and thus a higher NoC busy time than LWCRS. LU, LM mappers have the highest NoC usage due to the sparse

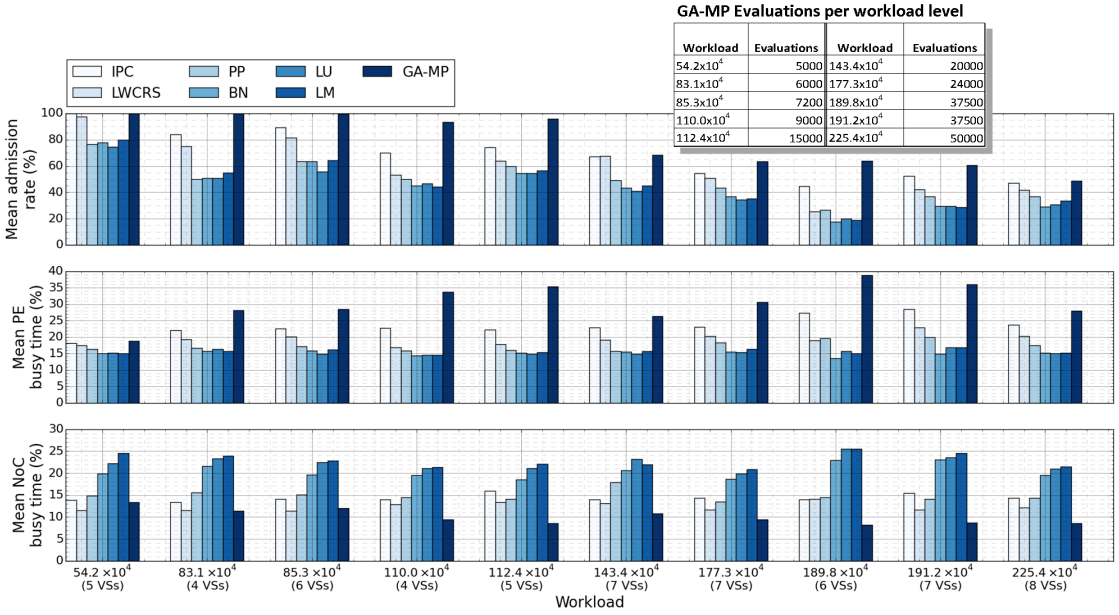

Figure 5 Workload vs. (*Top*: Admission rate; *Middle*: PE busy time; *Bottom*: NoC busy time).

distribution of tasks on all PEs. PP shows similar NoC usage to IPC, but it does not consider blocking, hence, it might place the grouped tasks to PEs that might cause higher interference.

7.3.2 Static Mapper Performance Under Workload Variation

In the Figure 5, GA-MP results denote the task mapping using the genetic algorithm based static hard real-time mapper. The number of GA evaluations taken to obtain these results are given in Figure 5(top-right table). GA-MP has full knowledge of the task characteristics and is used as an upper baseline. The trend of the results closely match the dynamic mappers (i.e. admission rates decrease and PE busy times increase as the workload is increased). Even though the GA-MP outperform all the dynamic mappers at every workload level, we notice a gradual decrease in relative improvement as the workload level increases (e.g. at workload 225.4×10^4 the IPC and GA-MP show comparable mean admission rates). Under certain conditions the GA-MP and the proposed dynamic mappers show similar results in admission rates, but the GA-MP has higher PE busy times (e.g. at workload 143.4×10^4 LWCRS show similar admission rates but poor PE utilisation). This is because in certain scenarios the GA-MP obtains a mapping which rejects lower resolution video streams but accepts higher resolution streams, thus giving rise to higher PE busy times. We noticed that GA-MP uses only a few cores per job (on average 2 or 3 cores). Furthermore, the GA tries to map children of the same parent together on the same PE, thus avoiding redundant data traffic flows. Due to these reasons, the GA-MP mapping significantly reduces the number of flows injected into the NoC. This reduces the NoC busy time, flow contention and therefore leading to higher admission rates.

As the workload increases, the number of the tasks and flows as well as their computation and communication costs increase; hence increasing the complexity of the optimisation problem. To compensate, the number of evaluations also needed to be increased to obtain a reasonable performance level. To illustrate this, the GA-based mapping optimisation is executed for the ten different workloads with a fixed generation and population size (500, 200 respectively). The search terminates when an acceptable mapping solution is found. Results (Figure 6) show that

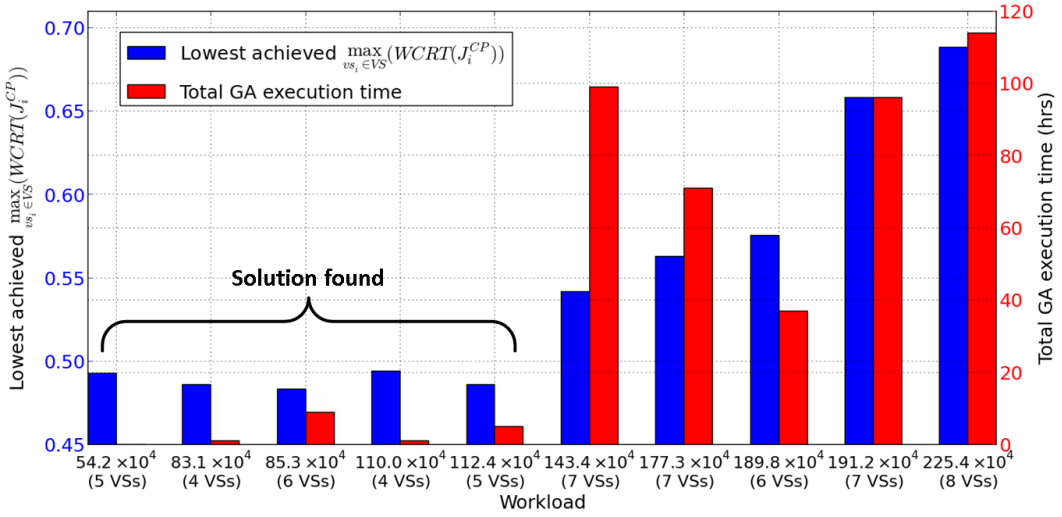

Figure 6 GA-MP performance analysis for different workload levels.

the total execution time (red bars) of the GA increases exponentially as the workload level is increased and all except the lowest workload level show an execution time in the order of tens of hours. Workload 189.8×10^4 shows a lower execution time, because the number of parallel video streams are 6 even though the workload value is higher. Note that there is a sudden increase in GA execution time from 112.4×10^4 to 143.4×10^4. This is because the number of video streams have increased from 5 to 7 and also because the GA does not terminate early, as no solution can be found for the higher workload within the fixed number of evaluations. Hence, both the number of parallel video streams and their resolutions are directly related to the GA execution time. The GA was able to find a satisfactory solution for workloads up to 112.4×10^4, however a solution was not found for the larger workloads. Similar to the execution time, we see that the lowest achieved $\max_{VS_i \in WL}(WCRT(J_i^{CP}))$ gets worse as the workload level increases; showing that as the search space and complexity increases the GA evaluations become computationally expensive as well as is unable to find a solution within a reasonable time frame.

7.3.3 Scalability and CCR Variation Evaluation

Figure 7 shows the performance variation in the different dynamic mapping techniques when evaluating for scalability and different communication loads. The y-axis of each subplot in Figure 7 displays the normalised calculated analytical $WCRT(J_i^{CP})$ of the video streams for different mapping techniques under different NoC sizes and varying CCR values. $CCR < 1.0$ denote computation-bound workloads while $CCR > 1.0$ denotes communication-bound workloads. In this experiment, we disable the admission-controller, hence all generated video streams are admitted; still, lower analytical $WCRT(J_i^{CP})$ distributions are preferred as this will lead to higher number of video streams being schedulable. The level of workload is kept proportional to the number of PEs in the platform, such that on average (over 30 seeded runs), the workload=2.2×10^5 per PE. The analytical $WCRT(J_i^{CP})$ of all mappers increase as the CCR increases, since the communication latency has effectively increased. LU and LM are computation-centric mappers, and therefore their performance deteriorates significantly under higher CCR conditions. Furthermore, LU and LM may map communicating tasks further apart as the NoC size increases, which results in a broader distribution of data points. BN does not group tasks together, but takes into account the

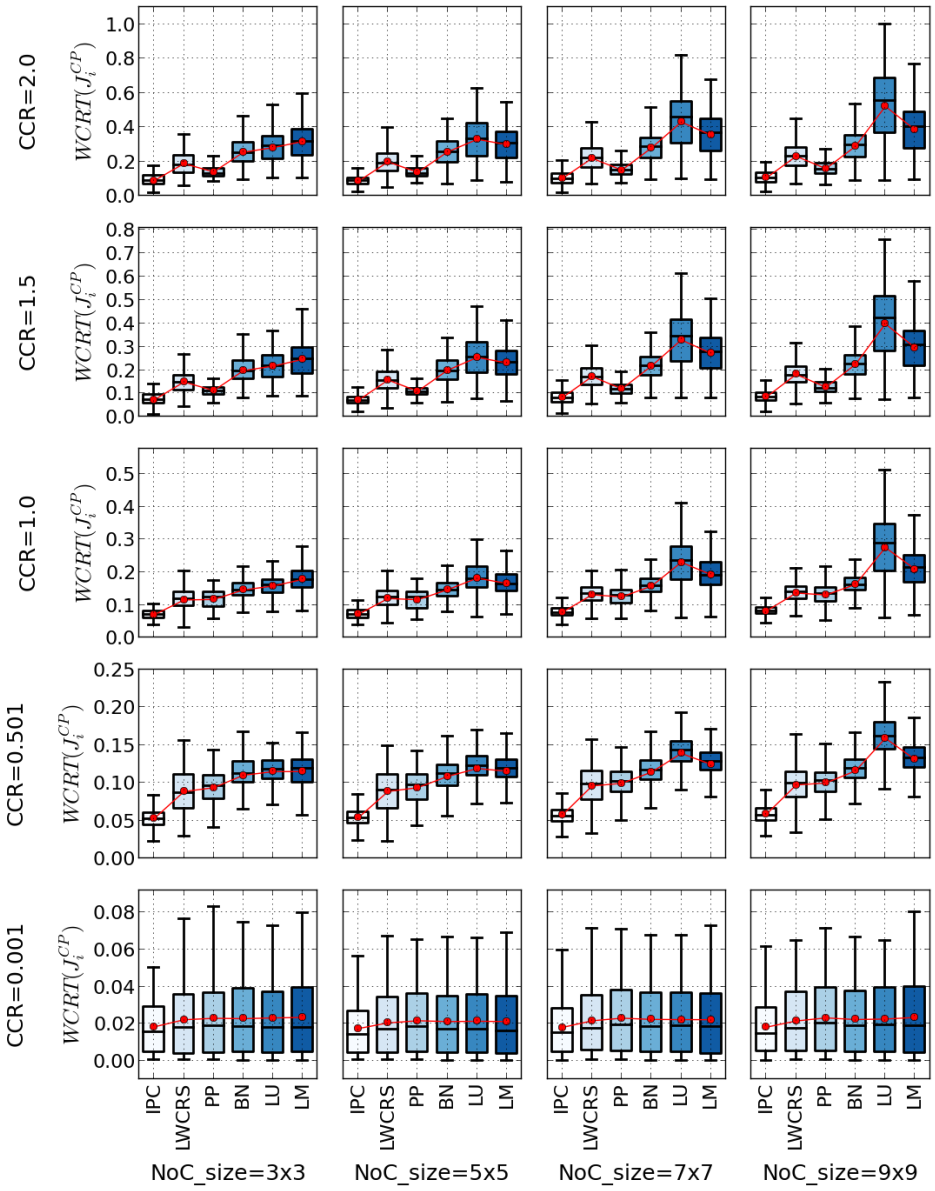

Figure 7 Normalised analytical $WCRT(J_i^{CP})$ obtained using different dynamic mapping approaches, under varying CCR and NoC sizes.

communication channel load as a metric; hence, it performs better than LM and LU in higher CCRs but still shows a higher $WCRT(J_i^{CP})$ distribution when compared to PP, LWCRS and IPC. The proposed IPC mapping method perform relatively better than all the baselines in all conditions while the proposed LWCRS method performs better than BN, LU and LM. The PP heuristic which attempts to balance computation and communication, is a strong competitor to the proposed mapping techniques. It performs better than LWCRS for $CCR > 0.5$ due to better grouping of the TG, but still shows a slightly higher $WCRT(J_i^{CP})$ distribution when compared with IPC. However, the lower bottom whiskers of LWCRS tells us that in certain workload conditions it can produce a lower $WCRT(J_i^{CP})$ than PP.

8 Conclusion

This paper formally describes a multi-stream video decoding application model and an algorithm for a deterministic admission controller, which uses video stream schedulability tests. A novel point-based, WCRT-aware fitness function for a design-time hard real-time task mapper is presented and compared against dynamic mapping techniques. This work describes two application and platform aware runtime task mapping strategies, that attempt to decrease the end-to-end response-time of the video stream decoding jobs. The first (LWCRS) technique attempts to tightly pack tasks in the temporal domain by using a novel worst-case remaining slack-aware metric of the tasks. The second technique (IPC) groups the I and P frame decoding tasks and maps them to a single PE, and the remaining tasks according to LWCRS. The techniques improve the admission rates of a hard real-time deterministic admission controller and thereby increasing system utilisation. We also present extended evaluation of these two mappers against other existing runtime mappers, under varying platform sizes and communication-to-computation loads.

Simulations carried out reflect that the proposed dynamic task mappers show an improvement of about 10%–20% in mean admission rates and an improvement of about 5%–15% in PE busy times, when compared against other existing heuristic based dynamic task-mappers. Furthermore, better admission rates and PE utilisation can be obtained at a lower usage of the NoC, which could potentially lead to lower power consumption in the system. The results from the static hard real-time GA-MP mapping shows that for lower workloads a suitable mapping solution can be achieved within a reasonable amount of time (3 hours on average). However, for larger workloads the GA-MP can take on average 100 hours to achieve a task-to-PE mapping which is only 5%–10% better than the proposed dynamic mappers.

Results also show that mapping heuristics that rely purely on communication/computation load does not scale well as the NoC size and workload increases. This work shows how the dynamic mappers behave under different CCR workloads. LWCRS and IPC group tasks together to minimise communication and to reduce the computation interference; they perform significantly better than the BN, LU and LM baselines and marginally better than the PP mapping technique under high CCR workloads. By taking into account task and flow blocking factors, better mapping decisions can be achieved. For communication and memory bound applications such as parallel video stream decoding, the performance results of the mapping techniques at higher orders of CCR are of particular interest. Potential further work in this area will be to explore combined priority assignment and mapping techniques to reduce the worst-case response time further.

References

1 Hazem Ismail Abdel Aziz Ali, Luís Miguel Pinho, and Benny Akesson. Critical-Path-First based allocation of real-time streaming applications on 2D mesh-type multi-cores. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2013, Taipei, Taiwan, August 19-21, 2013*, pages 201–208. IEEE Computer Society, 2013. doi:10.1109/RTCSA.2013.6732220.

2 Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. A Multi-objective Genetic Approach to Mapping Problem on Network-on-Chip. *J. UCS*, 12(4):370–394, 2006. doi:10.3217/jucs-012-04-0370.

3 Neil C. Audsley, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. doi:10.1049/sej.1993.0034.

4 Mohamed A. Bamakhrama and Todor P. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Autom. for Emb. Sys.*, 17(2):221–249, 2013. doi:10.1007/s10617-012-9086-x.

5 Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2-3):105–128, 2004. doi:10.1016/j.sysarc.2003.07.004.

6 Giorgio C. Buttazzo, Enrico Bini, and Yifan Wu. Partitioning Real-Time Applications Over Multicore Reservations. *IEEE Trans. Industrial Informatics*, 7(2):302–315, 2011. doi:10.1109/TII.2011.2123902.

7 William J. Dally and Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 684–689. ACM, 2001. doi:10.1145/378239.379048.

8 Ewerson Luiz de Souza Carvalho, Ney Laert Vilar Calazans, and Fernando Gehm Moraes. Dynamic Task Mapping for MPSoCs. *IEEE Design & Test of Computers*, 27(5):26–35, 2010. doi:10.1109/MDT.2010.106.

9 Michael Ditze, Peter Altenbernd, and Chris Loeser. Improving Resource Utilization for MPEG-4 Decoding in Embedded End-Devices. In Vladimir Estivill-Castro, editor, *Computer Science 2004, Twenty-Seveth Australasian Computer Science Conference (ACSC2004), Dunedin, New Zealand, January 2004*, volume 26 of *CRPIT*, pages 133–142. Australian Computer Society, 2004. URL: http://crpit.com/confpapers/CRPITV26Ditze.pdf.

10 Piotr Dziurzanski, Amit Kumar Singh, and Leandro Soares Indrusiak. Feedback-Based Admission Control for Hard Real-Time Task Allocation Under Dynamic Workload on Many-Core Systems. In Frank Hannig, João M. P. Cardoso, Thilo Pionteck, Dietmar Fey, Wolfgang Schröder-Preikschat, and Jürgen Teich, editors, *Architecture of Computing Systems – ARCS 2016 – 29th International Conference, Nuremberg, Germany, April 4-7, 2016, Proceedings*, volume 9637 of *Lecture Notes in Computer Science*, pages 157–169. Springer, 2016. doi:10.1007/978-3-319-30695-7_12.

11 ETSI. ETSI TS 101 154 v1.10.1 (2011-06) – digital video broadcasting (DVB) – specification for the use of video and audio coding in broadcasting applications based on the MPEG-2 transport stream. Technical report, European Telecommunications Standards Institute (ETSI), June 2011.

12 Mohammad Abdullah Al Faruque and Jörg Henkel. QoS-supported On-chip Communication for Multi-processors. *International Journal of Parallel Programming*, 36(1):114–139, 2008. doi:10.1007/s10766-007-0039-0.

13 Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele, and Benoît Dupont de Dinechin. Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. *Real-Time Systems*, 52(4):399–449, 2016. doi:10.1007/s11241-015-9227-y.

14 Blake Hannaford, Jacob Rosen, Diana C. W. Friedman, Hawkeye H. I. King, Phillip Roan, Lei Cheng, Daniel Glozman, Ji Ma, Sina Nia Kosari, and Lee White. Raven-II: An Open Platform for Surgical Robotics Research. *IEEE Trans. Biomed. Engineering*, 60(4):954–959, 2013. doi:10.1109/TBME.2012.2228858.

15 Jia Huang, Andreas Raabe, Christian Buckl, and Alois Knoll. A workflow for runtime adaptive task allocation on heterogeneous MPSoCs. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, pages 1119–1134. IEEE, 2011. doi:10.1109/DATE.2011.5763189.

16 Leandro Soares Indrusiak. End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. *Journal of Systems Architecture – Embedded Systems Design*, 60(7):553–561, 2014. doi:10.1016/j.sysarc.2014.05.002.

17 Leandro Soares Indrusiak, James Harbin, and Osmar Marchi dos Santos. Fast Simulation of Networks-on-Chip with Priority-Preemptive Arbitration. *ACM Trans. Design Autom. Electr. Syst.*, 20(4):56:1–56:22, 2015. doi:10.1145/2755559.

18 Damir Isovic and Gerhard Fohler. Quality Aware MPEG-2 Stream Adaptation in Resource Constrained Systems. In *16th Euromicro Conference on Real-Time Systems (ECRTS 2004), 30 June – 2 July 1004, Catania, Italy, Proceedings*, pages 23–32. IEEE Computer Society, 2004. doi:10.1109/ECRTS.2004.29.

19 Damir Isovic, Gerhard Fohler, and Liesbeth Steffens. Timing Constraints of MPEG-2 Decoding for High Quality Video: Misconceptions and Realistic Assumptions. In *15th Euromicro Conference on Real-Time Systems (ECRTS 2003), 2-4 July 2003, Porto, Portugal, Proceedings*, pages 73–82. IEEE Computer Society, 2003. doi:10.1109/EMRTS.2003.1212730.

20 Ben Kao and Hector Garcia-Molina. Deadline Assignment in a Distributed Soft Real-Time System. *IEEE Trans. Parallel Distrib. Syst.*, 8(12):1268–1274, 1997. doi:10.1109/71.640019.

21 Samarth Kaushik, Amit Kumar Singh, and Thambipillai Srikanthan. Computation and communication aware run-time mapping for NoC-based MPSoC platforms. In *IEEE 24th International SoC Conference, SOCC 2011, Taipei, Taiwan, September 26-28, 2011*, pages 185–190. IEEE, 2011. doi:10.1109/SOCC.2011.6085078.

22 Cor Meenderinck, Arnaldo Azevedo, Ben H. H. Juurlink, Mauricio Alvarez, and Alex Ramírez. Parallel Scalability of Video Decoders. *Signal Processing Systems*, 57(2):173–194, 2009. doi:10.1007/s11265-008-0256-9.

23 Hashan Roshantha Mendis, Neil C. Audsley, and Leandro Soares Indrusiak. Task allocation for decoding multiple hard real-time video streams on homogeneous NoCs. In *13th IEEE International Conference on Industrial Informatics, INDIN 2015, Cambridge, United Kingdom, July 22-24, 2015*, pages 246–251. IEEE, 2015. doi:10.1109/INDIN.2015.7281742.

24 Hashan Roshantha Mendis, Leandro Soares Indrusiak, and Neil C. Audsley. Predictability and Utilisation Trade-off in the Dynamic Management of Multiple Video Stream Decoding on Network-on-Chip based Homogeneous Embedded Multi-cores. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS'14, Versaille, France, October 8-10, 2014*, page 161. ACM, 2014. doi:10.1145/2659787.2659826.

25 Marcelo Ruaro, Guilherme A. Madalozzo, and Fernando Gehm Moraes. A hierarchical LST-based task scheduler for NoC-based MPSoCs with slack-time monitoring support. In *2015 IEEE International Conference on Electronics, Circuits, and*

Systems, ICECS 2015, Cairo, Egypt, December 6-9, 2015, pages 308–311. IEEE, 2015. doi:10.1109/ICECS.2015.7440310.

26 M. Norazizi Sham Mohd Sayuti and Leandro Soares Indrusiak. Real-time low-power task mapping in Networks-on-Chip. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2013, Natal, Brazil, August 5-7, 2013*, pages 14–19. IEEE Computer Socity, 2013. doi:10.1109/ISVLSI.2013.6654616.

27 Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms. *IEEE Trans. Industrial Informatics*, 6(4):692–707, 2010. doi:10.1109/TII.2010.2062192.

28 Zheng Shi, Alan Burns, and Leandro Soares Indrusiak. Schedulability Analysis for Real Time On-Chip Communication with Wormhole Switching. *IJERTCS*, 1(2):1–22, 2010. doi:10.4018/jertcs.2010040101.

29 Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *The 50th Annual Design Automation Conference 2013, DAC'13, Austin, TX, USA, May 29 – June 07, 2013*, pages 1:1–1:10. ACM, 2013. doi:10.1145/2463209.2488734.

30 Amit Kumar Singh, Thambipillai Srikanthan, Akash Kumar, and Wu Jigang. Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms. *Journal of Systems Architecture – Embedded Systems Design*, 56(7):242–255, 2010. doi:10.1016/j.sysarc.2010.04.007.

31 Ying Tan, Parth Malani, Qinru Qiu, and Qing Wu. Workload prediction and dynamic voltage scaling for MPEG decoding. In Fumiyasu Hirose, editor, *Proceedings of the 2006 Conference on Asia South Pacific Design Automation: ASP-DAC 2006, Yokohama, Japan, January 24-27, 2006*, pages 911–916. IEEE, 2006. doi:10.1109/ASPDAC.2006.1594802.

32 Mitchell D. Theys, Tracy D. Braun, H. J. Siegal, Anthony A. Maciejewski, and Y. K. Kwok. Mapping tasks onto distributed heterogeneous computing systems using a genetic algorithm approach. *Solutions to Parallel and Distributed Computing Problems: Lessons from Biological Sciences*, pages 135–178, 2001.

33 Wayne H. Wolf. *Multimedia Applications of Multiprocessor Systems-on-Chips*, pages 86–89. IEEE Computer Society, 2005. doi:10.1109/DATE.2005.217.

EMSBench: Benchmark and Testbed for Reactive Real-Time Systems*

Florian Kluge[†1], **Christine Rochange**[2], **and Theo Ungerer**[3]

1 **Department of Computer Science, University of Augsburg, Augsburg, Germany**
fkuau@gmx.net
2 **IRIT, Université de Toulouse, CNRS, France**
http://orcid.org/0000-0001-7257-7114
christine.rochange@irit.fr
3 **Department of Computer Science, University of Augsburg, Augsburg, Germany**
ungerer@informatik.uni-augsburg.de

Abstract

Benchmark suites for real-time embedded systems (RTES) usually contain only pure computations that are often used in this domain. They allow to evaluate computing performance, but do not reproduce the complexity and behaviour that is typical for such systems. Actual RTES have to interact with the physical environment, which is often reflected by code that is executed concurrently. In this article, we present the software package EMSBench that mimics such complex behaviour, and highlight some of its use cases. The benchmark code `ems` of EMSBench is based on the open-source engine management system (EMS) FreeEMS. Additionally, EMSBench contains a trace generator (`tg`) that provides input signals for `ems` and enables to execute `ems` close to reality. We provide detailed descriptions of the `ems`'s execution behaviour and of trace generation. EMSBench can be used as test or benchmark program to compare different hardware platforms, e.g. in terms of schedulability. Also, we use EMSBench as a benchmark for static worst-case execution time (WCET) analysis and compare these results to measurements performed on existing hardware. Our results based on the OTAWA WCET estimation tool show WCET overestimations by the static analysis from 11.9% to 41.1% depending on the complexity of the analysed functions.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Software and its engineering → Real-time systems software

Keywords and Phrases Real-time benchmark, WCET Analysis, Engine Management System

Digital Object Identifier 10.4230/LITES-v004-i002-a002

Received 2016-07-29 **Accepted** 2017-05-08 **Published** 2017-07-07

1 Motivation

Benchmark programs are widely used to assess the performance of execution platforms and development tools. In hard real-time computing domains, they also play an important role when comparing tools for WCET analysis. Widely used, for example, is the *Mälardalen Benchmark Suite* [4]. A drawback of this and similar suites is that the contained programs do not reproduce the complexity of actual real-time systems. Usually, each program is a closed system that implements only a single algorithm. In contrast, actual real-time software mostly consists of multiple interacting modules. The modules are executed concurrently and may even interfere with each other, thus mutually affecting each other's timing behaviour. Also, real-time software usually is an open system that interacts with processes in the physical world. It must react to physical events and

* Parts of this article have been published before in [10].
† Florian Kluge is now with Elektronische Fahrwerkssysteme GmbH.

its timing behaviour is heavily depending on the physical processes. Thus, there is also a need for *system benchmarks* that enable a more global assessment of real-time platforms. However, only few works exist that tackle this need.

In our view, a system benchmark could be employed in multiple manners: (1) During platform (e.g. processor + operating system) development, it may act as a test program to investigate functional aspects of a platform. When development is finished, the system benchmark will be used for (2) an experimental evaluation of important aspects of the system. This could be, e.g., the evaluation of operating system mechanisms, schedulability analysis, or response-time analysis (RTA). Finally, the program can be used as a (3) benchmark for the evaluation of WCET analysis tools. Thus, it would be integrated in the whole development process.

This work is guided by the following requirements:

Complexity. The overall behaviour of the program shall arise from the interaction of multiple modules. These modules should be scheduled independently from each other.

Reactivity. The program shall react upon external events. Reaction times should be constrained by deadlines.

Ease of Use. The program should be as easy to use as possible. Thus its potential for a widespread use would be increased.

When considering actual RTESs, it becomes obvious that there must be a tradeoff between the first two requirements and the last one. Real RTESs may employ a large variety of sensors to monitor the physical world. The software can consist of hundreds of tasks. So while being both complex and reactive, we guess such a software would never be easy to use. For the purpose of this work, we set our focus differently: We aim to have a program that requires as few as possible sensor inputs, but still exhibits as much as possible of its original dynamic behaviour.

The software package EMSBench is based on the open source EMS FreeEMS[1]. We have stripped down the FreeEMS software such that it requires only positional signals of the crankshaft as inputs. Additionally, we developed a testbed for EMSBench that generates these input signals and thus allows to execute the benchmark program. EMSBench is available for download [2] at GitHub under the conditions of the GNU GPL. The benchmark code derived from FreeEMS consists mainly of multiple interrupt service routines (ISRs) that interact among each other to control fuel injection and ignition in a spark ignition engine. Thus, EMSBench exhibits some of the complexity and reactivity of a real use-case application, even though not on an industrial scale. We balance these properties against an easy porting to and employment on other hardware platforms.

In this article, we present the software package EMSBench and examine some of its use-cases. Therefore, we provide a detailed characterisation of the code's structure and execution behaviour. This information is used to ease flow analysis in static WCET estimation which we demonstrate using the OTAWA toolset [1]. Additional use-cases are execution time measurements and the schedulability/response-time analysis for tasks. Execution time measurements are derived from realistic execution traces on two hardware platforms.

We proceed as follows: We describe FreeEMS in Section 2, and the EMSBench software package in Section 3. Possible uses of the benchmark (execution time measurements, static analysis of the worst-case execution time, analysis of task interferences) are discussed and experimented in Section 4. In Section 5, we review existing benchmarks and discuss how they compare to EMSBench. We conclude this article in Section 6.

[1] `http://freeems.org/`

2 FreeEMS

FreeEMS is an open source engine management system for four-stroke spark-ignition engines. It is designed for execution on a 16-bit microcontroller from the Freescale HCS12X family. Hitherto, it was deployed successfully to over 20 different engines. We use version 0.1.1 of FreeEMS as base for this work. Although newer versions are available, the dependencies between the individual modules are recognisable more clearly in version 0.1.1. Furthermore, the newer version is split in many more modules to be applicable more universally. For the purpose of our work using the newer version thus would only have increased the required effort, but would not have changed the outcome. FreeEMS is designed such that it can be used with different types of rotary encoders. For this work, the implementation for a 24/2 camshaft encoder from Denso for engines with intake-manifold fuel injection was chosen. In the following discussions, the term FreeEMS shall refer to this specific version of the FreeEMS software.

2.1 Spark Ignition Engine and Engine Management

Before we explain the structure of FreeEMS, let us recall the operation of a spark ignition engine. Each combustion chamber (cylinder) of the engine is terminated downwards by a movable piston. Connecting rods join all pistons to the crankshaft. Thus, the vertical movement of the pistons is converted to an axial movement of the crankshaft. The cylinder housing has at least two valve openings, one as intake for air and fuel, the other as outlet for exhaust. The valves are controlled mechanically by two camshafts. These move synchronously with the crankshaft, but only at half its speed. A spark plug is placed in each cylinder head. The inlet valve discharges into the intake system.

One vertical movement of a piston resembles one stroke, during which the crankshaft moves by 180°. A full engine cycle consists of four strokes which corresponds to a movement of the crankshaft by 720°. During the first stroke, the piston moves downwards and the inlet valve is open. The cylinder ingests air that is enriched with fuel through an injection valve. During the following upward movement of the piston during the second stroke, the mixture of air and fuel is compressed. All valves are closed now. The third stroke is initiated by spark at the spark plug. The resulting combustion leads to a downward movement of the piston. During the fourth stroke, exhaust is diverted from the cylinder through the outlet valve by the upward movement of the piston.

The opening times of the injection valves and the ignition times are controlled by the EMS. The calculations of the EMS are based on the positions of crank- and camshafts which are captured with encoders. The duration of injection is mainly influenced by the state of the throttle position, and additionally by air pressure and temperature. The ignition time, i.e. when the spark is produced, depends on the position of the crankshaft and the current speed. It is calculated such that the piston of the cylinder is near its top dead point when the air/fuel mixture ignites. Additional sensors are used to capture, e.g., temperature and air pressure in some components of the engine.

2.2 Interfacing with the Physical World

Like any other EMS [6, 23], FreeEMS utilises sensors and actuators to interact with the engine and the car's driver. The driver's command is recognised through the throttle position. Several temperature and pressure sensors provide information about the current state of the engine and the environment, e.g. monitoring of exhaust gas oxygen allows to draw conclusions about the current combustion behaviour. The data collected from these sensors mainly influences the calculation

of injection and ignition parameters, e.g. the amount of fuel that is injected in each cycle. Most sensors are connected to A/D converter (ADC) channels of the microcontroller on which FreeEMS is executed. In total, FreeEMS currently uses 11 ADC channels.

For further monitoring, and also to set outputs in time, FreeEMS uses the enhanced capture timer (ECT) of the HCS12X microcontroller. The ECT has a global counter and 8 channels that can either act as input capture (IC) or output compare (OC). The counter is incremented continuously. If a channel is configured for input capture mode, it monitors an input pin. On the configured event, e.g. if the input level on the pin toggles, the current value of the global counter is stored in a register, and an interrupt request (IRQ) is generated. If a channel is used in output compare mode, a timestamp calculated by software is stored in a register. When the global counter equals the timestamp, an output action (set high/low, toggle) is performed on a pin and also an IRQ is generated.

The dynamic behaviour of the engine is monitored with at least one rotary encoder mounted to the crank- or camshaft. Certain angular positions trigger reactions of the EMS that control actual fuel injection and ignition. The 24/2 camshaft sensor possesses 24 equally spaced primary teeth and 2 equally spaced secondary teeth that generate corresponding signals during each camshaft revolution. The movements of crank- and camshaft are coupled together. The camshaft revolves with half the speed of the crankshaft. Insofar, the 24/2 camshaft sensor is equivalent to a 12/1 crankshaft sensor. To achieve a low latency and a high accuracy of the EMS reactions, the signal lines of the encoder are connected to two IC channels of the microcontroller's ECT. If a tooth is detected, the IC channel automatically saves the current time stamp from the ECT's global counter and raises an IRQ that is handled by one of the FreeEMS ISRs.

The main actuators that are controlled by the EMS are the fuel injection valves, and the spark coil and plugs. FreeEMS performs fuel injection in a semi-sequential manner, i.e. the injection valve for any cylinder opens twice per engine cycle and injects fuel into the cylinder's intake system. The amount of fuel that is injected is regulated through the opening times of the valves. The injection valves are controlled through OC channels of the ECT. Opening and closing is performed automatically by the OC channels at times that are set by FreeEMS. To trigger fuel combustion, FreeEMS uses wasted-spark ignition: In any cylinder, two sparks are produced during each engine cycle, but actually only one triggers a combustion. This approach allows to simplify the software and hardware for ignition distribution. The ignition channels are connected to regular I/O pins of the microcontroller. The pins are controlled by ISRs that are triggered by periodic interrupt timer (PIT) units. The PITs are set anew for each new ignition. FreeEMS can generate a tachometer signal to display the engine's current revolution speed. Finally, FreeEMS provides a serial (UART) interface for monitoring and tuning of the EMS.

The minimum hardware requirements to execute FreeEMS on a microcontroller can be summed up as follows: The controller must possess at least 8 capture/compare (C/C) channels that can access a global counter. Two periodic interrupt timers are needed for further control of I/O operations. Additionally, FreeEMS uses another timer to control the execution of periodic tasks. The original implementation uses the real-time interrupt functionality of the HCS12X microcontroller. Concerning I/O, at least 25 pins are required in total. 11 pins must be accessible by an ADC unit. Each of the 8 C/C channels must be connected to a separate I/O pin, 2 for input from the rotary encoder, and 6 for control of the injection valves. Another 6 output pins are needed for driving the ignition channels.

2.3 Operation of FreeEMS

The most important relationships between FreeEMS and the underlying hardware are shown in Figure 1. Two ECT channels (0 and 1 in the figure) are configured as IC. The remaining ECT

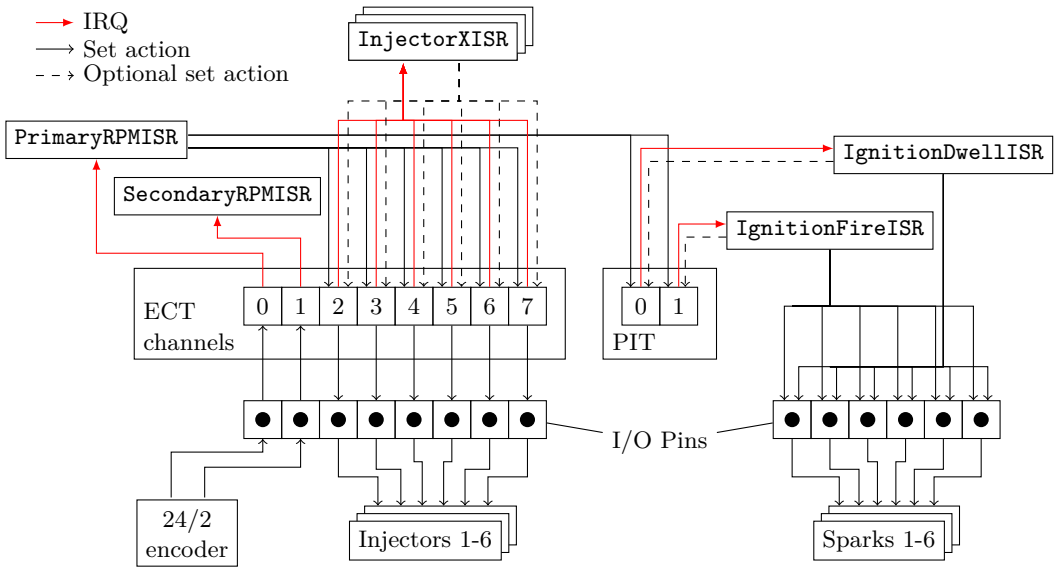

Figure 1 Important FreeEMS IRQs and their interaction with with the µC peripherals and external sensors/actuators.

channels are configured as output compare to trigger up to 6 injection valves. The IC channels monitor 24/2 encoder mounted to the camshaft. The interrupts generated by these channels are used by FreeEMS to determine the speed and position of the camshaft. Based on this and further data (derived from the sensors connected to the ADCs), times for opening the injection valves are calculated and set in the respective channels. Once an injection valve is opened through its channel's timer expiring, it triggers an ISR that reconfigures the channel for closing of the valve. Similar actions are performed for ignition, i.e. dwelling and firing of the spark plugs. Here, the PIT of the microcontroller is used. The ignition pins are completely controlled by software (`IgnitionDwellISR`, `IgnitionFireISR`).

The timing requirements for the single ISRs can be derived from their chain of effects: the `PrimaryRPMISR` and the `InjectionXISR`s are responsible for (re-)activating timers, based on calculations they perform. Obviously, these calculations must be finished before the calculated timer values expire. Similar requirements can be found in the `IgnitionXISR`s, which may prepare their next activation. Numbers about the execution times and the timer values will be presented in Section 3.4.2, where we discuss the execution behaviour, and in Section 4.1, where we present execution time measurements.

2.4 Code Structure

The code of FreeEMS can roughly be divided into functions that are performed inside a loop in the `main` function, and a number of ISRs. The modules communicate via global variables, critical sections are protected by disabling interrupts. Figure 2 gives an overview of the most important modules and their dependencies. First, it shows which modules access global data (reading/writing). Second, trigger dependencies are indicated in the figure. These can be either the setting of an interrupt timer, or the use of global flags, if functions in the main loop are concerned.

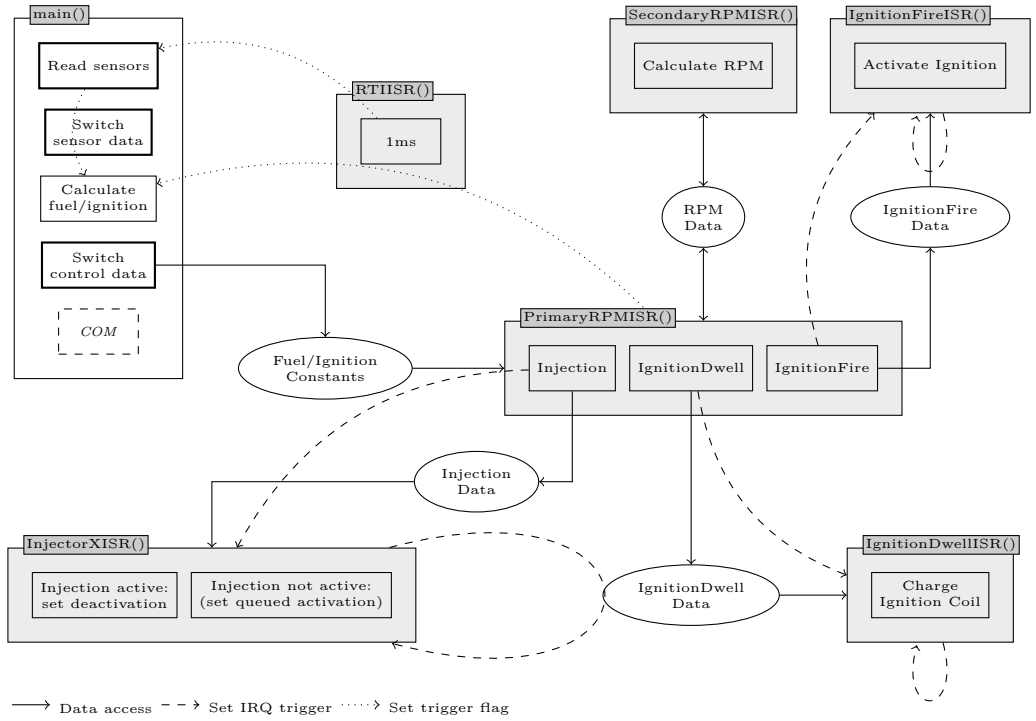

Figure 2 Structure and dependencies between ISRs and main function tasks in FreeEMS; red frames indicate critical sections during which IRQs are disabled.

2.4.1 `main`

After having initialised the whole EMS, the `main` function executes an infinite loop. Inside the loop, three tasks are performed:

1. If demanded from other modules, sensors are read. The demand is signalled by a flag in a global variable. The snapshot of all sensors is stored in a data structure. To ensure consistency of the structure and a low latency between the single readings, this task is a critical section during which interrupts are disabled. Reading of new sensor data automatically triggers the second task.
2. If new sensor data has been read, the sensor data set is switched inside a critical section with interrupts disabled.
3. Fuel and ignition parameters are calculated based on new sensor data. Both input and output data structures are allocated twice to ensure that always one structure with consistent data is available. The input data structure is filled by the previous task in the `main` function. Switching between the input (resp. output) structures is performed at the beginning (resp. end) of this task. Both operations are critical sections that are protected by disabling interrupts.
4. If new parameters have been calculated, the parameter data set is switched inside a critical section with interrupts disabled.
5. Requests that were received over the serial interface are handled. The requests are used for debugging, monitoring and tuning of the system. This task can be interrupted any time. It is not covered by the work at hand and will be ignored in the following considerations.

The code in the main function specifies low-priority tasks, as these can be interrupted any time (except during critical sections) by an ISR.

2.4.2 PrimaryRPMISR

The PrimaryRPMISR is bound to an IC channel of the platform's ECT. It is triggered by each primary pulse of the rotary encoder, i.e. it is executed 24 times each camshaft revolution. It counts the number of primary pulses since the last secondary pulse. The counter is used together with the SecondaryRPMISR to ensure synchronism between engine and EMS (see Section 2.4.3). If a loss of synchronism due to losses of primary or secondary pulses is detected, the ISR terminates immediately. Else, each second pulse, several control tasks are performed: (1) Injection times are calculated and the OC timer of the relevant injection channel is set. If another injection is already pending for the channel, the event is queued for evaluation by the InjectorXISR (see 2.4.4). (2) Times for charging of the ignition coil (IgnitionDwellISR) and triggering of the ignition (IgnitionFireISR) are calculated. If no other ignition events are pending, PITs for both events are set directly. Else, the times are put into queues that are handled by the IgnitionDwellISR resp. IgnitionFireISR.

2.4.3 SecondaryRPMISR

A second IC channel of the ECT activates the SecondaryRPMISR any time a secondary pulse from the rotary encoder is captured. The ISR's main task is to ensure synchronism between the engine and the EMS. This is achieved by checking whether the correct number of primary pulses has arrived since the last secondary pulse. For the 24/2 rotary encoder, this means that between any two secondary pulses 12 primary pulses must occur. If loss of synchronism is detected, a flag is set to signal this to the PrimaryRPMISR. Additionally, the SecondaryRPMISR calculates the current revolution speed of the engine.

2.4.4 InjectorXISR

FreeEMS supports up to 6 injection channels. Each injection channel X is handled by a separate InjectorXISR (with $X = 1, 2, \ldots, 6$), which in turn is bound to a separate OC channel of the ECT. Activation and deactivation of the injection valve is performed automatically when the associated interrupt is triggered. If injection was activated when the ISR is released, the time for deactivation is determined and the channel is configured appropriately. Upon deactivation, the ISR checks whether another injection event is queued for this channel. If necessary, it sets the timer anew.

2.4.5 IgnitionDwellISR

The IgnitionDwellISR is responsible for charging of the ignition coil. This is done by activating the power supply of the relevant ignition channel. If further *IgnitionDwell* events are queued, the associated PIT channel is restarted with a new offset, else the channel is deactivated.

2.4.6 IgnitionFireISR

Actual ignition is triggered through the IgnitionFireISR. It deactivates the power supply of the ignition coil which leads to an immediate discharge. The discharge results in a spark at the associated spark plug. If further *IgnitionFire* events are queued, the associated PIT channel is restarted with a new offset, else it is deactivated.

2.4.7 RTIISR

The RTIISR manages the execution of tasks that must be performed periodically. It implements intervals of 1 ms, 100 ms, 1 s, and 60 s. The RTIISR is released each 1/8 ms to accommodate also

for this interval if need should arise. Depending on internal counters, it decides whether a task of one of the implemented intervals must be executed. Task execution can either take place within the `RTIISR`, or the ISR sets corresponding flags to trigger the execution inside the `main` loop. In the current implementation, only each 500 ms (via the 1 ms part of the `RTIISR`) a flag is set to trigger sampling of the ADC channels in the `main` loop.

2.4.8 Further ISRs

The `TimerOverflow` ISR extends the maximum time span that can be measured with the ECT. The 16-bit counter of the ECT is configured to be incremented each 0.8 µs. An overflow occurs after ≈ 52 ms. On each release, the `TimerOverflow` ISR increments an additional 16-bit counter, thus extending the counter effectively to 32 bits. An overflow of the thus available time span of ≈ 57 min is handled appropriately.

The `LowVoltageISR` is currently only used for diagnostics and counts the frequency of low voltage events. The `ModDownCtrISR` generates a tachometer signal. These three ISRs yield only a minor contribution to the overall behaviour and therefore are ignored in the following analysis.

2.5 Interaction between ISRs

The interaction between the important ISRs and hardware units is depicted in Figure 1. The `PrimaryRPMISR` is the most important ISR in FreeEMS. Supported by the `SecondaryRPMISR`, it has exact knowledge about the crankshaft's position and speed, and thus can calculate the times for fuel injection and ignition. By setting the relevant timers, it controls the execution of the injection and ignition ISRs (continuous lines in Figure 1). If there are still pending events for the timers, the events are queued instead. In this case, the timers are set by the corresponding ISRs themselves as soon as the pending event occurs (dashed lines). Furthermore, the `PrimaryRPMISR` can trigger a recalculation of the injection and ignition parameters that is performed in the `main` loop. Reading of sensor data inside the `main` loop is triggered only by the `RTIISR` each 500 milliseconds, also leading to a recalculation of the injection and ignition parameters.

3 EMSBench

In this section, we present the software package EMSBench in detail. First, we describe the changes we made to the FreeEMS code. To execute the code successfully, signal traces must be generated, which we will discuss in Section 3.2. Furthermore, we explain how EMSBench can be ported to other hardware platforms, and discuss the timing behaviour of the single modules.

3.1 Code Changes

The FreeEMS code was adjusted to provide a preferably simple program that still exhibits a behaviour that is as close to the original one as possible. The resulting implementation will be termed `ems` in the following. Most accesses to input devices (see Section 2.2) were replaced by initialised constants. Only the input signals of the rotary encoder were kept as they influence code execution significantly. By triggering the `PrimaryRPMISR`, they also trigger the ISRs for injection and ignition indirectly. The corresponding output signals are produced and can be tapped from the associated pins. The input signals from the rotary encoder can be provided by a trace generator that emulates arbitrary driving cycles (see Section 3.2). Due to all other input values being constant, we expect no variations in the injection times. Ignition times should vary with the speed of the engine. Thus, EMSBench can only reproduce an abstract variant of an EMS's actual behaviour.

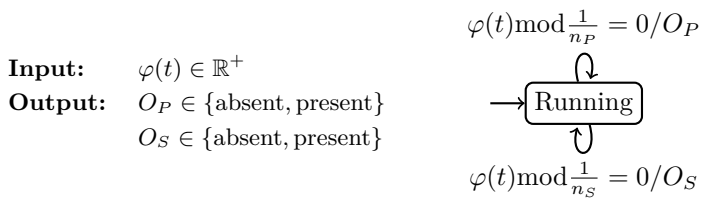

Figure 3 Behaviour of the crankshaft rotary encoder.

Portability of the `ems` to arbitrary platforms is enabled by the definition of a hardware abstraction layer (HAL). The HAL defines interfaces that are used by `ems` to control the various hardware timers and capture/compare channels. It also declares the `ems` functions that must be called by platform-specific ISRs that are part of the HAL. Currently, HAL implementations for the STM32F4-Discovery platform using an ARM Cortex-M4, and a custom FPGA-based Nios II platform are available.

3.2 Trace Generation

To run the EMSBench benchmark program in a meaningful manner, it needs signal traces that emulate the behaviour of the 24/2 camshaft sensor. We provide a trace generator that generates these signals based on driving cycles. Such driving cycles are used to perform reproducible and comparable experiments with cars. For example, the *New European Driving Cycle* [13] is widely used to estimate cars' fuel consumption. A driving cycle consists of multiple phases, which can, in turn, consist of one or multiple operations. Acceleration, initial and terminal velocity, duration, and used gear are given for each operation.

Signal generation in EMSBench is divided in two components: A preprocessor (`tgpp`) converts the driving cycle data into a crank shaft cycle. It was introduced because car movement cannot be directly related to crankshaft movement in all cases. For example, if the clutch is open, car and crankshaft move independently from each other. Actual signal generation (`tg`) executes the crankshaft cycle and generates the corresponding signals on which the `ems` must react. In the following, we first describe the model underlying the signal generation, and then its implementation.

3.2.1 Model

The relevant control signals are generated by a rotary encoder that monitors the crankshaft. The variant of FreeEMS used in this work uses a 24/2 rotary encoder that monitors the camshaft. This is equivalent to a 12/1 rotary encoder monitoring the crankshaft, which runs with twice the speed of the camshaft (see Section 2.1). Figure 3 shows a model of the rotary encoder as a real-time automaton. The current angle of the crankshaft $\varphi(t)$, measured in revolutions of the crankshaft, is used as input. Depending on the number of primary and secondary teeth, n_P and n_S, appropriate primary and secondary signals O_P and O_S are generated at certain times.

The behaviour of the crankshaft is modeled in Figure 4. This model evolves current angle $\varphi(t)$ and angular speed $\omega(t)$. If a new angular acceleration is set via the input α_N, the values for angular position and speed evolved so far are stored. The current time is used as new time offset t_0. To simplify the model, we assume that the crankshaft initially rotates with idle speed ω_I.

When combined, both models describe the signal generation by crankshaft and rotary encoder. To emulate the signal generation, we have to calculate the concrete signal times from the models and a given driving cycle. Therefore, angular position $\varphi(t)$ and speed $\omega(t)$ must be evolved based on the current angular acceleration α.

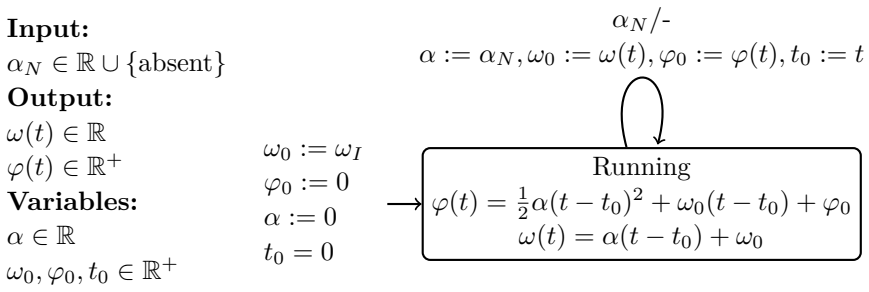

Figure 4 Behaviour of the crankshaft.

3.2.2 Preprocessor

The preprocessor `tgpp` requires two files as input. The first file contains the driving cycle that shall be emulated. The second file describes the car parameters that are needed to translate from car speed to angular speed of the crankshaft. These are the dimensions of the tyres, and the transmission ratios of the gearbox, axles and drive shaft. For idle phases, the idle speed of the engine and the acceleration with which the engine assumes this speed are given. Additionally, the file contains data that is directly passed on to signal generation. These are the number of primary teeth of the rotary encoder, and the angular distance between the secondary tooth and the preceding primary tooth. We assume that only one secondary tooth exists.

`tgpp` creates one or multiple crankshaft phases for each operation of the driving cycle. A crankshaft phase is described by its duration and the angular acceleration that acts on the crankshaft. We assume that the angular acceleration is constant during a phase. The translation of one operation into a single crankshaft phase is only possible, if the engine is idle, the car drives with constant speed, or accelerates or decelerates with closed clutch and set gear. The following operations are split into multiple crankshaft phases:

Driveaway from standstill is performed by slowly engaging the clutch. For simplification, we assume that the engine runs with its idle speed until the clutch is fully closed (first phase). The time of the full closure is calculated from the acceleration of the operation such that the car speed resembles the idle speed of the engine. In a second phase, the engine is accelerated as required by the end speed of the operation.

On a gear change the clutch is first opened, and engine speed converges to the idle speed. Then the gear is changed, and the clutch is closed again. For simplification, we assume that the opening of the clutch happens instantaneously at the beginning of the operation, and that during the operation no car speed is lost. So, concerning the crankshaft we can identify two intervals initially. For further simplification, we assume that each of these takes exactly one half of the duration of the operation. At the start of the first interval, the clutch is opened, and we assume that throttle control is free. The crankshaft speed converges to the idle speed following the idle acceleration. If the idle speed is reached before the end of the interval, we add another phase during which the angular acceleration $\alpha = 0$. During the second interval, the clutch is slowly being closed. We translate the interval to a phase where α is set such that the angular speed of the crankshaft at the end of the phase resembles the car speed of the operation, assuming that the throttle is being pressed by the driver.

Deceleration with open clutch is translated to one or two phases, depending on the initial angular speed ω_0. During the first phase, the idle acceleration acts on the crankshaft. If the crankshaft reaches its idle speed before the end of the operation, we add another phase with $\alpha = 0$ and appropriate duration to span the remaining time.

The single phases are stored as an array in a `C` source file. This file also contains additional constants that are important for signal generation, e.g. information about the rotary encoder, or idle speed. The file is then compiled, and linked with the code of the actual signal generator `tg`.

3.2.3 Signal Generation

The aim of the signal generator `tg` is to generate primary and secondary signals as they would be generated by an actual crankshaft sensor when a driving cycle is performed. `tg` is executed on an embedded platform. Its task is to evolve angular speed $\omega(t)$ and position $\varphi(t)$ according to the model presented previously, and to generate the primary and secondary signals at the appropriate times. For the calculations involved, please refer to the `tg` documentation [20]. We assume a perfect car driver who follows the operation cycle exactly. However, the throttle position is currently disregarded in EMSBench and assumed to be constant. Signal generation itself uses two OC channels of the embedded platform. The channels and associated ISRs are configured such that at the activation of the channel the pin is activated (logic 1). Simultaneously, the channel is reconfigured such that the pin is deactivated (logic 0) after a short time. Setting of the activation times for all channels is exclusively performed by the ISR for the primary channel (see Algorithm 1) when the channel is deactivated. The sole task of the ISR of the secondary channel is to set the channel's deactivation time. Additionally, the ISR for the primary channel has the following tasks:

- After each full revolution, $\varphi(t)$ is re-normalised to 0. Thus, we can keep the value of the variables in a range with high accuracy. Simultaneously, the time counter is reset and the current angular speed $\omega(t)$ is stored in ω_0.
- *Phase changes* are only performed after full revolutions, i.e. when the primary tooth at $\varphi(0)$ was released. This leads to small deviations between model and implementation (less than 1 revolution per phase change), which can affect only very short phases significantly. During a phase change, some parameters are recalculated which are used for the calculations of the succeeding activation times.
- The secondary tooth is placed between the third and the fourth primary tooth at $\phi_S \in [2\Delta_P, 3\Delta_P)$. Thus, we achieve good dispersal of the computing load of the primary ISR. This is illustrated in Figure 5, where the secondary tooth is released somewhere in the shaded area between P2 and P3. The ISR calculates on each second call (when the channel is deactivated, downward arrows) the next activation time. When it handles the first primary tooth P0, it performs additionally the re-normalisation, and, if necessary, the phase change. Furthermore, we require that for the calculation of the secondary tooth as much time is available as possible. This time is bounded by the distance between two primary teeth. Thus, the secondary calculation must be finished before the primary tooth preceding the secondary tooth is activated. Actually, the secondary tooth may be placed also between later primary teeth, but it should be avoided that the corresponding calculation coincides with renormalisation and phase change.

Similar to `ems`, the implementation of `tg` consists of two parts: A platform-specific abstraction layer provides a generic interface for managing the hardware units. All calculations are performed in a platform-independent application layer.

3.3 Adopting EMSBench

To execute `ems`, the target platform must have a timer device with at least 8 capture/compare channels that can access a common counter register. Additionally, three timers are required with a freely configurable activation interval. At least two pins must be connectable to capture/compare

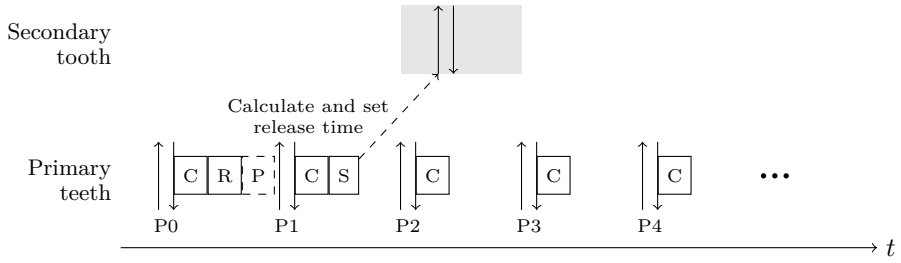

Figure 5 Timing for teeth calculations; ↑= activation of output pin and first call to ISR; ↓= deactivation of output pin and second call to ISR; C = Calculation for next primary tooth; R = Renormalisation; P = Phase change (optional); S = Calculation for secondary tooth.

channels, such that the signals of the trace generator can be routed to the correct device. For the execution of the trace generator, a timer with at least two compare channels and a common counter is required.

For all hardware-related functions we have defined a HAL that provides a generic interface for `ems` and `tg`. When porting EMSBench to a certain platform, only the relevant HAL functions have to be implemented. Due to the widespread use of 32-bit architectures, we have chosen such one for the implementation of our prototype. In a first step we have implemented EMSBench on the STM32F4-Disovery platform from ST Microelectronics. This cheap board contains a STM32F407VGT6 microcontroller, which is based on a ARM Cortex-M4 [19]. The Cortex-M4 implements the ARMv7 instruction set architecture (ISA). The microcontroller has several C/C timers with each providing up to four C/C channels. To accommodate the requirements of EMSBench, several C/C timers and their counters are configured to run synchronously with a common clock. Our second implementation of the HAL is aimed at a self-designed FPGA-based microcontroller. It uses the Nios II IP-Core from Altera and features a capture/compare timer with 8 channels which was developed in our group [9]. The main aim of this implementation was the validation of the HAL.

Porting EMSBench to new platforms requires the implementation of all HAL interface functions. Detailed instructions on how to proceed with this task can be found in the EMSBench code repository at GitHub.

3.4 Timing properties

3.4.1 Execution Scenario

`ems` is executed using the new european driving cycle [13] for trace generation. The whole cycle consists of an urban and an extra-urban driving cycle. The urban cycle takes 195 s and is repeated four times, while the extra-urban cycle takes 400 s and is performed once. In total, the cycle takes 1,180 s ($\approx$ 20 minutes). During the cycle, the revolution speed of the crankshaft ranges from 11.67 s^{-1} to 63.64 s^{-1} (700 rpm to $\approx$ 3820 rpm).

The counter frequencies of the C/C timers in both implementations were set such as to approximate the time base of the original FreeEMS implementation as closely as possible. In FreeEMS, the counter of the ECT is incremented each 0.8 µs. The same time base is also used for signal generation by the trace generator.

Algorithm 1 ISR for primary channel.

```
k ← 0                                              ▷ Global counter for primary teeth
procedure PrimaryISR
    if pin active then
        set deactivation time
        return
    else
        if k == 0 then                                              ▷ re-normalise
            ω_0 ← ω(t)
            φ_0 ← 0
            t ← 0
            if phase change pending then                        ▷ execute phase change
                α ← α_N
            end if
        end if
        calculate primary release time t_P
        set primary release time
        if k == 1 then                                  ▷ also prepare secondary channel
            calculate secondary release time t_S
            set secondary release time
        end if
        k ← (k + 1)  mod n_p
    end if
end procedure
```

3.4.2 Execution Behaviour

Most ISRs are executed recurringly according to different time bases. Based on the physical time, the `RTIISR()` is called each 125 µs, and performs actual calculations each 1 ms. All ISRs that deal directly with the engine are coupled to the time base that is generated by the rotary sensor of the crank- or camshaft. Their release frequencies during one revolution of the crankshaft are specified in Table 1. The `PrimaryRPMISR()` is released on each primary tooth, i.e. 12 times per crankshaft revolution. It performs actual work only on each second call, based on an internal counter. On odd teeth, only internal variables are advanced and debug outputs are set. On even teeth, also calculations for fuel injection and ignition are performed. The `SecondaryRPMISR()` is called once each crank revolution. Each `InjectionXISR()` is released twice, once for opening and once for closing of the injection valve. The `IgnitionDwellISR()` and `IgnitionFireISR()` are released six times per revolution, each pair activating a different ignition channel. The frequency of the crank-angle-triggered ISRs with respect to physical time depends on the current revolution speed of the crank shaft.

When the adjusted implementation is executed using the standardised driving cycle [13], no queuing of injection or ignition events can be observed. Instead, all timers are set and activated directly. Furthermore, all times are aligned such that injection and ignition is finished before the occurrence of the next `PrimaryRPMISR()`.

Figure 6 shows an exemplary execution sequence of ISRs for one fuel channel. Start times are derived from ticks of the global clock in the STM32F4-Discovery implementation. Execution on a different platform will yield similar results, if timer settings are kept similar. Due to abstraction from many inputs, most times are fixed. Only the closing of the injection valve (`Inj-L`) varies in

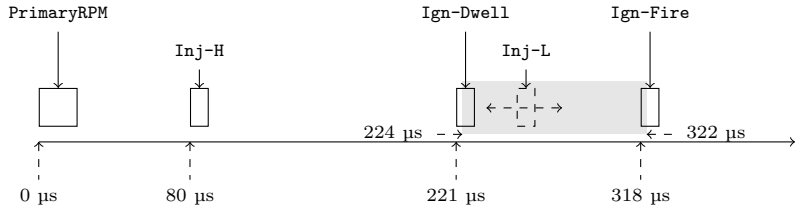

Figure 6 Exemplary sequence for one channel; Injection-Low start times vary inside the shaded interval (box widths are symbolic).

Table 1 Release frequency of crank-angle-triggered ISRs; CR frequency = based on one crankshaft revolution.

ISR	IRQ Source	Frequency	
		CR	s^{-1}
`PrimaryRPMISR()`	ECT-IC	12	144–768
`SecondaryRPMISR()`	ECT-IC	1	12–64
`InjectionXISR()`	ECT-OC	2 (×6)	24–128 (×6)
`IgnitionDwellISR()`	PIT	6	60–384
`IgnitionFireISR()`	PIT	6	60–384

time. Here, the behaviour of EMSBench deviates from a real EMS: Assuming the adjustments described in Section 3.1, one would rather expect the injection times to be constant and the ignition times to vary, as the latter clearly must depend on the engine's current speed. We could trace the depicted behaviour back to a fault in the original implementation of FreeEMS that we were not able to correct, and to some simplifying assumptions in our adjustment. As the goal of this work is to provide a benchmark that behaves *similar* to a real EMS, but not to really control an engine, we neglect this deviation.

Furthermore, the injection may overlap with the ignition event. Although this may seem strange in the first place, this behaviour is valid as injection is performed not directly into the cylinder but into the intake manifold. When an actual ignition occurs, fuel is already injected for the next cycle of the channel. From the diagram, one can observe that overlaps of ISRs probably may occur between the Injection-Low ISR and one of the ignition ISRs. This may pose a problem, as the ignition pins are managed by software inside the ISR and thus dwelling and firing could be deferred. A deeper discussion of this and similar problems can be found in Section 4.3.

4 Use of EMSBench

4.1 Execution-Time Measurements

In the following, we present the execution-time measurements that we have performed on the STM32F4-Discovery and our custom Nios II platforms. For each series of measurements, we used two identical hardware boards, one for trace generation and another to run `ems`. The counters of the C/C timers on both boards were configured to run with identical frequencies. For the `PrimaryRPMISR()`, two classes of measurements are shown: On *even teeth*, calculations for fuel and ignition are performed, while on *odd teeth* the ISR only basic management functions are executed. Similarly, the measurements for the `InjectorXISR`s distinguish the instances for opening and closing of the injection valves. The different instances of these ISRs are merged in these two classes.

Table 2 Measured execution times of ISRs and critical sections on the STM32F4-Discovery platform (clock cycles).

ISR	min	max	avg	med
`PrimaryRPMISR` (even teeth)	1403	1438	1415	1415
`PrimaryRPMISR` (odd teeth)	361	384	364	364
`SecondaryRMPISR`	275	291	275	275
`InjectorXISR` (open)	553	594	561	560
`InjectorXISR` (close)	508	537	518	516
`IgnitionDwellISR`	158	169	165	162
`IgnitionFireISR`	143	153	149	149
`RTIISR`	112	301	121	112
`main` (sample)	238	238	238	238
`main` (switch sensor data)	65	67	66	66
`main` (switch control data)	53	73	63	63

4.1.1 STM32F4-Discovery

The STM32F407 microcontroller (µC) on this platforms runs at a frequency of 168 MHz. The common clock for the timers has a period of 0.8 µs (125 kHz). Code is executed directly from on-chip Flash memory. Instruction prefetching and caching are disabled to ease the comparison of the measurements with static WCET analysis. Volatile data is stored in on-chip SRAM, and data caching is also disabled. The memory footprint of the `ems` is about 52 kB for code and about 49 kB for data.

Table 2 shows minimum, maximum, average and median execution times that were observed during one driving cycle on the STM32F4-Discovery platform. As code and data are loaded directly from Flash resp. SRAM memories, there is only a low variance in the execution times of the different ISRs. Compared to the other ISRs, the `PrimaryRPMISR` for even teeth has a very high execution time, as it performs a large number of calculations. Also, each `InjectorXISR` has to perform several calculations. The other ISRs execute only few calculations or, like the ISRs related to ignition, only set output pins, and thus have lower execution times. The observed execution time of the `SecondaryRPMISR` is mostly 275 cycles. The maximum value of 291 cycles was observed only once during the driving cycle. It represents a corner case due to error conditions like lost synchronisation between primary and secondary teeth. Concerning the `main` function, only the execution times of the critical sections during which IRQs are disabled are shown. These numbers stand for the release delay any ISR might experience.

The seemingly high variance in the execution times of the `RTIISR` is due to the different branches that can be taken during the ISR. For example, during most calls (7 out of 8), only a counter is increased. Each 8th call, i.e. each millisecond, the ISR additionally can perform periodic tasks (see Section 2.4.7). Depending on the number of periods that must be checked and the amount of work to be performed accordingly, the execution time increases. Table 3 shows the measured execution times for the different periods that are handled in `RTIISR`. Tasks with higher period include the work for lower-period tasks. As the numbers show, the seemingly high variance in the `RTIISR` execution times in Table 2 can be attributed to the different branches that are taken. Inside a single task class in `RTIISR`, the execution times are quite stable.

We must note that the execution times of the 1 s and 1 min paths differ only slightly by one cycle. This low difference is based on the structure of the assembler code that is generated by the compiler and the fetch and execution behaviour of the STM32F407 µC, and also on the fact that

```
                // 1 s code ...
80063ca:        cbnz r6, 80063f2 ; branch if 1 min task should NOT be executed
                // Now execute 1 min task ...
80063cc:        ldrh r0, [r2, #8]
80063ce:        strh r4, [r2, #14]
80063d0:        adds r3, r0, #1
80063d2:        strh r3, [r2, #8]
80063d4:        movs r4, #102 ; Execution path indicator (1 min)
80063d6:        b.n 80062e6 ; Jump to return block of function
                ...
80063f2:        movs r4, #101 ; Execution path indicator (1 s)
80063f4:        b.n 80062e6 ; Jump to return block of function
80063f6:        nop
```

Figure 7 Assembler code of 1 min part in `RTIISR`.

Table 3 Measured execution times of `RTIISR` depending on execution path (clock cycles).

Period	min	max	avg	med
125 µs	112	112	112	112
1 ms	183	183	183	183
500 ms (via 1 ms)	203	203	203	203
100 ms	243	263	243	243
1 s	300	300	300	300
1 min	301	301	301	301

no actual work is performed in the 1 min path. Concerning fetching of instruction, the processor always loads a full line of 16 bytes from Flash memory. Thus, fetching from a new line incurs a longer latency (in our case 4 cycles), while all further instructions from the same line can be fetched immediately. Any time the 1 min path is taken, the 1 s part is also executed. The relevant part of the assembler code is depicted in Figure 7. The branch instruction at address `80063ca` decides whether the 1 min part is executed (branch not taken) or not (branch taken). On the one hand, if the branch is not taken, the first two instructions are already loaded from Flash, and can directly be fetched. Only at `80063d0` another long-latency fetch from Flash is necessary. On the other hand, if the branch is taken, i.e. the 1 min part is not executed, the processor incurs a branch penalty, and also must wait for the new line (containing instructions at `80063f2` and following) being loaded from Flash. Thus, the overheads of the different paths are lying in balance.

4.1.2 Nios II

The Nios II platform is deployed to a Cyclone II FPGA on a Terasic DE2-70 development board. The processor runs with a clock frequency of 50 MHz. It comprises 32 kB of L1 data and instruction cache each. We employ the *simple capture/compare timer (SCCT)* [9] which provides a global global counter and 8 C/C channels. The global counter of the SCCT is configured to run with 125 kHz. Code and data are both stored in off-chip SDRAM. On this platform, the memory footprint is about 36 kB for code and 52 kB for data sections. The large difference of the code size compared to the STM32F4-Discovery stems mainly from the use of a different board support package.

The execution times measured on the Nios II platform can be found in Table 4. Additionally, the table also shows the measured execution times of the first execution of each ISR/function

Table 4 Measured execution times of ISRs and critical sections on the Nios II platform (clock cycles).

ISR	first	min	max	avg	med
`PrimaryRPMISR()` (even teeth)	1316	732	1422	959	886
`PrimaryRPMISR()` (odd teeth)	322	290	809	419	306
`SecondaryRMPISR()`	238	180	352	239	238
`InjectorXISR()` (open)	462	315	462	347	345
`InjectorXISR()` (close)	290	255	337	274	273
`IgnitionDwellISR()`	112	65	112	68	67
`IgnitionFireISR()`	84	64	114	66	66
`RTIISR()`	170	66	280	89	82
`main()` (sample)	174	174	242	213	210
`main()` (switch sensor data)	35	35	93	37	38
`main()` (switch control data)	37	34	73	36	37

block, when it was executed with a cold cache. The first execution of each function is not always the one with the highest execution time, as in some cases a shorter path through the function may be taken. All execution times exhibit a greater variation due to the caches used on this platform. In terms of median or average execution time, the results are comparable to those from the STM32F4-Discovery platform, even though the ratio between any two ISRs may vary. In average, the execution times are lower due to the usage of SDRAM to store the code and caches.

4.2 Static WCET Analysis

Another suggested use of EMSBench is exercising static WCET analysis techniques and tools. In this section, we briefly review the main principles of static WCET analysis, then show how it might be applied to EMSBench. This is illustrated with some preliminary results.

4.2.1 Principles of Static WCET Analysis

The building of a valid scheduling of tasks in a real-time system relies on the knowledge of each task's worst-case execution time. In a system that runs tasks in isolation (i.e. where a task executes without being delayed due to resource sharing with any other piece of software), the execution time of a task only depends on (a) the initial state of the system (e.g. the contents of cache memories), and (b) the input data set.

Measurement-based timing analysis techniques require the selection of relevant input data sets: unfortunately, when the longest possible execution time is searched, it might be difficult to determine the worst-case input data, or to show that a given input data set leads to an execution time that is close to the WCET. In addition, initializing the hardware to any possible state before performing measurements is usually infeasible, while identifying the worst-case state might be complex. Static WCET analysis techniques instead abstract input data and derive an upper bound of the execution time that is valid for any input data within a domain that might be restricted by user-provided annotations (e.g. to express the range of a sensor outputs) and for any initial hardware state. The usual method to derive this upper bound is the Implicit Path Enumeration Technique [11] that considers short segments of code (basic blocks). It maximizes the execution time of the program defined as the sum of the individual execution times of basic blocks weighted by their respective execution counts, under some constraints on the possible execution flow (e.g. loop bounds, infeasible paths). Flow constraints can be provided as user annotations [26]

and/or extracted automatically from the source or binary code [5, 12, 7]. The individual WCETs of basic blocks are derived from a model of the target hardware (this phase is often referred to as *low-level analysis*) [21, 17, 16]. The difficulty of getting detailed and reliable information on commercial platforms and to design accurate and safe models, is clearly the weak point of static WCET analysis and we were faced with this issue for the experiments we report in this paper.

Various WCET analysis tools, either commercial or academic, exist [24]. In this paper, we use the OTAWA toolset [1].

4.2.2 Methodology

In this section, we estimate WCETs considering the STM32F4-Discovery platform that features a single-core processor. If we assume a non preemptive scheduling scheme, only interrupts can interfere with tasks and impact their WCETs. Although the Cortex M4 processor does not include any standard instruction nor data cache, it features a mechanism designed to hide the latency of accesses to the Flash memory, the ART accelerator. It is based on specific small-size memory that behaves similarly to a cache memory. An interrupt routine might alter the contents of this memory and thus degrade the WCET of a task. However, interrupts are disabled during the execution of interrupt service routines, as well as during the execution of the three critical sections in the main function. As a consequence, we can safely consider that all the tasks and ISRs under analysis run in isolation.

The first step when performing the timing analysis of a task is to determine flow facts, primarily loop bounds and targets of indirect branches (used in switch-like statements). Since OTAWA does not support value analysis for the ARMv7 ISA, we had to annotate indirect branches with their possible targets. The code contains few loops which are easy to bound. The only additional flow facts that had to be specified are the direction of the two conditional branches that distinguish the even/odd case for `PrimaryRPMISR()` and the open/close case for `InjectorXISR()`.

The second step of static WCET analysis is to determine the local WCET of sequential pieces of code, i.e. basic blocks. OTAWA extracts the control flow graph (CFG) of the task under analysis from the binary code of the application using the indirect branch targets provided as flow fact annotations. Then, based on a model of the hardware architecture, it determines the worst-case execution cost of each basic block whatever the execution path before it. This model must reflect the pipeline architecture and the instruction latencies of the real hardware so that a valid abstract state of the processor after each basic block can be computed from the initial abstract state (when the fetching of the block into the pipeline starts), using the technique described in [18].

For the purpose of this paper, we have designed a model of the Cortex M4 processor featured by the STM32F4-Discovery board. This model was validated against measurements using a micro-benchmark that we designed to observe specific instruction latencies (taken branches, load and store accesses to the Flash and SDRAM memories, etc.) as well as the overhead due to the measurement process (enabling a timer, then reading it after the function under analysis has been executed). However, we did not model the ART Accelerator device used to hide part of the latency to the embedded Flash memory. Instead, we have considered the full Flash latency for each access to a new Flash line. Specific latencies for accesses to the registers of I/O devices and timers have been considered, based on their address ranges.

4.2.3 Static WCET Estimations

The WCET estimations, as well as the overestimation against the highest observed watermark (i.e. numbers given in Table 2), are reported in Table 5. It appears that the overestimation is reasonable (ranging from +11.9% to +41.1%) with respect to what is usually expected from static

Table 5 Estimated WCETs of ISRs and critical sections considering the STM32F4-Discovery platform (clock cycles).

ISR	WCET	overestimation
`PrimaryRPMISR()` (even teeth)	1695	17.9%
`PrimaryRPMISR()` (odd teeth)	542	41.1%
`SecondaryRMPISR()`	343	17.9%
`InjectorXISR()` (open)	668	12.5%
`InjectorXISR()` (close)	601	11.9%
`IgnitionDwellISR()`	228	34.9%
`IgnitionFireISR()`	199	30.1%
`RTIISR()`	343	14.0%
`main()` (sample)	304	27.7%
`main()` (switch sensor data)	88	31,3%
`main()` (switch control data)	97	32.9%

Table 6 Estimated WCETs of ISRs and critical sections considering the STM32F4-Discovery platform without ART vs. with a perfect ART (clock cycles).

ISR	without ART	with a perfect ART
`PrimaryRPMISR()` (even teeth)	1695	915
`PrimaryRPMISR()` (odd teeth)	542	254
`SecondaryRMPISR()`	343	192
`InjectorXISR()` (open)	668	375
`InjectorXISR()` (close)	601	325
`IgnitionDwellISR()`	228	118
`IgnitionFireISR()`	199	110
`RTIISR()`	343	216
`main()` (sample)	304	161
`main()` (switch sensor data)	88	54
`main()` (switch control data)	97	51

WCET analysis. However, it can still be considered as a bit high given that the target processor is very simple and time-predictable. However, not modelling the ART accelerator and assuming maximum latency for each access to the memory (either Flash or SRAM) is pessimistic and has a noticeable impact on estimated WCETs. Table 6 gives an insight into this impact by showing WCETs computed with a perfect (always hit) vs. without ART.

4.3 Interferences

An important aspect in real-time systems is a RTA, which helps to ensure that reactions happen in time and supports schedulability analysis. The response times of tasks or ISRs must not only take their execution times into account, but also possible interferences from other tasks/ISRs. In the following, we discuss the timing interferences that can occur in `ems`, and how they can influence reactions.

In the EMS implementation, the `main()` loop may interfere with the ISRs, and ISRs may interfere among each other (in terms of delaying each others execution). These interferences can

delay the execution of an ISR. In the current implementation, the following interferences can occur (all numbers refer to the STM32F4-Discovery platform):

- Any ISR can be delayed through one of the three critical sections in the `main()` loop. In the first critical section, input signals from the ADCs are sampled. In our implementation, these I/O accesses are replaced by reading constants from memory. In the other two critical sections, only buffers for input resp. injection/ignition data are switched.
- The `RTIISR()` may delay the start of any other ISR and vice versa. The `RTIISR()` is triggered according to physical time each 125 µs. All important other ISRs are bound to the crankshaft rotation. Their trigger times and intervals change over time in accordance with the current rotation speed of the crankshaft. Occasionally, their activation/execution times may overlap, resulting in the ISR triggered later being delayed.
- As already noted in Section 3.4.2, the activation time of each second instance of any `InjectorXISR()` (which pulls the output pin back to low) varies inside a certain interval (see Figure 6). If the associated timer interrupt is triggered very early in the cycle, the execution of the `InjectorXISR()` can be delayed by the `IgnitionDwellISR()`. If it is activated very late, its execution may overlap with the activation of the `IgnitionFireISR()` and thus delay actual ignition. It may even happen that the `InjectorXISR()` is activated after the `IgnitionFireISR()`, in which case the `InjectorXISR()` is delayed.

Not all delays mentioned above have the same criticality. ISRs that are bound to a C/C timer channel can cope better with an execution delay: On an IC channel, the timestamp of the relevant event was already stored by the hardware. On an OC channel, the relevant pin output was already set by the hardware in time. Here, a delay may be critical, if the corresponding ISR has to set the channel's timer anew. The `IgnitionDwellISR()` and `IgnitionFireISR()` have a higher sensitivity to execution delays, as the output pins for ignition control are set by software. This means that e.g. delaying the execution of the `IgnitionFireISR()` will result in an ignition happening later. Here, it mainly depends on the actual system (EMS+engine), whether a certain delay is acceptable. Delays that are incurred by the `RTIISR()` can only have minor effects on the behaviour of the EMS. As already specified in the original FreeEMS implementation, tasks that must be performed periodically should be included in `main()` loop, while the `RTIISR` should only be used to set activation flags. A delay of the `RTIISR()` can lead to later execution of the relevant tasks, leading to a certain jitter. However, these tasks actually have to accept that they might be interrupted any time if they execute outside a critical section, and thus possible delays must be heeded during design.

To quantify the possible delay of an ISR more clearly, consider the `PrimaryRPMISR` which has the highest WCET of all tasks. The STM32F407 µC core runs at 168 MHz. Thus, the WCET of 1695 cycles corresponds to a time of $\frac{1695}{168\,\mathrm{MHz}} \approx 10.09\,\mu\mathrm{s}$. Assuming the maximum observed revolution speed of the crankshaft ($63.64\,\mathrm{s}^{-1} \approx 3820\,\mathrm{rpm}$), this time corresponds to an angle $\alpha = 10.09\,\mu\mathrm{s} \cdot 63.64\,\mathrm{s}^{-1} \approx 0.16°$. For the other critical sections, this angle is even smaller. In [25], ignition timing is varied in the range of -41° to 10° relative to the top dead point of the piston. Thus, we infer that the above deviation α is tolerable.

5 Existing Benchmarks and Related Work

Since the initial works on WCET analysis, a growing number of programs have been used as benchmarks. Earlier works have been evaluated considering short C programs inspired by algorithms described in [15] (Fast Fourier Transform, FIR filter, array sort, Fibonacci computation, etc.) and developed at the Singapore National University. Later, these benchmarks have been included in a larger collection at Mälardalen University [4]. This collection extends the former

one with programs that exhibit more complex flow patterns, so that flow analysis techniques can be exercised. More recently, the TACLeBench collection has been released [3]. It gathers 55 re-formatted and versioned benchmarks.

The first report of using an industrial application as a WCET benchmark can be found in [8]. The application is on-board software for the Debie satellite instrument that measures impacts of small space debris or micro-meteoroids. The application consists of six tasks, including three interrupt service routines, that record information on debris hits and handle the reception of telecommands as well as the transmission of telemetry. This application is now available as open-source software[2] and has been considered in the last editions of the WCET Tool Challenge[3] for which it has been ported to Java. However, no input data generator is publicly available, which prevents from executing the benchmark to compare measured execution times to statically estimated WCETs, or from using measurement-based timing analyses.

PapaBench [14] is a benchmark built from Paparazzi, an open-source drone hardware and software project[4]. This application consists of two parts, *fbw* (fly-by-wire) that controls the drone in flight (engines and flap control, radio link with ground, IR sensor support, stabilization) and *autopilot* that controls the GPS and executes a flight plan. It is composed of about 20 tasks (including ISRs) that are statically scheduled. In contrast, the tasks (ISRs) in EMSBench are mostly triggered through external events or events that depend on the physical state of the system.

In [22], the authors argue that real-world applications might be too complex for existing academic WCET tools, mainly because of their complex flow structure. They introduce GenE, a benchmark generator, that provides flow fact annotations together with the generated code. Benchmarks are generated from code patterns that are commonly found in real-time applications. The idea is to focus on these patterns and to get rid of specific/unpredictable flow structures.

6 Conclusions

The work presented in this article was motivated by the fact that only few free benchmark programs for real-time systems exist that exhibit a behaviour similar to real applications. Widely spread benchmark suites consist usually of rather small, self-contained programs. More complex programs have so far only been reported from the aerospace domain [8, 14]. With the software package EMSBench we are undertaking a step beyond existing benchmarks to close this gap between actual real-time software and benchmark programs. In this article, we have described EMSBench and examined several of its use cases.

EMSBench is based on FreeEMS, an open source software for engine management, and was developed as a system benchmark for embedded real-time systems [10]. It consists of several ISRs and periodic tasks that are executed concurrently. Thus, it exhibits a behaviour that is significantly more *complex* than that of simple linear programs. Especially, some ISRs may interfere with other ones and delay their execution. The ISRs cannot be scheduled statically, as they must *react* upon events occurring in the phyical world with low latency. To *ease the use* of FreeEMS as a system benchmark, we have applied adjustments to the code and provide additional programs. We have removed most of the input dependencies in the `ems` part of EMSBench, and kept only the use of the crankshaft decoder. To allow for a realistic execution of `ems`, we provide a trace generator (`tg`) that emulates the behaviour of the crankshaft encoder according to arbitrary driving cycles. A HAL allows to adapt EMSBench to other hardware platforms.

[2] `http://www.tidorum.fi/debie1/`
[3] `http://www.mrtc.mdh.se/projects/WTC/`
[4] `https://wiki.paparazziuav.org`

We demonstrated the application of EMSBench with several use cases. Timing measurements were performed on two hardware platforms, the ARM Cortex-M4-based STM32F4-Discovery, and a self-designed Nios II-based FPGA microcontroller. A static WCET analysis of important parts of the EMS code from EMSBench was performed using the OTAWA toolset [1]. Reported WCET estimations considering the STM32F4-Discovery platform are above the longest observed execution times. Modelling the behaviour of hardware components that have been ignored in this first study should improve accuracy.

The results of the WCET analysis were used for an analysis of interferences that may occur between ISRs and periodic functions. Due to the low utilisation generated by the EMS on the STM32F4-Discovery platform, only minor interferences were identified. However, on platforms with less performance, the EMS will generate higher utilisation, and thus possibly more serious interferences might be found. It appears that EMSBench is a valuable benchmark for static WCET analysis tools. First of all, its structure is very similar to industrial applications that we could see in projects. Several modules (ISRs or tasks invoked in the main loop) can be analysed separately. Several of them exhibit different behaviours, depending on when they are executed (e.g. `PrimaryRPMISR()` is triggered either by an even or an odd tooth), which suggests that several scenario-related WCET values can be derived. In addition, the code contains a lot of indirect branches and a specific WCET could be computed for each possible target (e.g. for the `PrimaryRPMISR()` routine related to each injection channel, which should improve the accuracy of the overall WCET).

Due to its complexity, the use of EMSBench is not restricted to WCET benchmarking. If used with the accompanying trace generator, it can also act as a test program to evaluate other aspects of an execution platform. For example, schedulability aspects of an underlying operating system could be examined based on the results of a WCET analysis. To mimic the higher processor utilisation of industrial EMSs, some ISRs in EMSBench might be extended by code that synthetically increases the load. Such code could be based on signal processing algorithms like the Fast Fourier Transform, to, e.g. imitate software for knocking detection. Thus, higher interferences could be generated, allowing for a more challenging schedulability analysis. To allow for a more realistic execution of EMSBench, it would also be interesting to extend signal generation by a throttle signal. Thus, a higher degree of variance in the `ems`'s behaviour could be generated as the injection times would no longer be constant. Further works on EMSBench could include such extensions, but also fixing bugs and shortcomings that exist in the current version. The source code of EMSBench is available at `https://github.com/unia-sik/emsbench`. We encourage the research community to submit their own HAL implementations for EMSBench via GitHub to extend the useability of EMSBench.

References

1 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems – 8th IFIP WG 10.2 Int'l Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.

2 Benchmark program and test bed for reactive embedded systems. GitHub repository. URL: `https://github.com/unia-sik/emsbench`.

3 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASIcs*, pages 2:1–2:10. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.2.

4 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In Björn Lisper, editor, *10th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASIcs*, pages

136–146. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.136.

5 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 57–66. IEEE Computer Society, 2006. doi:10.1109/RTSS.2006.12.

6 Jeff Hartman. *How to Tune and Modify Engine Management Systems*. Motorbooks Workshop. MBI Publishing Company, 2004.

7 Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Analysing Switch-Case Code with Abstract Execution. In Francisco J. Cazorla, editor, *15th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, volume 47 of *OASIcs*, pages 85–94. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/OASIcs.WCET.2015.85.

8 Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. *European Space Agency Publications – ESA SP*, 457:307–312, 2000.

9 Florian Kluge. A Simple Capture/Compare Timer. Technical Report 2015-01, Department of Computer Science, University of Augsburg, June 2015. doi:10.13140/2.1.1251.2321.

10 Florian Kluge and Theo Ungerer. EMSBench: Benchmark und Testumgebung für reaktive Systeme. In Wolfgang A. Halang and Olaf Spinczyk, editors, *Echtzeit 2015*, Informatik Aktuell, pages 11–20. Springer, 2015. doi:10.1007/978-3-662-48611-5_2.

11 Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In Bryan Preas, editor, *Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995.*, pages 456–461. ACM Press, 1995. doi:10.1145/217474.217570.

12 Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 161–166. IEEE Computer Society, 2008. doi:10.1109/RTCSA.2008.53.

13 COUNCIL DIRECTIVE of 20 March 1970 on the approximation of the laws of the Member States on measures to be taken against air pollution by emissions from motor vehicles. Version from 01.01.2007.

14 Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. PapaBench: a Free Real-Time Benchmark. In Frank Mueller, editor, *6th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 4 of *OASIcs*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2006. doi:10.4230/OASIcs.WCET.2006.678.

15 William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992. 2nd edition.

16 Wolfgang Puffitsch. Efficient Worst-Case Execution Time Analysis of Dynamic Branch Prediction. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 152–162. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.23.

17 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. doi:10.1007/s11241-007-9032-3.

18 Christine Rochange and Pascal Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Trans. HiPEAC*, 2:222–241, 2009. doi:10.1007/978-3-642-00904-4_12.

19 STMicroelectronics. *UM1472 User Manual – Discovery kit for STM32f407/417 lines*. STMicroelectronics, November 2013.

20 Trace Generation in EMSBench. URL: https://github.com/unia-sik/emsbench/blob/master/doc/tg/tg.pdf.

21 Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004. URL: http://scidok.sulb.uni-saarland.de/volltexte/2005/466/index.html.

22 Peter Wägemann, Tobias Distler, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat. GenE: A Benchmark Generator for WCET Analysis. In Francisco J. Cazorla, editor, *15th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, volume 47 of *OASIcs*, pages 33–43. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/OASIcs.WCET.2015.33.

23 Henning Wallentowitz and Konrad Reif, editors. *Handbuch Kraftfahrzeugelektronik: Grundlagen, Komponenten, Systeme, Anwendungen*. Vieweg, Wiesbaden, 2006.

24 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

25 Javad Zareei and Amir H. Kakaee. Study and the effects of ignition timing on gasoline engine performance and emissions. *European Transport Research Review*, 5(2):109–116, 2013. doi:10.1007/s12544-013-0099-8.

26 Jakob Zwirchmayr, Pascal Sotin, Armelle Bonenfant, Denis Claraz, and Philippe Cuenot. Identifying Relevant Parameters to Improve WCET Analysis. In Heiko Falk, editor, *14th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, volume 39 of *OASIcs*, pages 93–102. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014. doi:10.4230/OASIcs.WCET.2014.93.

Per Processor Spin-Based Protocols for Multiprocessor Real-Time Systems*

Sara Afshar[1], Moris Behnam[2], Reinder J. Bril[3], and Thomas Nolte[4]

1 **Mälardalen University, Västerås, Sweden**
`sara.afshar@mdh.se`
2 **Mälardalen University, Västerås, Sweden**
`moris.behnam@mdh.se`
3 **Mälardalen University, Västerås, Sweden**
`reinder.j.bril@mdh.se`
Technische Universiteit Eindhoven, Eindhoven, The Netherlands
`r.j.bril@tue.nl`
4 **Mälardalen University, Västerås, Sweden**
`thomas.nolte@mdh.se`

Abstract

This paper investigates preemptive spin-based global resource sharing protocols for resource-constrained real-time embedded multi-core systems based on partitioned fixed-priority preemptive scheduling. We present preemptive spin-based protocols that feature (*i*) an increased schedulability ratio of task sets and reduced response jitter of tasks compared to the classical non-preemptive spin-based protocol, (*ii*) similar memory requirements for the administration of waiting tasks as for the non-preemptive protocol whilst only causing (*iii*) a minimal increase of the minimal number of required stacks per core from one to at most two, and (*iv*) strong progress guarantees to tasks. We complement these protocols with a unified worst-case response time analysis that specializes to the classical analysis for the non-preemptive protocol. The paper includes a comparative evaluation of the preemptive protocols and the non-preemptive protocol based on synthetic data.

2012 ACM Subject Classification Computer systems organization~Real-time systems, Software and its engineering~Multiprocessing / multiprogramming / multitasking, Software and its engineering~Real-time schedulability

Keywords and Phrases multiprocessor, resource sharing, spin-lock protocols

Digital Object Identifier 10.4230/LITES-v004-i002-a003

Received 2017-02-06 **Accepted** 2017-10-10 **Published** 2018-01-08

1 Introduction

In this paper, we consider industrial real-time embedded multi-core systems. These systems typically control dedicated hardware and have strict timing requirements, e.g. the system shall not only provide responses to events within well-defined intervals, but also minimizes response-time fluctuations to guarantee specified quality levels of control. Due to their embedded nature, these systems are in many cases resource constrained for cost-efficiency reasons. For industrial real-time multi-core systems, partitioned fixed-priority preemptive scheduling is the defacto standard, i.e. tasks are statically allocated to cores, as exemplified by AUTOSAR [5], which is a standard for the automotive industry. For global resource sharing, i.e. sharing of, e.g., data or memory mapped

* This work is supported by the Swedish Foundation for Strategic Research via the research program PRESS, the Swedish Knowledge Foundation and ARTEMIS Joint Undertaking project EMC2 (grant agreement 621429).

I/O between tasks that are executing on different cores, AUTOSAR prescribes spin-based global resource-access protocols, which is the focus of this paper.

Two main requirements of these systems include cost-efficiency (i.e. resource constraints) and quality of control (i.e. jitter constraints). As a result of being resource constrained, resource usage, such as the amount of memory shall be restricted. Given our focus on global resource-access protocols for industrial embedded multi-core systems, we therefore aim at a high schedulability ratio of task sets on individual cores and low memory requirements. In the context of single-core systems, the stack resource policy (SRP) [6], which provides mutual exclusive access to shared resources, allows tasks to share a single stack by preventing interleaved executions of tasks, reducing memory requirements. The multiprocessor stack resource policy (MSRP) [18] generalizes SRP from a single core to a multi-core, whilst maintaining the attractive property of allowing tasks that are executing on the same core to share a single stack. MSRP essentially provides non-nested, non-preemptive spinning (i.e. busy-waiting) and non-preemptive resource access to global resources, and assumes first-in-first-out (FIFO) queueing of tasks that are waiting for those resources. Non-preemptive spinning has as an attractive side-effect that the length of individual global resource queues (and even the sum of the lengths of the global resource queues) is bounded by the number of cores.

On the other hand, embedded systems have jitter constraints. Response time fluctuations, i.e. response jitter, of control tasks may significantly degrade the control performance and, in the worst case, make control systems unstable. Response jitter shall therefore be limited for critical control tasks. The response jitter of a control task is bounded by the difference of the worst-case and best-case response time of that task [25, 11]. Assuming (a lower bound on) the best-case response time to be independent of a global resource sharing protocol, the bound on the response jitter decreases when the worst-case response time decreases. To minimize response jitter, control tasks are typically given the highest priorities in a system.

Unfortunately, non-preemptive spinning can impose a reduction in system schedulability, since a task that is spinning on a global resource blocks tasks with a higher priority on the same core that arrive during its busy-waiting time. *Preemptive spin-based* protocols reduce the blocking time of tasks with a priority higher than the priority at which the waiting task is spinning. Moreover, non-preemptive spinning may significantly increase the worst-case response time of control tasks due to remote blocking of tasks with a lower priority than the control tasks. *preemptive spin-based protocols* can reduce the blocking time of control tasks, thereby reducing their response jitter. In this paper we focus on a set of spin-based protocols that offer low memory usage and confine the length of the global resource queues to the number of cores the same as MSRP. Further in this section, we explore preemptive spin-based protocols, and conclude the section with the contributions.

1.1 Preemptive spin-based protocols

Before presenting the specific protocol considered in this paper, we first briefly describe this field. In particular, we identified three main characteristics of preemptive spin-based protocols, being the *spin-lock priority*, the *ordering* during waiting and the *impact of preemption on ordering*. We subsequently consider memory requirements and progress guarantees in more detail.

1.1.1 Main characteristics

We use *spin-lock priority* to refer to the priority at which a task is spinning while waiting for a global resource. Assuming a fixed spin-lock priority for a spin-based protocol, we identified that there are five possibilities for tasks to use spin-lock priorities. Tasks can use a fixed spin-lock

priority (*i*) per core, (*ii*) per task, (*iii*) per resource, (*iv*) per request and (*v*) hybrid, i.e. combination of any of the previous ones.

In the literature, preemptive spin-based global resource sharing protocols typically assume that a task is spinning at the priority of the task itself, i.e. the task's "own" or "original" priority [3, 13, 22, 27, 29] and hence it can be preempted due to the arrival of any higher priority task. We classify this protocol as of type (*ii*) and refer to it as *OP* (own priority).

The second and third characteristic concerns the ordering and the impact of preemption on ordering. Traditionally, there are two main techniques to determine which task is allowed to access a global resource when multiple tasks have pending requests, i.e. *ordered* (also termed *queued*) or *unordered*. In case of *ordered*, first-in-first-out (FIFO) queueing or queueing based on the priority of tasks are most common. While using priority-ordered resource queues may cause longer delays for a set of tasks (low priority tasks), it can decrease the waiting times of higher priority tasks. In fact, Wieder and Brandenburg [29] have shown that none of the queueing techniques dominates the other.

For *ordered*, three policies are typically considered in the literature for handling tasks that are residing in a global resource queue while being preempted during spinning, being *de-queuing* [13, 22, 3, 29], *skipping* [27] and the *classic* policy upon pre-emption, i.e. a task is neither de-queued nor skipped. The two former policies are typically used in conjunction with the spin-lock priority of type (*ii*), in particular with *OP*, allowing preempting tasks on the same core to access the global resource before the preempted task. De-queuing implies that a task that is preempted while spinning on a global resource is removed from the resource queue. It will again be put in the global resource queue when it is allowed to continue spinning. As a result, it may have to wait for, i.e. may be blocked by, additional remote tasks with later requests to the same global resource. Skipping implies that the task remains in the queue, but is not amenable for selection when the global resource becomes available. As a result, it may have to wait for an additional remote task with a later request to the same global resource that has been granted the resource while it was preempted. Under the classic policy, a task remains in the global resource queue when preempted and it is granted access to the global resource when it is at the head of the queue and the resource becomes available.

In this work, we consider spin-based protocols of type (*i*), where a fixed priority level is used for spinning for all tasks that are allocated to the same core. Moreover, to be consistent with MSRP, similar to MSRP [18], we assume FIFO-ordered queueing and the *classic* policy upon pre-emption.

1.1.2 Memory Requirements

As we described previously [1], the traditional spin-based and suspension-based global resource sharing protocols can conceptually be unified by viewing a suspension-based protocol as a spin-based protocol that uses the lowest priority level on a core, i.e. a priority lower than any "original" priority of tasks on that core. We refer to a suspension-based protocol as *LP* (lowest priority) and to the non-preemptive spin-based protocol as *HP* (highest priority). The *flexible spin-lock model* (FSLM) [1] allows the selection of an arbitrary priority level in the range of *LP* to *HP*, and addressed both specific instantiations of type (*i*), e.g. *LP* and *HP*, and type (*ii*), i.e. *OP*. Next to *LP* and *HP*, it also considered *CP* (ceiling priority), i.e. the highest priority of any task on a core using a global resource. In this paper, we focus on a particular subset of spin-based protocols from FSLM, i.e. those protocols that spin at a fixed priority per core in the range $[CP, HP]$ (with a slight misuse of notation, where we refer to both the protocol and its associated spin-lock priority by means of the same identifier, e.g. *CP*). An attractive point of this subset of protocols is that at most one task at any time on a core can either have a pending request on or access to a global

resource (see Lemma 12 in [1]). Using protocols from this range in combination with FIFO-ordered queueing will confine the length of any global resource queue to m, where m is the number of cores [1]. Using priority levels other than this range from the whole spectrum, such as of LP or OP, may result in a higher number of pending requests for a global resource on a core and thus longer queue sizes. Moreover, this subset of protocols can be used with a minimal increase of the number of required stacks per core. Later, in Section 4, we show that using spin-lock priorities from the range $[CP, HP)$ requires an increase from one, for MSRP (i.e. HP) to at most two stacks. This number is significantly lower than when using LP or OP which require in the worst-case n stacks per core, where n is the number of tasks allocated to that core. We therefore leave the study of the rest of the spectrum of spin-lock priorities as future work.

1.1.3 Progress Guarantees

As described above, the policies de-queueing and skipping are typically used for handling tasks that are residing in a global resource queue while being preempted during spinning. Both policies may, however, significantly increase the remote blocking time experienced by a preempted task. Under the de-queueing policy, every time a task is preempted during spinning, it has to wait for all the tasks that have been enqueued which the task was preempted. Therefore, the task may have to wait in the worst-case for an additional amount of at most $m-1$ remote tasks when using FIFO-ordered queues. Under the skipping policy, in the worst-case, every time a task is waiting for a global resource it may get preempted by a higher priority task just before it can get access to the resource and thus it will loose the access to the next queued task. Therefore, it may be delayed for an additional global resource access on a remote core. To feature the same strong progress guarantee as non-preemptive spin-based protocols, such as MSRP [18], we use the same policy as MSRP, where we keep the task in the resource queue upon preemption and immediately grant the task access to the global resource when it becomes available. In this way, a task has to wait for at most $m-1$ remote tasks when requesting a global resource, thereby preventing extra delays that can be imposed to a task under both de-queueing and skipping.

1.2 Main contributions and outline

This paper investigates preemptive spin-based global resource sharing protocols for resource-constrained real-time embedded systems based on partitioned fixed-priority preemptive scheduling. We focus on protocols with a fixed spin-lock priority per core, where the spin-lock priorities are taken from the range $[CP, HP]$. By design, the protocols feature similar memory requirements for the administration of waiting tasks as for the non-preemptive protocol and strong progress guarantees to tasks.

This paper has five main contributions. Firstly, we prove that these protocols feature a minimal increase of the minimal number of required stacks per core from one to at most two. Secondly, we introduce a special spin-based protocol from the introduced range, denoted by $\widehat{CP}$, where we (*i*) prove that it dominates the classical non-preemptive spin-based protocol and all spin-based protocols that use spin-lock priorities between $\widehat{CP}$ and HP, (*ii*) show by means of examples that CP and $\widehat{CP}$ are incomparable. This means that $\widehat{CP}$ performs always equal to or better than HP, unlike CP. Although $\widehat{CP}$ does not dominate CP, still we show that there are cases in which $\widehat{CP}$ performs better than CP. Thirdly, we provide a unified worst-case response time analysis for these protocols that specializes to the classical analysis for the non-preemptive protocol. Moreover, we show that our new analysis provides tighter blocking bounds for CP than the analysis in [1]. Fourthly, we show that there may exist an intermediate spin-lock priority within the range $[CP, \widehat{CP}]$ that can make a task set schedulable if CP and $\widehat{CP}$ cannot, which can be found via a simple linear search. Finally, we perform a comparative evaluation of HP, CP and

$\widehat{CP}$, based on the schedulability ratio of task sets and the improvement in response times of tasks.

The remainder of this paper is organized as follows. Sections 2 summarizes related work and Section 3 presents the system model. In Section 4, we prove that the minimal number of required stacks per core for the spin-based protocols under consideration increases to at most two. Section 5 proposes a new spin-based protocol $\widehat{CP}$. We subsequently present a generalized worst-case response time analysis for the protocols in Section 6. A theoretical comparison of *HP*, *CP*, and $\widehat{CP}$ is presented in Section 7. In Section 8, a comparative evaluation of *HP*, *CP*, and $\widehat{CP}$ is presented. We conclude the paper in Section 9.

2 Related Work

A non-exhaustive amount of work has been done on spin-based resource sharing protocols. In the following we briefly present the most related synchronization protocols used for multiprocessor systems.

Mellor-Crummey and Scott [23] investigate scalable spin-based protocols to minimize the network transactions that lead to contention, with a focus on non-preemptive spin-based protocols with FIFO-ordering. This work later inspired Craig and Johnson [13, 21] to use a priority-ordered variant. Whereas Craig [21] mainly focuses on non-preemptive spin-based protocols, Johnson[13] also investigates preemptive spin-based protocols with FIFO-ordering, using the de-queueing technique upon preemptions. There are extensions of these works[22, 3] that used a preemptive version with FIFO-ordering. Another work which has used a preemptive spin-based protocol is by Takada and Sakamura [27] which is based on a skipping policy. As described above, we neither use a de-queueing technique nor a skipping technique, because they expose tasks to longer remote blocking delays.

The Multiprocessor Stack Resource Policy (MSRP) was introduced by Gai et al. [18] for partitioned systems based on a non-preemptive spin-based protocol. MSRP is an extension of the Stack Resource Policy (SRP) [6] for multiprocessors and was the first work which carried out a formal blocking analysis for a spin-based protocol. Global resource waiting queues are FIFO-ordered under this protocol.

Devi et al. [16] introduced a non-preemptive spin-based protocol for global scheduling under the EDF policy. Faggioli et al. presented the Multiprocessor Bandwidth Inheritance (M-BWI) protocol [17], an extension of the Bandwidth Inheritance (BWI) protocol, for reservation-based scheduling and preemptive spinning. Under their protocol not only a spinning task can be preempted but also lock holder tasks may be preempted which leads to longer delays for releasing a resource compared to non-preemptive resource accesses that we use. M-BWI can be used in open systems where tasks can dynamically be added or removed. The resource queues used in M-BWI are FIFO-ordered.

The Flexible Multiprocessor Locking Protocol (FMLP) introduced by Block et al. [8] combines both spin-based and suspension-based protocols. Thus, it is categorized of type per-resource spin-based protocols in our classification described in Section 1.1.1. Tasks spin non-preemptive on so-called “short resources” and suspend on so-called “long resources”. FMLP uses FIFO-ordered global resource queues and has been introduced for both partitioned and global scheduling. The partitioned FMLP, was later extended for fixed-priority scheduling in [10].

A recent work by Wieder and Brandenburg [29] has investigated both preemptive and non-preemptive spin-based protocols under four different queue handling policies: FIFO and unordered spin-based protocols, and priority ordered spin-based protocols with FIFO-ordered and unordered tie breaking. They use a de-queueing technique in order to avoid transitive arrival blocking problem which occurs in combination with FIFO-ordered queues and preemptive spin-based

protocols, where tasks spin with their original priority (which we refer to it as *OP*). Transitive arrival blocking occurs when a task waiting in a global resource queue is preempted by a higher priority task on the core that require the same global resource, thus the higher priority task has to wait for that lower priority task. This problem does not occur for the spin-lock priority levels considered in this paper, since spinning is performed at a priority level equal to or higher than the priority of any task using a global resource. Wieder et al. [29] achieve tighter blocking bounds using mixed-integer linear program (ILP) techniques to bound the maximum cumulative blocking imposed to a task. We show later in Section 6.4 how ILP can be used for the set of spin-based protocols considered in this paper. In this paper, we investigate alternative spin-lock priorities that can improve schedulability, reduce memory requirements, and reduce response jitter.

The Multiprocessor resource sharing Protocol (MrsP) is a preemptive spin-based protocol that is proposed by Burns and Wellings [12]. MrsP is an extension of PCP [26] for multiprocessor fixed-priority partitioned scheduling where each global resource on each core is associated with a ceiling that is the highest priority among the tasks that request that resource on that core. Since ceiling of resources are used as the spin-lock priority for tasks on a core, unlike the spin-based protocols considered in this paper, MrsP uses spin-lock priorities of type (*iii*) mentioned in Section 1.1.1. Another key difference of this protocol from the protocols considered in this paper is that the critical sections are preemptive. Moreover, a helping method [28] has been used for MrsP where a spinning task donates its spinning time to a task that has locked the resource but cannot proceed since it has been preempted on its core. Under this method the preempted task migrates to a core where a task is spinning to lock the same resource and access its locked resource there. Global resource queues are FIFO-based under this protocol.

3 System Model

Our system consists of m identical processors executing a set of n sporadic tasks using fixed-priority partitioned scheduling. The set of tasks allocated to a processor P_k is denoted by $\mathcal{T}_{P_k}$. Each task τ_i is presented by $< C_i, D_i, T_i >$ and consists of an infinite sequence of jobs. C_i denotes the worst-case execution time of task τ_i. T_i denotes the minimum inter-arrival time of τ_i and D_i denotes the relative deadline of τ_i. We assume constrained deadlines tasks, i.e. $D_i \leq T_i$. The priority of the task τ_i is denoted by π_i where $\pi_i \geq 1$. U_i denotes the utilization of a task τ_i and is calculated as $U_i = C_i/T_i$. We assume that a task τ_i has a priority higher than task τ_j, i.e., $\pi_i > \pi_j$, if $i > j$, e.g. $\pi_2 > \pi_1$. We assume tasks with unique priorities on each processor.

Tasks in the system may use *local* or *global* resources. Local resources are those that are accessed only by tasks on the same processor, whereas global resources are accessed by tasks on different processors. The section of a task that uses global and local resource is called global and local critical section (*gcs,lcs*), respectively. The sets of local and global resources which are accessed by tasks on a processor P_k are denoted by $\mathcal{R}^{\mathrm{L}}_{P_k}$ and $\mathcal{R}^{\mathrm{G}}_{P_k}$, respectively. Similarly, we denote the set of local and global resources that are accessed by jobs of a task τ_i as $\mathcal{RS}^{\mathrm{L}}_i$ and $\mathcal{RS}^{\mathrm{G}}_i$, respectively. Further, $Cs_{i,q}$ denotes the worst-case execution time among all requests of any job of a task τ_i for a resource R_q. Moreover, $n^{\mathrm{G}}_{i,q}$ denotes the maximum number of possible requests by any job of a task τ_i for a specific global resource R_q. The set of tasks on a processor P_k requesting access to a specific resource R_q is denoted by $\mathcal{T}_{P_k,q}$. Nested resource access is not the focus of this paper. A complete set of notations can be found in Table 1 in Appendix A.

Based on the partitioned fixed-priority scheduling schema, we categorize the delay that can be introduced to any task due to resource sharing in this paper into two general blocking notions: *(i)* priority inversion blocking (pi-blocking) [24, 26] and *(ii)* remote blocking. Pi-blocking happens due to tasks assigned on the same core. When a lower priority job on the same core is scheduled while

a higher priority task is pending but not scheduled, the lower priority task causes a pi-blocking to the higher priority task. A job of a task is pending when it has arrived but not finished. Remote blocking, on the other hand, is the type of blocking that a job of a task experiences due to waiting for obtaining a global resource that is in use by a task on a remote core. The maximum duration of remote blocking experienced by a task is referred to as spin-lock time under a spin-based protocol (see Definitions 10 and 11).

In this paper, we denote the B_i as the total pi-blocking that is imposed to a task τ_i and we exclude the spin-lock time of τ_i from this term.

We assume negligible run-time overhead for the analysis step. We will leave investigation of such overheads for th next step towards implementation of the protocol.

3.1 General Definitions

Below, we present a set of definitions which will be used in the rest of this paper.

▶ **Definition 1.** The highest priority level on a processor P_k is denoted by $\pi^{\max}_{P_k}$ as follows, $\pi^{\max}_{P_k} = \max_{\tau_i \in \mathcal{T}_{P_k}} \{\pi_i\}$. This is the spin-lock priority used for the *HP* spin-based protocol (see Section 3.4.1).

▶ **Definition 2.** Ceiling-based resource-access protocols (such as SRP) assign a ceiling to any local resource $R_l \in \mathcal{R}^{\mathrm{L}}_{P_k}$, where $ceil_{P_k}(R_l) = \max\{\pi_i |\ \tau_i \in \mathcal{T}_{P_k} \wedge R_l \in \mathcal{RS}^{\mathrm{L}}_i\}$. [6]

▶ **Definition 3.** We denote the highest local ceiling of any regular[1] local resource on a processor P_k as $\pi^{\mathrm{L}}_{P_k}$, where $\pi^{\mathrm{L}}_{P_k} = \max\{\pi_i | \tau_i \in \mathcal{T}_{P_k} \wedge \mathcal{RS}^{\mathrm{L}}_i \neq \emptyset\}$, i.e., $\pi^{\mathrm{L}}_{P_k} \in [1, \pi^{\max}_{P_k}]$.

▶ **Definition 4.** We denote the highest local ceiling of any global resource on a processor P_k as π^{G}_{Pk}, where $\pi^{\mathrm{G}}_{P_k} = \max\{\pi_i | \tau_i \in \mathcal{T}_{P_k} \wedge \mathcal{RS}^{\mathrm{G}}_i \neq \emptyset\}$, i.e., $\pi^{\mathrm{G}}_{P_k} \in [1, \pi^{\max}_{P_k}]$. This is the spin-lock priority used for the *CP* spin-based protocol (see Section 3.4.2).

▶ **Definition 5.** We denote the highest local ceiling of any resource on a processor (either local or global) P_k as $\pi^{\mathrm{LG}}_{P_k}$, where $\pi^{\mathrm{LG}}_{P_k} = \max(\pi^{\mathrm{L}}_{P_k}, \pi^{\mathrm{G}}_{P_k})$, i.e., $\pi^{\mathrm{LG}}_{P_k} \in [1, \pi^{\max}_{P_k}]$. This is the spin-lock priority used for the $\widehat{CP}$ spin-based protocol (see Section 5).

▶ **Definition 6.** When a task spins on a processor to acquire a global resource, its priority might change during spinning depending on the spin-based protocol that is used. The spin-lock priority of a spin-based protocol σ is denoted by $\pi^{\mathrm{spin}_\sigma}_{P_k}$ which denotes an arbitrary spin-lock priority level that is used for every task when it spins on a processor P_k. We simply use $\pi^{\mathrm{spin}}_{P_k}$ if we do not refer to a specific spin-based protocol. In this paper, we consider $\pi^{\mathrm{spin}}_{P_k} \in [\pi^{\mathrm{G}}_{P_k}, \pi^{\max}_{P_k}]$.

▶ **Definition 7.** We denote the spin-lock priority used by a task τ_i as π^{spin}_i. According to our system model, $\pi^{\mathrm{spin}}_i = \pi^{\mathrm{spin}}_{P_k}$ where $\{\forall \tau_i \in \mathcal{T}_{P_k} | \mathcal{RS}^{\mathrm{G}}_i \neq \varnothing, k = 1, ..., m\}$

▶ **Definition 8.** We refer to pi-blocking that is imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ by lower priority tasks on the same core that request local resources as *local blocking due to local resources* (*LBL*) and global resources as *local blocking due to global resources* (*LBG*).

▶ **Definition 9.** The LBG delay that is imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ is divided into two different sections: *(a) spin-delay* blocking of an LBG which is due to spinning of a lower priority task that can preempt τ_i and *(b)* global *resource access* blocking of an LBG which is due to non-preemptive access of a lower priority task to a global resource. Note that a LBG delay does not necessarily need to contain the spin-delay part, e.g., when the lower priority task access the resource immediately.

[1] We define a special (local) spin resource $R^{\mathrm{spin}}_{P_k}$ in Rule 21.

▶ **Definition 10.** The maximum time that any task on a processor P_k has to spin to acquire a global resource $R_q \in \mathcal{R}^{\mathrm{G}}_{P_k}$ is referred to as spin-lock time to acquire R_q and is denoted by $spin_{P_k,q}$, which is the maximum imposed remote blocking to acquire R_q.

▶ **Definition 11.** The maximum time that a task τ_i has to spin to acquire all its global resources is referred to as spin-lock time of task τ_i and is denoted by $spin_i$, which is the maximum imposed remote blocking to τ_i for acquiring all its global resources .

Under spin-based protocols usually the execution times of tasks are inflated by the spin-lock time [18, 20, 1] as presented by the following definition.

▶ **Definition 12.** The inflated execution time of a task τ_i is denoted by $\acute{C}_i$ and is calculated as $\acute{C}_i = C_i + spin_i$.

▶ **Note 13.** *By definition (see Definition 4),* $\forall \pi_i > \pi^{\mathrm{G}}_{P_k} |\ \tau_i \in \mathcal{T}_{P_k} \Longrightarrow \acute{C}_i = C_i$ *since* $spin_i = 0$.

▶ **Definition 14.** Davis et al. [15] defined: algorithm A dominates algorithm B, if all of the task sets that are schedulable according to algorithm B are also schedulable according to algorithm A, and task sets exist that are schedulable according to A, but not according to B. Moreover, algorithms A and B are incomparable, if there exist task sets that are schedulable according to algorithm A, but not according to algorithm B and vice versa. Since resource sharing protocols are part of scheduling algorithms, these definitions also apply for spin-based protocols in this paper. Based on this conclusion, if a task set is schedulable by both algorithms and the worst-case response times of tasks under spin-based protocol 1 is always smaller than or equal to under spin-based protocol 2 and there is at least one task that has a strictly smaller worst-case response time under protocol 1 compared to protocol 2, then by reducing the deadline of this task we create a new task set for which it is schedulable under protocol 1 but not 2 anymore which infers the dominance of protocol 1 over 2. Inferred similarly, spin-based protocols 1 and 2 are incomparable if a task set is schedulable under both protocols and there is a task that has a strictly smaller worst-case response time under one compared to the other and vice versa.

3.2 Resource Sharing Rules

This section presents the resource sharing rules based on FSLM [1] for any spin-based protocol with $\pi^{\mathrm{spin}}_{P_k} \in [\pi^{\mathrm{G}}_{P_k}, \pi^{\max}_{P_k}]$. The key idea is that a task τ_i waiting for a global resource, will busy wait, i.e. spin, whenever the resource is not available using a specific priority level in the aforementioned range. However, the priority level on which the task spins is fixed for a core; see Definition 6.

▶ **Rule 15.** *Local resources are handled by means of the SRP uniprocessor synchronization protocol [6].*

▶ **Rule 16.** *For each global resource, a FIFO-ordered queue is used to enqueue the tasks waiting for the related resource.*

The key idea behind using FIFO-ordered queues for global resources is to use a similar setup as the existing protocols (*HP* and *CP*), so that the comparison is feasible.

▶ **Rule 17.** *Whenever a task τ_i on a processor P_k requests a global resource that is in use by another processor, it places its request in the associated resource queue and spins. The task will spin with a priority level $\pi^{\mathrm{spin}}_{P_k}$ (Definition 6).*

▶ **Rule 18.** *When a task is granted access to its requested global resource on a processor P_k, its priority is boosted in an atomic operation to $\pi^{\max}_{P_k} + 1$, i.e., it access the resource immediately and executes non-preemptively on the core.*

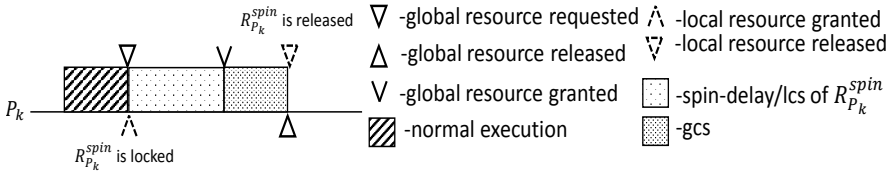

Figure 1 Spinning on P_k is viewed as locking a special local resource $R^{\text{spin}}_{P_k}$.

▶ **Rule 19.** *The priority of the task is changed to its original priority as soon as it finishes the global critical section where it becomes preemptable again.*

▶ **Rule 20.** *When the global resource becomes available (i.e. it is released), the task at the head of the global resource queue (if any) is granted the resource.*

The analysis of blocking bounds and all claims regarding the considered spin-based protocols in this paper are based on our system model and presented resource sharing rules.

3.3 View on spinning and global resource access

Under a spin-based protocol a task spins whenever it is blocked on a global resource. In classical spin-based protocols such as MSRP [18, 20] (which we refer to it as *HP*) as soon as a task is blocked on a global resource, its priority is increased to the highest on its assigned core. The task maintains this priority until it releases the resource. In this paper, we use a different approach which is increasing the priority of the task in two steps. In the first step, i.e., as soon as the task on a processor P_k request a global resource, its priority is increased to $\pi^{\text{spin}}_{P_k}$. In the second step, i.e., as soon as the task is granted access to the global resource, its priority is boosted such that it becomes non-preemptive (see Rule 18). The idea behind increasing the priority of the blocked task in two steps is to allow the high priority tasks on the core, which may even not use any (global) resource, to proceed when a lower priority task is waiting.

Conceptually, we can view spinning as accessing a "virtual" local resource (similar to a local *pseudo resource* [18]). Under a local resource sharing protocol when a task acquires a local resource its priority is raised to the ceiling of the resource, which is higher than or equal to the task's own priority. Since we only consider spin-lock priorities that are higher than or equal to the priority of any task using a global resource on the core based on our system model (Definition 6), we can treat spinning in the same way as acquiring a regular local resource by assigning the ceiling of such local resource equal to the priority during spinning. The benefit of such a view is that a local resource sharing protocol can take care of changing the priority of the task for spinning which removes the need for operating system to take such an action. Moreover, such a view simplifies validating the analysis. Having this in mind, we refer to such a virtual resource as *spin resource* and denote it for a processor P_k as $R^{\text{spin}}_{P_k}$. Rule 21 is the outcome of such a view.

▶ **Rule 21.** *For each processor P_k a special (local)* spin resource $R^{\text{spin}}_{P_k}$ *is dedicated. Every task τ_i on P_k that wants to request a global resource R_q, first locks $R^{\text{spin}}_{P_k}$ where, $ceil_{P_k}(R^{\text{spin}}_{P_k}) = \pi^{\text{spin}}_{P_k}$. The global resource access of R_q is nested within the local spin resource access of $R^{\text{spin}}_{P_k}$. The spin resource is released after the global resource is released.*

Figure 1 illustrates nesting of the global critical section within the local critical section of the special local resource $R^{\text{spin}}_{P_k}$. As can be seen, for a task the access time to the local spin resource consists of two parts: (*1*) the time that the task non-preemptively access a global resource and (*2*) the time that the task is spinning to acquire the global resource. Based on this and according to Rule 21 and the fact that the maximum time that a task may spin to acquire a global resource R_q is $spin_{P_k,q}$ (see Definition 10), hence we can reformulate Definition 9, as follows.

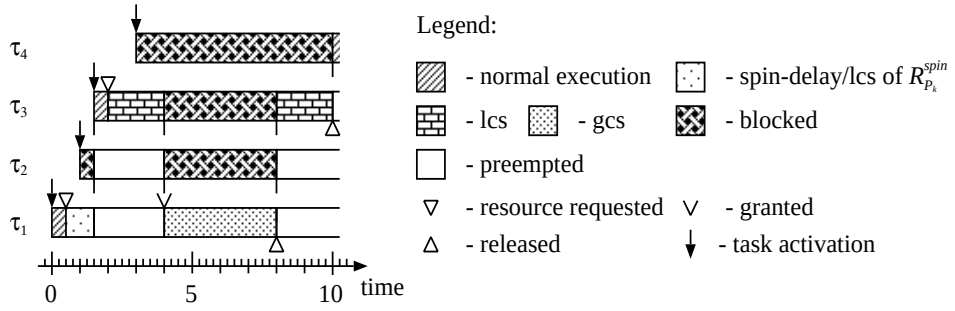

Figure 2 Task τ_4 experiences both a spin-delay and a global resource access blocking delay of an LBG.

▶ **Definition 22.** Any LBG that is imposed to a task on a processor P_k is due to non-preemptive access to a global resource of a job of a lower priority task that is nested within an access to the special spin resource $R^{\text{spin}}_{P_k}$. The maximum duration of such blocking is equal to $spin_{P_k,q} + \max_{\substack{\forall q,j:\tau_j\in\mathcal{T}_{P_k} \\ \wedge R_q\in\mathcal{RS}^{\text{G}}_j\wedge\pi_j<\pi_i}} Cs_{j,q}$.

In Figure 2 it can be seen that a task τ_4 experiences two types of blocking delay from lower priority tasks, due to both local resource access as well as global resource access. In the time interval $[3,4)\cup[8,10)$ τ_4 experiences LBL from τ_3 that has arrived earlier and has requested a local resource with a ceiling higher than or equal to τ_4's priority. In the time interval $[4,8)$ it experiences LBG from τ_1 that has arrived earlier and has requested a global resource. τ_1 has issued its request for a global resource at time 0.5, however has got blocked on the resource since the resource has been in use on a different processor. Thus when τ_1 is granted access to the global resource at time 4 it preempts τ_3 and execute its gcs non-preemptively (see Rule 18). By viewing spinning as access to a special local resource $R^{\text{spin}}_{P_k}$ on processor P_k, the time duration in which τ_1 is spinning can be viewed as an lcs duration which τ_1 access $R^{\text{spin}}_{P_k}$ with a ceiling equal to π_2.

3.4 Recap of Existing Analysis and Lemmas

In this section we briefly present the blocking analysis of the two existing spin-based protocols each of which uses a fixed spin-lock priority from the introduced spin-lock range in the system model, i.e., $[\pi^{\text{G}}_{Pk}, \pi^{\max}_{P_k}]$.

3.4.1 *HP* Spin-Based Protocol

Under *HP*, $\pi^{\text{spin}_{HP}}_{P_k} = \pi^{\max}_{P_k}$ (recall Definitions 1 and 6) which makes a task non-preemptive while spinning. This protocol has been introduced by Gai et al. [18]. Below we present the blocking delays that occur under this protocol.

LBL (Definition 8) imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ due to normal local resources is denoted as B^{L}_i and is upper bounded as follows:

$$B^{\text{L}}_i = \max_{\substack{\forall j,l:\pi_j<\pi_i\wedge\ \tau_i,\tau_j\in\tau_{P_k} \\ \wedge\ R_l\in\mathcal{RS}^{\text{L}}_j\wedge\ \pi_i\le ceil_{P_k}(R_l)}} \{Cs_{j,l}\}. \tag{1}$$

LBG (Definition 8) imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ is denoted as B^{G}_i and is upper bounded as follows:

$$B^{\text{G}}_i = \max_{\substack{\forall j,q:\pi_j<\pi_i\wedge\tau_i,\tau_j\in\mathcal{T}_{P_k} \\ \wedge R_q\in\mathcal{RS}^{\text{G}}_j}} \{Cs_{j,q} + spin_{P_k,q}\}. \tag{2}$$

The total pi-blocking imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ is denoted by B_i and is upper bounded as follows:

$$B_i = \max\{B_i^{\mathrm{L}}, B_i^{\mathrm{G}}\}. \tag{3}$$

$spin_{P_k,q}$ (Definition 10) and $spin_i$ (Definition 11) are upper bounded as follows [20]:

$$spin_{P_k,q} = \sum_{\forall P_r \neq P_k} \max_{\forall \tau_j \in \mathcal{T}_{P_r,q}} Cs_{j,q}. \tag{4}$$

$$spin_i = \sum_{\forall q: R_q \in \mathcal{RS}_i^{\mathrm{G}} \wedge \tau_i \in \mathcal{T}_{P_k}} n_{i,q}^{\mathrm{G}} \times spin_{P_k,q}. \tag{5}$$

For simplicity, under *HP* the execution times are inflated with the spin-lock time of the task. The inflated execution time of a task τ_i, $\acute{C}_i$ is calculated according to Definition 12 where $spin_i$ incorporated in it is calculated by (5).

3.4.2 *CP* Spin-Based Protocol

Under *CP* $\pi_{P_k}^{\mathrm{spin}_{CP}} = \pi_{P_k}^{\mathrm{G}}$ (recall Definitions 4 and 6) which makes a task to be non-preemptive while spinning for any task that uses a global resource on the core. This protocol has been studied previously [1]. Below we present the blocking delays that occur under this protocol.

LBL (Definition 8) imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ for *CP* is upper bounded similar as in *HP*, according to (1).

LBG (Definition 8) imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ is upper bounded as follows:

$$B_i^{\mathrm{G}} = \max_{\substack{\forall j,q: \pi_j < \pi_i \wedge \tau_i, \tau_j \in \mathcal{T}_{P_k} \\ \wedge R_q \in \mathcal{RS}_j^{\mathrm{G}}}} \{Cs_{j,q} + spin_{P_k,q} | (\pi_i \leq \pi_{P_k}^{\mathrm{G}})\}. \tag{6}$$

The total pi-blocking imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ B_i is upper bounded as follows:

$$B_i = \begin{cases} B_i^{\mathrm{L}} + B_i^{\mathrm{G}} & \text{if } \pi_i > \pi_{P_k}^{\mathrm{G}} + 1 \\ \max\{B_i^{\mathrm{L}}, B_i^{\mathrm{G}}\} & \text{if } \pi_i \leq \pi_{P_k}^{\mathrm{G}} + 1 \end{cases}. \tag{7}$$

▶ **Note 23.** *$spin_{P_k,q}$ and $spin_i$ are calculated as in (4) and (5), respectively and the inflated execution time of a task τ_i, i.e., $\acute{C}_i$ is calculated according to Definition 12.*

▶ **Note 24.** *If $\pi_{P_k}^{\mathrm{G}} = \pi_{P_k}^{\max}$ then CP is equal to HP, hence (7) and (6) specialize from (2) and (3), respectively.*

3.4.3 Recap of Useful Lemmas

Here we repeat some lemmas presented previously [1] that will be used in this paper.

▶ **Lemma 25.** *A job of a task $\tau_i \in \mathcal{T}_{P_k}$ experiences at most one LBL (recall Definition 8) from any lower priority task when SRP is used for local resource sharing (Property of SRP [6]).*

▶ **Lemma 26.** *A job of a lower priority task τ_j cannot issue any resource request after any job of a higher priority task τ_i on the same core arrives, where $\pi_i \leq \pi_i^{\mathrm{spin}}$ (Lemma 2 in [1]).*

▶ **Lemma 27.** *A job of a lower priority task τ_j can cause pi-blocking to any job of a higher priority task τ_i at most once, where $\pi_i \leq \pi_i^{\mathrm{spin}}$ (Lemma 3 in [1]).*

▶ **Note 28.** *According to Definitions 6 and 7 based on our system model,* $\{\forall\tau_i \in \mathcal{T}_{P_k} | \mathcal{RS}_i^{\mathrm{G}} \neq \varnothing \Longrightarrow \pi_i^{\mathrm{spin}} = \pi_{P_k}^{\mathrm{spin}} \geq \pi_{P_k}^{\mathrm{G}}\}$*. Since by definition, i.e., Definition 4,* $\pi_{P_k}^{\mathrm{G}}$ *is the highest priority of any task requesting a global resource therefore, for any such task* τ_i *that uses a global resource* $\pi_i \leq \pi_{P_k}^{\mathrm{G}}$*. Thus,* $\pi_i \leq \pi_i^{\mathrm{spin}}$*. As a result Lemmas 26 and 27 are valid based on our system model as well.*

4 Number of Stacks

In this section we present the maximum number of required stacks for tasks that use a spin-lock priority $\pi_{P_k}^{\mathrm{spin}}$ where $\pi_{P_k}^{\mathrm{spin}} \in [\pi_{P_k}^{\mathrm{G}}, \pi_{P_k}^{\max}]$. It has been shown by Gai et al. [18] that *HP* spin-based protocol (MSRP) allows using a single stack for all tasks on a core. Here, we show that a spin-based protocol that uses any spin-lock priority in the range $\pi_{P_k}^{\mathrm{spin}} \in [\pi_{Pk}^{\mathrm{G}}, \pi_{P_k}^{\max})$ allows using of only two stacks for scheduling all tasks on a core. For this purpose, we show that (*i*) all tasks with a priority at most $\pi_{P_k}^{\mathrm{spin}}$ can share one stack, and (*ii*) all tasks with a priority higher than $\pi_{P_k}^{\mathrm{spin}}$ and smaller than or equal to $\pi_{P_k}^{\max}$ can share another stack. It has been shown [19] that if task executions are non-interleaved then it is possible to use a single stack for scheduling all tasks. Therefore, it is enough to show that executions of tasks in group (*i*) and similarly in group (*ii*) are non-interleaved. Non-interleaved execution means that if a job of a task τ_i preempts a job of a task τ_j, the job of τ_j cannot execute before the job of τ_i is finished. Thus, we present such a property by Lemmas 30 and 31 for the execution of tasks in the two above mentioned groups. First we recapitulate the outcome of using SRP in Lemma 29 which will be used in those two lemmas.

▶ **Lemma 29.** *The local resource sharing protocol SRP ensures that once a job is started, it cannot be blocked due to a local resource until completion; it can only be preempted by higher priority jobs. [6]*

▶ **Lemma 30.** *For any task* $\tau_i \in \mathcal{T}_{P_k}$ *where* $\pi_i \leq \pi_{P_k}^{\mathrm{spin}}$ *a job of a lower priority task which is preempted by a job of* τ_i *cannot execute until the job of* τ_i *is finished.*

Proof. Proof by contradiction. Let us assume a job of τ_i preempts a job of a lower priority task τ_j with priority π_j at time t_1, and before τ_i is finished at time t_3 τ_j preempts τ_i at time t_2. This implies that at time t_2, the priority of τ_j must have been raised to or above τ_i's priority. Three situations may happen for τ_j so that its priority is raised: (*1*) τ_j accesses a local resource R_l where $ceil_{P_k}(R_l) > \pi_i$, (*2*) τ_j is granted access to the special local spin resource $\pi_{P_k}^{\mathrm{spin}}$ due to a request for a global resource R_g (see Rule 21) where $\pi_{P_k}^{\mathrm{spin}} > \pi_i$ and (*3*) τ_j accesses a global resource R_g and becomes non-preemptive (see Rule 18). In Cases (*1*) and (*2*), τ_i is blocked by τ_j at time t_2 which cannot happen according to Lemma 29, having in mind that SRP treats the local spin resource similar to any other normal local resource. In Case (*3*) the priority of τ_j has been raised to $\pi_{P_k}^{\max} + 1$ at time t_2. According to Rule 21 τ_j must first have locked the spin resource, let us assume at time t_0. τ_j cannot issue any request after arrival of τ_i at time t_a (Lemma 2 in [1]; see Lemma 26), thus $t_0 < t_a$. Further, it is obvious that $t_1 \geq t_a$ therefore it is inferred that $t_0 < t_1$. This implies that the priority of τ_j is raised to $\pi_{P_k}^{\mathrm{spin}}$ when it locks $R_{P_k}^{\mathrm{spin}}$ at time t_0 until t_2 where it gets access to R_g. Therefore, τ_i could only preempt τ_j at time t_1 if its priority has been raised higher than $\pi_{P_k}^{\mathrm{spin}}$. The only situation that τ_i's priority is raised higher than $\pi_{P_k}^{\mathrm{spin}}$ is when it is granted access to a global resource (see Rule 18). However, according to Rule 21, in order to access a global resource τ_i must have locked the local spin resource $R_{P_k}^{\mathrm{spin}}$ at time t_1 as well. This is not possible since τ_j is already holding $R_{P_k}^{\mathrm{spin}}$ at time t_1. We therefore conclude that τ_j could not have preempted τ_i at time t_2 which means that it cannot execute until τ_i is finished. ◀

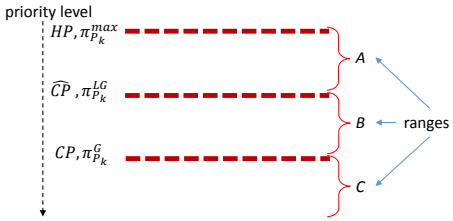

Figure 3 Priority ranges when $\pi_{P_k}^{\mathrm{G}} < \pi_{P_k}^{\mathrm{L}}$

▶ **Lemma 31.** *For any two tasks $\tau_i, \tau_j \in \mathcal{T}_{P_k}$ where $\pi_{P_k}^{\mathrm{spin}} < \pi_i, \pi_j \leq \pi_{P_k}^{\mathrm{max}}$ a job of τ_j which is preempted by a job of τ_i cannot execute until the job of τ_i is finished.*

Proof. According to Definition 6 $\pi_{P_k}^{\mathrm{spin}} \geq \pi_{P_k}^{\mathrm{G}}$ thus, by definition (see Definition 4), any task with a priority higher than $\pi_{P_k}^{\mathrm{spin}}$ does not use any global resource. therefore, τ_i and τ_j may only share local resources. Hence, this lemma can be inferred based on Lemma 29. ◀

▶ **Theorem 32.** *It is enough to use only two stacks for scheduling tasks on a processor P_k when selecting spin-based protocols with a spin-lock priority in the range $[\pi_{Pk}^{\mathrm{G}}, \pi_{P_k}^{\mathrm{max}})$.*

Proof. It is shown by Lemmas 30 and 31 that tasks of group (*i*) as well as tasks of group (*ii*) have non-interleaved executions, respectively. This implies that tasks of each group can use a single stack. However, tasks of group (*i*) cannot share the same stack with group (*ii*). This is due to the fact that if a task τ_j from group (*i*) is blocked on a global resource, i.e., it has locked the spin resource with ceiling $\pi_{P_k}^{\mathrm{spin}}$ but has not yet been granted access to the global resource, it can be preempted by a task τ_i from group (*ii*). If τ_j is granted access to the global resource before τ_i is finished, it raises its priority to $\pi_{P_k}^{\mathrm{max}} + 1$ and therefore will preempt τ_i. This means that the execution of τ_i and τ_j will not be interleaved and cannot share the same stack. We therefore conclude that all tasks on P_k can be scheduled using two stacks. ◀

5 A Special Spin-Based Protocol $\widehat{CP}$

In this section we introduce a special spin-based protocol from the range [*CP*, *HP*] which we denote by $\widehat{CP}$. This protocol uses the lowest priority for spinning such that no other task at the same core using either a local or global resource can preempt during spinning, i.e., $\pi_{P_k}^{\mathrm{spin}_{\widehat{CP}}} = \pi_{P_k}^{\mathrm{LG}}$ (recall Definitions 5 and 6). We show in Section 5.1 that $\widehat{CP}$ dominates *HP* and all spin-based protocols that use spin-lock priorities in between. In Section 5.2, we show by means of an example that *CP* and $\widehat{CP}$ are incomparable.

When $\pi_{P_k}^{\mathrm{L}} \leq \pi_{P_k}^{\mathrm{G}}$, then according to Definition 5 $\pi_{P_k}^{\mathrm{LG}} = \pi_{P_k}^{\mathrm{G}}$, having in mind that $\pi_{P_k}^{\mathrm{G}}$ is the spin-lock priority of *CP* [1]. Therefore, $\widehat{CP}$ only differs from *CP* when $\pi_{P_k}^{\mathrm{L}} > \pi_{P_k}^{\mathrm{G}}$. By this observation we introduce three priority ranges based on these two key priority levels, as illustrated in Figure 3 which later are elaborated in Sections 7 and 8. We specify these ranges as follows: (*A*) $\forall \pi \mid \pi > \pi_{P_k}^{\mathrm{LG}}$, (*B*) $\forall \pi \mid \pi_{P_k}^{\mathrm{G}} < \pi \leq \pi_{P_k}^{\mathrm{LG}}$ and (*C*) $\forall \pi \mid \pi \leq \pi_{P_k}^{\mathrm{G}}$ where π denotes an arbitrary priority level on a processor P_k.

5.1 Dominance of $\widehat{CP}$ over *HP* and In-Between Spin-Based Protocols

We show by means of Lemma 33 that, following our proposed spin-lock model, $\widehat{CP}$ dominates all spin-based protocols that use a spin-lock priority higher than π_{Pk}^{LG}.

▶ **Lemma 33.** *$\widehat{CP}$ dominates any spin-based protocol that uses a spin-priority level higher than what is used by $\widehat{CP}$, including HP.*

Proof. To prove this we assume an arbitrary spin-based protocol σ with a spin-lock priority $\pi_{P_k}^{\text{spin}_\sigma}$ higher than that of $\widehat{CP}$, i.e., $\pi_{P_k}^{\text{spin}_\sigma} > \pi_{P_k}^{\text{LG}}$ on P_k. Let us assume that a lower priority task τ_j incurs an LBG to a task τ_i. According to Definition 22, the LBG delay incurred to τ_i is divided into two parts: (a) delay due to a non-preemptive access to the global resource by τ_j and (b) delay due to the remaining access to the special local spin resource $R_{P_k}^{\text{spin}}$ with ceiling $\pi_{P_k}^{\text{spin}}$ (Definition 21). Since the access to a global resource is non-preemptive (Rule 18) hence the incurred LBG delay regarding case (a) is incurred to τ_i under any spin-based protocol, thus under both $\widehat{CP}$ and σ. However, the remaining access to the spin resource by τ_j will only block τ_i, i.e., τ_i will incur LBG delay regarding case (b) if and only if $\pi_i \leq \pi_{P_k}^{\text{spin}}$ where $\pi_{P_k}^{\text{spin}} = \pi_{P_k}^{\text{LG}}$ under $\widehat{CP}$ and $\pi_{P_k}^{\text{spin}} = \pi_{P_k}^{\text{spin}_\sigma}$ under σ spin-based protocol. Let us assume three different possible priority ranges for a task τ_i on P_k being: (i) $\pi_i \leq \pi_{P_k}^{\text{LG}}$, (ii) $\pi_{P_k}^{\text{LG}} < \pi_i \leq \pi_{P_k}^{\text{spin}_\sigma}$ and (iii) $\pi_i > \pi_{P_k}^{\text{spin}_\sigma}$. τ_i will experience the delay of type (b) using both spin-based protocols σ and $\widehat{CP}$ under the condition of case (i) since $\pi_i \leq \pi_{P_k}^{\text{LG}} < \pi_{P_k}^{\text{spin}_\sigma}$ and will not experience it using both spin-based protocols under the condition of case (iii) since $\pi_i > \pi_{P_k}^{\text{spin}_\sigma} > \pi_{P_k}^{\text{LG}}$. Thus, for a task under conditions (i) and (iii), there is no difference in using either of the spin-based protocols. Looking at the condition of case (ii), however, τ_i experiences the delay of type (b) using protocol σ but does not experience the delay using $\widehat{CP}$. This implies that the response time of τ_i is smaller when using $\widehat{CP}$ compared to when using σ. Hence, when the task set is schedulable under $\widehat{CP}$, we can make the task set unschedulable under σ by reducing the deadline of τ_i (see Definition 14). As a result, $\widehat{CP}$ dominates σ. Since σ can be any spin-based protocol where $\pi_{P_k}^{\text{LG}} < \pi_{P_k}^{\text{spin}_\sigma}$, it can be concluded that $\widehat{CP}$ dominates any spin-based protocol with a spin-lock priority higher than that of $\widehat{CP}$, i.e., also *HP* since $\pi_{P_k}^{\text{LG}} < \pi_{P_k}^{\max}$. This finishes the proof. ◀

From Lemma 33, we draw the following conclusion.

▶ **Corollary 34.** *If $\pi_{P_k}^{\text{G}} = \pi_{P_k}^{\text{LG}} < \pi_{P_k}^{\max}$, then CP dominates HP.*

5.2 $\widehat{CP}$ and CP incomparability

Next, we show by means of an example that CP and $\widehat{CP}$ are incomparable (Definition 14).

▶ **Example 35.** In the example depicted in Figure 4 which consists of two scenarios a task set is scheduled on two processors P_1 and P_2 where $\mathcal{T}_{P_1} = \tau_1, ..., \tau_6$ and $\mathcal{T}_{P_2} = \tau_7$. $T_1 = T_7 = 100$, $T_2 = 100.2$, $T_3 = T_4 = 101$ and $T_5 = T_6 = 106$, moreover, $D_1 = 9$, and the deadline for the rest of the tasks is 20. For each task τ_i, the given task specifications are specified in the format (C_i, $Cs_{i,l}$, $Cs_{i,g}$). For the following tasks these values are the same under both scenarios. τ_1: (4, 0, 3), τ_2: (1, 0, 1), τ_4: (3, 0, 0), τ_5: (1, 1, 0) and τ_6: (1, 0, 0). Note that value zero implies that the task does not use that specific resource. Under scenario *(1)*, τ_3: (2, 1, 0) and τ_7: (7, 0, 5). In scenario *(1)* τ_4 misses its deadline under $\widehat{CP}$. In scenario *(2)*, τ_3: (4, 4, 0) and τ_7: (4, 0, 1), and τ_4 misses its deadline under CP in this scenario.Given the tasks resource requesting specification under both scenarios $\pi_{P_k}^{\text{spin}_{CP}} = 2$ and $\pi_{P_k}^{\text{spin}_{\widehat{CP}}} = 5$. Since in each scenario τ_4 misses its deadline using either CP or $\widehat{CP}$, thus according to Definition 14 CP or $\widehat{CP}$ are incomparable.

6 Generalized Analysis

In this section we derive a general blocking analysis for selection of any arbitrary fixed spin-lock priority from FSLM where $\pi_{P_k}^{\text{spin}} \in [\pi_{Pk}^{\text{G}}, \pi_{P_k}^{\max}]$. To provide the maximum blocking delay to a task,

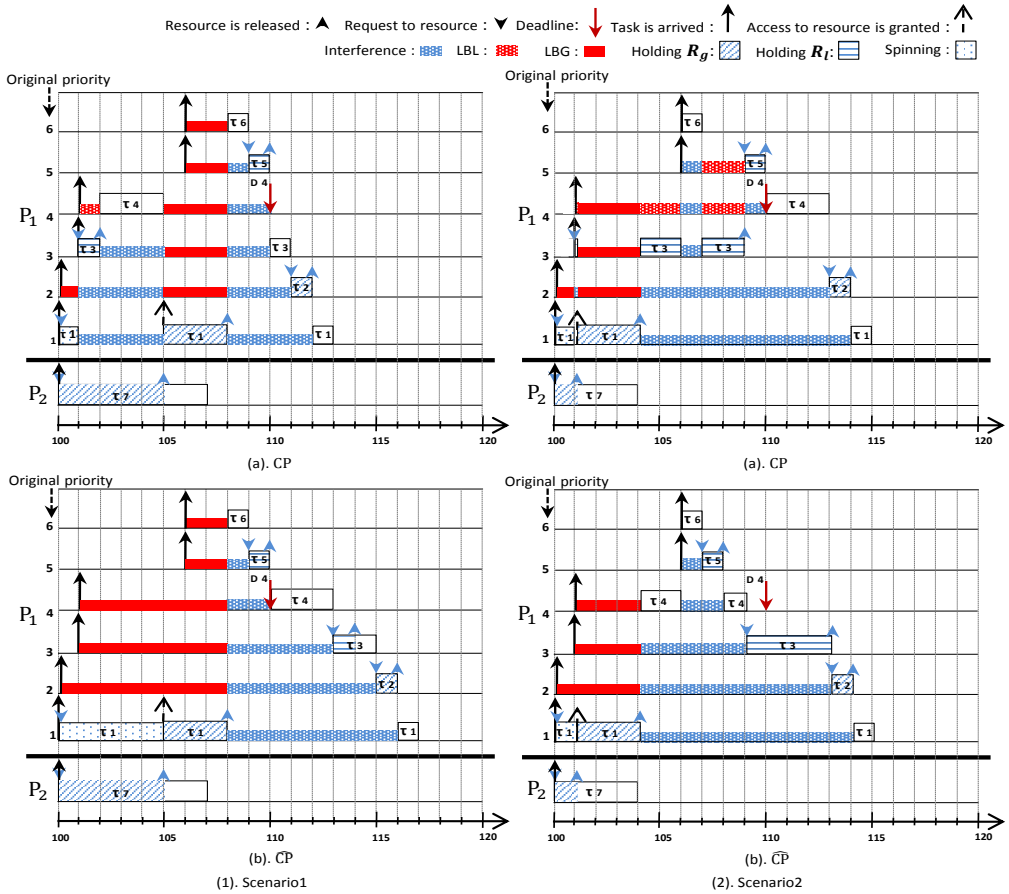

Figure 4 Incomparability of CP and $\widehat{CP}$. $\pi_{P_k}^{\text{spin}_{CP}} = 2$ and $\pi_{P_k}^{\text{spin}_{\widehat{CP}}} = 5$ in both scenarios. Scenario *(1)*: τ_4 misses its deadline under $\widehat{CP}$ but not under CP. Scenario *(2)*: τ_4 misses its deadline under CP but not under $\widehat{CP}$.

we need the maximum number of occurrence of each type of blocking i.e., LBL and LBG, to a task , and the incorporation of it with the maximum length of such blocking. To do so, we first, in Section 6.1, present the maximum possible number of blocking as well as the identified type (i.e., LBL/LBG) that a task may experience. Next, in Section 6.2 we calculate the maximum amount of such blocking using the identified number and type of blocking provided in Section 6.1.

6.1 Number and Type of Blocking

In this section we show by means of Lemmas 37 and 39, and Corollaries 36 and 38 the maximum number and type of blocking a task τ_i experiences under three determinant cases: *(i)* $\pi_{P_k}^{\text{L}} < \pi_i$, *(ii)* $\pi_i \leq \pi_{P_k}^{\text{spin}}$ and *(iii)* $\pi_{P_k}^{\text{spin}} < \pi_i \leq \pi_{P_k}^{\text{L}}$. It is enough to investigate the amount of blocking under the three aforementioned cases since other cases which appears under the assumption $\pi_{P_k}^{\text{L}} \leq \pi_{P_k}^{\text{spin}}$ falls under one of the above mentioned categories. As an example under such assumption, the cases $\pi_{P_k}^{\text{L}} < \pi_i \leq \pi_{P_k}^{\text{spin}}$ and $\pi_i \leq \pi_{P_k}^{\text{L}}$ both falls under Case *(ii)*.

Any LBG delay imposed to a task on a core P_k is due to a global resource access of a lower priority task which according to Definition 22 is nested within the access to the local spin resource $R_{P_k}^{\text{spin}}$ on P_k. According to SRP any job of a task can be blocked for at most one (outermost) local

critical section of any lower-priority task (Lemma 25). $R^{\text{spin}}_{P_k}$ is treated by SRP similar as any other regular local resource on P_k, thus, Lemma 25 can be extended to the following corollary.

▶ **Corollary 36.** *A job of any task $\tau_i \in \mathcal{T}_{P_k}$ can experience at most one LBG from any lower priority task.*

Next, we present the type and the maximum number of blocking that a task experiences under Cases *(i)* and *(ii)*.

▶ **Lemma 37.** *A job of a task $\tau_i \in \mathcal{T}_{P_k}$ experiences at most one either LBG or LBL due to normal local resources from any lower priority task under Cases* (i) $\pi^{\text{L}}_{P_k} < \pi_i$ *or* (ii) $\pi_i \leq \pi^{\text{spin}}_{P_k}$.

Proof. Under Case *(i)*, by definition, τ_i cannot experience any LBL delay due to a normal local resource. However, it still can experience blocking due to the local spin resource. Since according to Definition 22 access to the special local resource contains an access to a global resource. Thus τ_i, in the worst-cases, experiences LBG which according to Corollary 36 is at most one from any lower priority task. Therefore, the lemma is valid for this case. According to Lemma 25, τ_i experiences at most one LBL from lower priority task due to requesting local resources. Since under Case *(ii)*, $\pi_i \leq \pi^{\text{spin}}_{P_k}$ and remembering from Rule 21 that $\pi^{\text{spin}}_{P_k}$ is the ceiling of the special local spin resource on P_k, thus such an LBL to τ_i can be due to acquiring the local spin resource by a lower priority task. In other words, the imposed LBL to τ_i can be either due to a normal local resource or the spin resource on P_k. Moreover, since the access to the spin resource contains an access to a global resource, if τ_i experiences an LBL due to the local spin resource then it experiences an LBG. This concludes that τ_i experiences at most one either LBL due to a normal local resource or an LBG. This finishes the proof. ◀

To further realize the scenario of Lemma 37 let us assume that in the example in Figure 2 there exists another task τ_0 with priority lower than that of τ_2 besides τ_1 which also arrives earlier than τ_2 and uses the same local resource as τ_3. It is easy to observe that either τ_1 could issue its request for the "special" local resource $R^{\text{spin}}_{P_k}$ which has a ceiling equal to 2 and delay τ_2 upon its arrival or τ_0 could for its normal local resource with ceiling equal to 3.

Next, we present the type and the maximum number of blocking that a task experiences under Case *(iii)*. According to Corollary 36, a task τ_i experiences at most one LBG from lower priority tasks. According to Definition 22 an LBG to a task is due to non-preemptive global resource access of a lower priority task which is always nested within an access to the special spin resource $R^{\text{spin}}_{P_k}$. Unlike in Lemma 37, a task τ_i with a priority $\pi^{\text{spin}}_{P_k} < \pi_i$, cannot experience LBL due to the special spin resource $R^{\text{spin}}_{P_k}$ where $ceil_{P_k}(R^{\text{spin}}_{P_k}) = \pi^{\text{spin}}_{P_k}$ (Rule 21). However, since the access to global resource is non-preemptive (i.e., the priority of the task is raised higher than any task when the resource is granted, see Rule 18), thus, in the worst-case, τ_i experiences the resource access delay of an LBG from a lower priority task. Further, τ_i can experience at most one LBL a from a normal local resource when $\pi_i \leq \pi^{\text{L}}_{P_k}$ where according to Lemma 25 it can be at most one from any lower priority task. Therefore, the following corollary is drawn from Lemma 25 and Corollary 36.

▶ **Corollary 38.** *A job of a task $\tau_i \in \mathcal{T}_{P_k}$ experiences at most one LBL from lower priority tasks due to a normal local resource and one resource access delay of an LBG from any lower priority task under Case* (iii) $\pi^{\text{spin}}_{P_k} < \pi_i \leq \pi^{\text{L}}_{P_k}$.

The scenario of Corollary 38 can be remembered from Figure 2 where τ_4 experiences LBL from τ_3 and resource access blocking of an LBG from τ_1.

Next, we identify the set of lower priority tasks from Lemma 38 that causes LBL delay to a task τ_i under *(iii)*.

▶ **Lemma 39.** *If a job of a task $\tau_i \in \mathcal{T}_{P_k}$ experiences both an LBL and a resource access delay of an LBG then the LBL is caused by job of a lower priority task τ_l where $\pi_l > \pi^{\text{spin}}_{P_k}$.*

Proof. Let us assume that τ_i experiences LBG by the lower priority task τ_m due to a request for the global resource R_q which is issued at time t_m, and it experiences LBL by the lower priority task τ_l due to request for the local resource R_ℓ which is issued at time t_l.

According to Lemma 26 and having in mind Note 28, these requests are issued before τ_i's arrival at time t_i, thus, $t_l < t_i$ and $t_m < t_i$. However, one of the possible three following cases can be valid for t_l and t_m: *(i)* $t_l = t_m$, *(ii)* $t_l < t_m$, or *(iii)* $t_m < t_l$. According to Rule 21, τ_m first locks the special local resource $R^{\text{spin}}_{P_k}$ before locking R_q. Moreover, according to SRP, access to a local resource happens at the time of request. Thus, both τ_l and τ_m access R_ℓ and the special local resource $R^{\text{spin}}_{P_k}$ at times t_l and t_m, respectively. Further, according to Lemma 25, one LBL can happen to τ_i at any time which rules out the case *(i)*, i.e., $t_l \neq t_m$. Moreover, τ_l causing LBL to τ_i implies that it raises its priority to $ceil(R_\ell)$ at time t_l where $ceil(R_\ell) \geq \pi_i$. Therefore, no task with priority lower than that of τ_i can run in the interval $[t_l, t_i]$. This rules out case *(ii)*. Therefore, case *(iii)* is valid. τ_m first locks $R^{\text{spin}}_{P_k}$ at a time t_0 with $t_0 \leq t_m$ and raises its priority to the ceiling of this resource, i.e., $\pi^{\text{spin}}_{P_k}$ (Rule 21). Therefore, in order to τ_l be able to run at time $t_l > t_m$, it must be that $\pi_l > \pi^{\text{spin}}_{P_k}$. This finishes the proof. ◀

The scenario of Lemma 39 can be remembered from Figure 2 where τ_4 experiences both an LBL and LBG delay from τ_3 and τ_1, respectively where $\pi_3 > \pi^{\text{spin}}_{P_k} = 1$.

6.2 Amount of Blocking

In this section first we present maximum deuration of LBL and LBG delay that a task experiences by Corollary 40 and Lemmas 41 and 42. Finally, Theorem 43 concludes the total worst-case blocking delay calculation experienced by a task.

Maximum LBL duration experienced by a task is presented by the following corollary that is driven from Lemma 25.

▶ **Corollary 40.** *The maximum LBL blocking duration experienced by a task $\tau_i \in \mathcal{T}_{P_k}$ is formulated as follows.*

$$B^{\text{L}}_i = \max_{\forall j : \pi_j < \pi_i \wedge \tau_i, \tau_j \in \tau_{P_k}} B^{\text{L}}_{i,j}, \tag{8}$$

where $B^{\text{L}}_{i,j}$ is denoted as the maximum LBL delay duration imposed to a task $\tau_i \in \mathcal{T}_{P_k}$ by a local lower priority task τ_j and is calculated according to SRP specification [6] as below:

$$B^{\text{L}}_{i,j} = \max_{\substack{\forall l : R_l \in \mathcal{RS}^{\text{L}}_j \\ \wedge \pi_j < \pi_i \leq ceil_{P_k}(R_l)}} \{Cs_{j,l}\}. \tag{9}$$

In the following, we present the maximum LBG duration experienced by a task from a lower priority task.

▶ **Lemma 41.** *The maximum LBG blocking duration experienced by a task $\tau_i \in \mathcal{T}_{P_k}$ from any job of a local lower priority task τ_j using a spin-based protocol where $\pi^{\text{spin}}_{P_k} \geq \pi^{\text{G}}_{P_k}$ is denoted as $B^{\text{G}}_{i,j}(\pi^{\text{spin}}_{P_k})$ and is calculated as follows.*

$$B^{\text{G}}_{i,j}(\pi^{\text{spin}}_{P_k}) = \max_{\substack{\forall q : R_q \in \mathcal{RS}^{\text{G}}_j \\ \wedge \pi_j < \pi_i}} \left(Cs_{j,q} + \begin{cases} spin_{P_k,q} & \text{if } \pi_i \leq \pi^{\text{spin}}_{P_k} \\ 0 & \text{otherwise} \end{cases} \right), \tag{10}$$

where $spin_{P_k,q} = \sum_{\forall P_r \neq P_k} \max_{\forall \tau_j \in \mathcal{T}_{P_r,q}} \{Cs_{j,q}\}$ as in (4).

Proof. Since LBG is a type of pi-blocking (see Definition 8), thus, according to Lemma 27 (and having in mind Note 28), any job of a lower priority task τ_j can cause LBG to any job of a higher priority task τ_i at most once. Further according to Definition 22 the maximum duration of such a delay is $spin_{P_k,q} + \max_{\substack{\forall q,j:\tau_j \in \mathcal{T}_{P_k} \\ \wedge R_q \in \mathcal{RS}_j^{\mathrm{G}} \wedge \pi_j < \pi_i}} Cs_{j,q}$. However, the access to the special local resource $R_{P_k}^{\mathrm{spin}}$ by τ_j where $ceil_{P_k}(R_{P_k}^{\mathrm{spin}}) = \pi_{P_k}^{\mathrm{spin}}$ can preempt τ_i only if $\pi_i \leq \pi_{P_k}^{\mathrm{spin}}$. Based on this (10) is derived. ◀

Next, we present the maximum LBG duration experienced by a task from all lower priority tasks.

▶ **Lemma 42.** *The maximum LBG blocking duration experienced by a task τ_i from local lower priority tasks using a spin-based protocol where $\pi_{P_k}^{\mathrm{spin}} \geq \pi_{P_k}^{\mathrm{G}}$ is denoted as $B_i^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}})$ and is calculated as follows.*

$$B_i^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}}) = \max_{\forall j:\pi_j < \pi_i \wedge \tau_i, \tau_j \in \mathcal{T}_{P_k}} \{B_{i,j}^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}})\}, \tag{11}$$

where $B_{i,j}^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}})$ is calculated according to (10).

Proof. Follows immediately from Corollary 36 and Lemma 41. ◀

▶ **Theorem 43.** *The worst-case total pi-blocking experienced by a task $\tau_i \in \mathcal{T}_{P_k}$ when $\pi_{P_k}^{\mathrm{G}} \leq \pi_{P_k}^{\mathrm{spin}} \leq \pi_{P_k}^{\max}$ is denoted by $B_i(\pi_{P_k}^{\mathrm{spin}})$ and is calculated as follows.*

$$B_i(\pi_{P_k}^{\mathrm{spin}}) = \begin{cases} \begin{cases} a & \text{if } \pi_{P_k}^{\mathrm{L}} < \pi_i \\ b & \text{if } \pi_i \leq \pi_{P_k}^{\mathrm{L}} \end{cases} & \text{if } \pi_{P_k}^{\mathrm{L}} \leq \pi_{P_k}^{\mathrm{G}} \\ \begin{cases} c & \text{if } \pi_{P_k}^{\mathrm{L}} < \pi_i \\ d & \text{if } \pi_{P_k}^{\mathrm{spin}} < \pi_i \leq \pi_{P_k}^{\mathrm{L}} \\ e & \text{if } \pi_i \leq \pi_{P_k}^{\mathrm{spin}} \end{cases} & \text{if } \pi_{P_k}^{\mathrm{G}} < \pi_{P_k}^{\mathrm{L}} \end{cases} \tag{12}$$

where,

$$a = c = B_i^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}}), \tag{13}$$

$$b = e = \max(B_i^{\mathrm{L}}, B_i^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}})), \tag{14}$$

$$d = \max\left(\max_{\substack{\forall j:\pi_j < \pi_i \\ \wedge \pi_{P_k}^{\mathrm{spin}} < \pi_j \\ \wedge \tau_j \in \mathcal{T}_{P_k}}} \{B_{i,j}^{\mathrm{L}}\} + B_i^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}}), \max_{\substack{\forall j:\pi_j < \pi_i \\ \wedge \pi_j \leq \pi_{P_k}^{\mathrm{spin}} \\ \wedge \tau_j \in \mathcal{T}_{P_k}}} \{B_{i,j}^{\mathrm{L}}\} \right), \tag{15}$$

and $B_{i,j}^{\mathrm{L}}$, B_i^{L} and $B_i^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}})$ are calculated according to (9), (8) and (11).

Proof. We prove the calculation of the terms a, b, c, d and e under clauses *(a)*, *(b)*, *(c)*, *(d)* and *(e)*, respectively.

Proof of Clauses *(a)* and *(c)*: when $\pi_{P_k}^{\mathrm{L}} < \pi_i$ a task τ_i cannot experience any LBL due to normal local resources, thus, $B_i^{\mathrm{L}} = 0$. However, according to Corollary 36, τ_i can experience at most one LBG delay which its maximum duration according to Lemma 42 is (13).

Proof of Clauses *(b)* and *(e)*: according to the conditions of Clause *(b)* $\forall \tau_i | \pi_i \leq \pi^{\mathrm{L}}_{P_k}$, it is valid that $\pi_i \leq \pi^{\mathrm{G}}_{P_k}$ since $\pi^{\mathrm{L}}_{P_k} \leq \pi^{\mathrm{G}}_{P_k}$ under this clause. Further, since $\pi^{\mathrm{G}}_{P_k} \leq \pi^{\mathrm{spin}}_{P_k}$ (Definition 6) thus, $\pi_i \leq \pi^{\mathrm{spin}}_{P_k}$ under the conditions of Clause *(b)*, which is the same as the condition of Clause *(e)*. According to Lemma 37, τ_i experiences at most one either LBL or LBG from lower priority tasks when $\pi_i \leq \pi^{\mathrm{spin}}_{P_k}$. Thus, both terms *(b)* and *(e)* are inferred based on Corollary 40 and Lemma 42 which introduce the maximum amount of such blocking.

Proof of Clause *(d)*: we assume that the term d is constructed by three elements λ_1, γ and λ_2 such that $\lambda_1 = \max_{\substack{\forall j:\pi^{\mathrm{spin}}_{P_k} < \pi_j \\ \wedge \tau_j \in \mathcal{T}_{P_k}}} \{B^{\mathrm{L}}_{i,j}\}$, $\gamma = B^{\mathrm{G}}_i(\pi^{\mathrm{spin}}_{P_k})$ and $\lambda_2 = \max_{\substack{\forall j:\pi_j \leq \pi^{\mathrm{spin}}_{P_k} \\ \wedge \tau_j \in \mathcal{T}_{P_k}}} \{B^{\mathrm{L}}_{i,j}\}$. We construct three cases *(i)*, *(ii)* and *(iii)* for each of which we present a worst-case blocking delay that is imposed to τ_i under this clause.

Case *(i)*: since under the condition of Clause *(d)* $\pi^{\mathrm{spin}}_{P_k} < \pi_i \leq \pi^{\mathrm{L}}_{P_k}$, thus, according to Corollary 38 and Lemma 39, τ_i experience in the worst-case both a resource access delay of an LBG and an LBL from a lower priority task τ_j where $\pi^{\mathrm{spin}}_{P_k} < \pi_j$. By considering Lemma 42 and (9), this leads to τ_i experience at most a blocking equal to $\lambda_1 + \gamma$.

Case *(ii)*: on the other hand, it is true that according to Lemma 25 a task can experience at most one LBL from any lower priority task. By looking at (8), it is easy to see that the maximum LBL imposed to τ_i can be rewritten as $\max(\lambda_1, \lambda_2)$.

Case *(iii)*: furthermore, according to Corollary 36, it is also true that a task can experience at most one LBG from any lower priority task which based on Lemma 42 the maximum of such delay is the term γ.

All the three aforementioned cases are valid. However, the only way to find the maximum imposed delay to a task is to find the one that gives rise to the maximum blocking imposed to τ_i since they are overlapping cases. According to Lemma 25 the blocking delay derived under case *(i)* and case *(ii)* cannot both be imposed to τ_i. Similarly, according to Corollary 36 the blocking delay derived under case *(i)* and case *(iii)* cannot both be imposed to τ_i as well. Further, occurrence of blocking delay of case *(ii)* and *(iii)* is case *(i)*. Therefore, to find the maximum delay imposed to τ_i we present β that gives the maximum delay imposed under each of the three cases where, $\beta = \max(\lambda_1 + \gamma, \gamma, \max(\lambda_1, \lambda_2)) = \max(\lambda_1 + \gamma, \max(\lambda_1, \lambda_2))$. Let us assume two scenarios: *(1)* $\lambda_1 < \lambda_2$ and *(2)* $\lambda_2 \leq \lambda_1$. Under scenario *(1)*, $\beta = \max(\lambda_1 + \gamma, \lambda_2)$ which is the same as term d in (12). On the other hand, under scenario *(2)* $\beta = \lambda_1 + \gamma$. However, since under this scenario it is derived that $\lambda_2 \leq \lambda_1 + \gamma$, thus, $\max(\lambda_1 + \gamma, \lambda_2)$, i.e., term d, gives similar result as β here. As a result both scenarios *(1)* and *(2)* can be presented with the term d. This finishes the proof. ◀

It is easy to observe that the total pi-blocking to a task $\tau_i \in \mathcal{T}_{P_k}$, i.e., $B_i(\pi^{\mathrm{spin}}_{P_k})$ in (16) presented by Corollary 44 gives similar terms as in (12) under those conditions.

▶ **Corollary 44.** *$B_i(\pi^{\mathrm{spin}}_{P_k})$ for a task $\tau_i \in \mathcal{T}_{P_k}$ is presented as follows.*

$$B_i(\pi^{\mathrm{spin}}_{P_k}) = \max \left(\max_{\substack{\forall j:\pi_j < \pi_i \\ \wedge \pi^{\mathrm{spin}}_{P_k} < \pi_j \\ \wedge \tau_j \in \mathcal{T}_{P_k}}} \{B^{\mathrm{L}}_{i,j}\} + B^{\mathrm{G}}_i(\pi^{\mathrm{spin}}_{P_k}), \max_{\substack{\forall j:\pi_j < \pi_i \\ \wedge \pi_j \leq \pi^{\mathrm{spin}}_{P_k} \\ \wedge \tau_j \in \mathcal{T}_{P_k}}} \{B^{\mathrm{L}}_{i,j}\} \right), \tag{16}$$

▶ **Note 45.** *Similar to HP and CP, $spin_{P_k,q}$, $spin_i$ are calculated as in (4) and (5), respectively and the inflated execution time of a task τ_i, i.e., $\acute{C}_i$ is calculated according to Definition 12.*

6.3 Tighter Bounds under *CP*

In this section, we show by means of Lemma 46 that the new analysis for calculating the blocking terms given in (16) provides tighter bounds for *CP* compared to the analysis using (7), i.e. as given in [1].

▶ **Lemma 46.** *Maximum blocking imposed to any task τ_i under CP spin-based protocol, gives tighter bounds using (16) compared to the blocking analysis given using (7), i.e. as given in [1]in [1].*

Proof. We define the maximum blocking under *CP* that is presented in Section 3.4.2 by (7) as B_i^{old} and the maximum blocking under *CP* that is calculated by (16) as B_i^{new}.

By looking at (7), B_i^{old} is calculated differently, as presented by the following clauses that depend on the priority π_i of τ_i and the spin-priority $\pi_{P_k}^{\text{G}}$:

(i) if $\pi_{P_k}^{\text{G}} + 1 < \pi_i$ then $B_i^{\text{old}} = \acute{a} + \acute{b}$,
(ii) if $\pi_i = \pi_{P_k}^{\text{G}} + 1$ then $B_i^{\text{old}} = \max(\acute{a}, \acute{b})$,
(iii) if $\pi_i \leq \pi_{P_k}^{\text{G}}$ then $B_i^{\text{old}} = \max(\acute{a}, \tilde{b})$,

where $\acute{a}$ is the same as (8), i.e., $\acute{a} = \max_{\substack{\forall l: R_l \in \mathcal{RS}_j^{\text{L}} \\ \wedge \pi_j < \pi_i \leq ceil_{P_k}(R_l)}} \{Cs_{j,l}\}$, $\acute{b} = \max_{\substack{\forall j,q: \pi_j < \pi_i \\ \wedge \tau_i, \tau_j \in \mathcal{T}_{P_k} \wedge R_q \in \mathcal{RS}_j^{\text{G}}}} Cs_{j,q}$ and $\tilde{b} = \max_{\substack{\forall j,q: \pi_j < \pi_i \\ \wedge \tau_i, \tau_j \in \mathcal{T}_{P_k} \wedge R_q \in \mathcal{RS}_j^{\text{G}}}} \{Cs_{j,q} + spin_{P_k,q}\}$.

We investigate each clause separately. From (16) let us assume $B_i^{\text{new}} = \max(a + b, c)$ where $a = \max_{\substack{\forall j: \pi_j < \pi_i \\ \wedge \pi_{P_k}^{\text{G}} < \pi_j \wedge \tau_j \in \mathcal{T}_{P_k}}} \{B_{i,j}^{\text{L}}\}$, $b = \max_{\forall j: \pi_j < \pi_i \wedge \tau_i, \tau_j \in \mathcal{T}_{P_k}} \{B_{i,j}^{\text{G}}(\pi_{P_k}^{\text{spin}})\}$ calculated from (10) and $c = \max_{\substack{\forall j: \pi_j < \pi_i \\ \wedge \pi_j \leq \pi_{P_k}^{\text{G}} \wedge \tau_j \in \mathcal{T}_{P_k}}} \{B_{i,j}^{\text{L}}\}$. It can be observed under the condition of clause *(i)* that the set where $B_{i,j}^{\text{L}}$ is specified in a, i.e., $\forall j | \pi_j < \pi_i \wedge \pi_{P_k}^{\text{G}} < \pi_j$ is smaller than the set to specify $B_{i,j}^{\text{L}}$ in $\acute{a}$ where it is $\forall j | \pi_j < \pi_i$. This implies that $a \leq \acute{a}$. It also can be observed that $b = \acute{b}$ since the condition $\pi_i \leq \pi_{P_k}^{\text{spin}} = \pi_{P_k}^{\text{G}}$ is not valid in (10) under this clause. Moreover, it can be seen that the set to specify $B_{i,j}^{\text{L}}$ is smaller for c than $\acute{a}$, which leads to $c \leq \acute{a}$. As a result, $B_i^{\text{new}} \leq B_i^{\text{old}}$ under clause *(i)*.

It can be observed that under the condition of clause *(ii)* $a = 0$, $b = \acute{b}$ and $c = \acute{a}$, thus $B_i^{\text{new}} = \max(0 + \acute{b}, \acute{a})$ which means $B_i^{\text{new}} = B_i^{\text{old}}$ under clause *(ii)*. Furthermore, it can be observed that under the condition of clause *(iii)* $a = 0$, $b = \tilde{b}$ since the condition $\pi_i \leq \pi_{P_k}^{\text{spin}} = \pi_{P_k}^{\text{G}}$ is valid in (10) under this clause and $c = \acute{a}$, thus $B_i^{\text{new}} = \max(0 + \tilde{b}, \acute{a})$ which leads to $B_i^{\text{new}} = B_i^{\text{old}}$ under clause *(iii)*. This finishes the proof. ◀

6.4 Use of ILP

In this section we discuss the benefit of using optimization approaches such as mixed-Integer linear program (ILP) for bounding the maximum cumulative blocking imposed to tasks similar to [29] by Wieder and Brandenburg. Wieder and Brandenburg [29] showed that any blocking analysis that is based on inflation of the worst-case execution times of tasks with remote blocking can be pessimistic by a factor of $\Omega(\phi \cdot n)$ where $\phi \approx \lceil \frac{WR_i}{T_h} \rceil$ and τ_h is a higher priority task that spins and delays a lower priority task τ_i (Theorem 1 [29]).

Based on such a result, the tasks on a core with a priority within the range $(\pi_{P_k}^{\text{G}}, \pi_{P_k}^{\max}]$ do not suffer from such pessimism since, by definition, tasks in this range do not use any global resource (see Note 13). Therefore, using ILP could not tighten the blocking analysis for this range of tasks nor could benefit our comparative evaluation later in Section 8. Based on the results derived from Corollaries 49, 50 and 51 we show that it is enough to consider the tasks with a priority within

the above mentioned range when comparing the effectiveness of different spin-based protocols with a spin-lock priority from the same range.

However, ILP can tighten the blocking imposed to tasks on a core with a priority within the range $[1, \pi^{\mathrm{G}}_{P_k}]$, where 1 is the lowest possible priority level that a task can have on a core, since for this range of tasks the worst-case execution times are inflated with the remote blocking parameter in case those tasks use global resources. For tasks on a core with a priority within the range $[1, \pi^{\mathrm{G}}_{P_k}]$ selecting any spin-based protocol that uses a spin-lock priority within the range $[\pi^{\mathrm{G}}_{P_k}, \pi^{\max}_{P_k} - 1]$ will give the same blocking bound as when using *HP* (see Lemma 47 in Section 7.1). Therefore, for the tasks with a priority in the range $[1, \pi^{\mathrm{G}}_{P_k}]$ the same ILP constraints that has been presented for FIFO-ordered non-preemptive spin-based protocols [29] could be used to tighten the blocking bounds. Therefore, for the simplicity of the experiments we use the traditional analysis based on inflation of worst-case execution times of tasks when the schedulability of a core is checked for the set of tasks with a priority lower than this range.

Moreover, the holistic analysis of spin locks [9] encompasses pessimism due to inflating tasks' execution times [29] which has been overcome by the ILP-based analysis presented in [29]. However, since ILP cannot tighten the analysis for the set of tasks considered in our comparative evaluation, therefore, holistic analysis cannot as well. Thus, we do not consider this analysis approach here as well.

7 Properties of Spin-Based Protocols

In this section we specify the set of tasks on a processor for which the selection of any two spin-based protocols from the triple (*HP*, *CP*, $\widehat{CP}$), yield the same worst-case blocking bounds for a task τ_i. This facilitates the evaluation of the results, later, in Section 8. First, we present Lemmas 47 and 48 under which we show, respectively, that for selection of any two spin-lock priorities from the range $[\pi^{\mathrm{G}}_{P_k}, \pi^{\max}_{P_k}]$ where one is smaller than the other, for a task τ_i with a priority either *(a)* lower than the priority of both or *(b)* higher than the priority of both if τ_i does not use any local resource, using either spin-lock priorities will lead to the same blocking bounds for τ_i. This will help us to specify the set of tasks for which using either *CP* or *HP*, using either $\widehat{CP}$ or *HP* and using either either *CP* or $\widehat{CP}$ will lead to the same blocking bounds that we present by Corollaries 49, 50 and 51, respectively.

▶ **Lemma 47.** *Assume two different spin-based protocols σ_1 and σ_2 on P_k, with spin-lock priorities $\pi^{\mathrm{spin}_{\sigma_1}}_{P_k}$ and $\pi^{\mathrm{spin}_{\sigma_2}}_{P_k}$, respectively, where $\pi^{\mathrm{spin}_{\sigma_1}}_{P_k}, \pi^{\mathrm{spin}_{\sigma_2}}_{P_k} \in [\pi^{\mathrm{G}}_{P_k}, \pi^{\max}_{P_k}]$ and $\pi^{\mathrm{spin}_{\sigma_1}}_{P_k} \leq \pi^{\mathrm{spin}_{\sigma_2}}_{P_k}$. For any task $\tau_i \in \mathcal{T}_{P_k}$ where $\pi_i \leq \pi^{\mathrm{spin}_{\sigma_1}}_{P_k}$, using either σ_1 or σ_2 will yield the same value for the worst-case blocking B_i.*

Proof. We assume B_i which is calculated by (16) is equal to $max(A + B, C)$ where A, B and C are $A = \max_{\substack{\forall j:\pi_j<\pi_i \\ \wedge\pi^{\mathrm{spin}}_{P_k}<\pi_j\wedge\tau_j\in\mathcal{T}_{P_k}}} \{B^{\mathrm{L}}_{i,j}\}$, $B = B^{\mathrm{G}}_i(\pi^{\mathrm{spin}}_{P_k})$ and $C = \max_{\substack{\forall j:\pi_j<\pi_i \\ \wedge\pi_j\leq\pi^{\mathrm{spin}}_{P_k}\wedge\tau_j\in\mathcal{T}_{P_k}}} \{B^{\mathrm{L}}_{i,j}\}$. It is easy to see that $A = 0$ for any task τ_i that $\pi_i \leq \pi^{\mathrm{spin}_{\sigma_1}}_{P_k}$. Moreover, B is the same when using either σ_1 or σ_2 due to the fact that the condition $\pi_i \leq \pi^{\mathrm{spin}}_{P_k}$ in (10), which is the set from which B is derived, is satisfied using both σ_1 or σ_2. Further, C is also the same when using either σ_1 or σ_2 since the sets $\forall j : \pi_j \leq \pi^{\mathrm{spin}_{\sigma_1}}_{P_k}$ and $\forall j : \pi_j \leq \pi^{\mathrm{spin}_{\sigma_2}}_{P_k}$, which are the sets from which C is derived when σ_1 and σ_2 are used, respectively, are the same and leads to $\forall j : \pi_j < \pi_i$. ◀

▶ **Lemma 48.** *Assume two different spin-based protocols σ_1 and σ_2 on P_k, with spin priority levels $\pi^{\mathrm{spin}_{\sigma_1}}_{P_k}$ and $\pi^{\mathrm{spin}_{\sigma_2}}_{P_k}$ (remember Definition 6), respectively, where $\pi^{\mathrm{spin}_{\sigma_1}}_{P_k}, \pi^{\mathrm{spin}_{\sigma_2}}_{P_k} \in [\pi^{\mathrm{G}}_{P_k}, \pi^{\max}_{P_k}]$ and*

$\pi_{P_k}^{\text{spin}_{\sigma_1}} \leq \pi_{P_k}^{\text{spin}_{\sigma_2}}$, *and* $\pi_{P_k}^{\text{LG}} \leq \pi_{P_k}^{\text{spin}_{\sigma_2}}$. *For any task* $\tau_i \in \mathcal{T}_{P_k}$ *where* $\pi_{P_k}^{\text{spin}_{\sigma_2}} < \pi_i$, *then using either* σ_1 *or* σ_2 *will lead to the same worst-case blocking results.*

Proof. By assuming B_i calculated by (16) equal to $max(A+B, C)$ similar as in proof of Lemma 47, it is easy to see that $A = C = 0$ since by $\pi_{P_k}^{\text{LG}} < \pi_i$, by definition τ_i does not use any local resource which leads $B_{i,j}^{\text{L}} = 0$ in (16). Further, the condition of $\pi_i \leq \pi_{P_k}^{\text{spin}}$ in (10), which is the set from which B is derived, is not satisfied under both σ_1 and σ_2, thus B_i is the same under both σ_1 and σ_2 knowing that by definition, $\forall \tau_j | \pi_{P_k}^{\text{G}} < \pi_j < \pi_i \Rightarrow \mathcal{RS}_i^{\text{G}} = 0$. ◀

7.1 *CP* versus *HP*

In the following, we specify the set of tasks on a processor P_k that have the same blocking bounds for *CP* and *HP*.

From Lemma 47 the following corollary can be drawn.

▶ **Corollary 49.** *For any task* $\tau_i \in \mathcal{T}_{P_k}$ *where* $\pi_i \leq \pi_{P_k}^{\text{G}}$, *(i.e.,* τ_i*'s priority is in range C in Figure 3) then using either CP or HP will lead to the same blocking bounds.*

7.2 $\widehat{CP}$ versus *HP*

In the following, we specify the set of tasks on a processor P_k that have the same blocking bounds for *HP* and $\widehat{CP}$.

From Lemma 47, we draw the following conclusion.

▶ **Corollary 50.** *For any task* $\tau_i \in \mathcal{T}_{P_k}$ *where* $\pi_i \leq \pi_{P_k}^{\text{LG}}$, *(i.e.,* τ_i*'s priority is in ranges B or C in Figure 3) then using either* $\widehat{CP}$ *or HP will lead to the same blocking bounds.*

7.3 $\widehat{CP}$ versus *CP*

In the following, we specify the set of tasks on a processor P_k that have the same blocking bounds for *CP* and $\widehat{CP}$. This simplifies the comparison of the two protocols later in Section 8.

From Lemmas 47 and 48, we draw the following conclusion.

▶ **Corollary 51.** *If* $\pi_{P_k}^{\text{G}} \leq \pi_{P_k}^{\text{LG}}$, *for any task* $\tau_i \in \mathcal{T}_{P_k}$ *where* $\pi_i \leq \pi_{P_k}^{\text{G}}$, *or* $\pi_{P_k}^{\text{LG}} < \pi_i$ *(i.e.,* τ_i*'s priority is in range C or A in Figure 3) then using either CP or* $\widehat{CP}$ *will lead to the same blocking bounds.*

We already have shown in Section 5.2 by means of an illustrative example that *CP* and $\widehat{CP}$ are incomparable. The same result is also achievable based on the worst-case response time using (16) for calculating the blocking term.

▶ **Example 52.** The blocking term and worst-case response time of task τ_4 in Example35 for scenario *(1)* and *(2)* are denoted by $B_4^{sc_1-CP}$, $WR_4^{sc_1-CP}$ and by $B_4^{sc_2-CP}$, $WR_4^{sc_2-CP}$ for *CP* and by $B_4^{sc_1-\widehat{CP}}$, $WR_4^{sc_1-\widehat{CP}}$ and by $B_4^{sc_2-\widehat{CP}}$, $WR_4^{sc_2-\widehat{CP}}$ for $\widehat{CP}$, respectively. Their values for τ_4 under scenario *(1)* are as follows: $B_4^{sc_1-CP} = 4$, thus $WR_4^{sc_1-CP} = 9$ and $B_4^{sc_1-\widehat{CP}} = 8$, thus $WR_4^{sc_1-\widehat{CP}} = 13$, and under scenario *(2)* are as follows: $B_4^{sc_2-CP} = 7$, thus $WR_4^{sc_2-CP} = 12$ and $B_4^{sc_2-\widehat{CP}} = 4$, thus $WR_4^{sc_2-\widehat{CP}} = 9$. Therefore, τ_4 misses its deadline using $\widehat{CP}$ in scenario *(1)* and using *CP* in scenario *(2)*.

7.4 Key Trade-Off Factors

In this section, we elaborate the trade-off factors between CP and $\widehat{CP}$. According to Corollary 51 only if a task's priority level is within range B, then using $\widehat{CP}$ and CP may lead to different schedulability results. Note that, $\widehat{CP}$ exists only when $\pi^{\mathrm{G}}_{P_k} < \pi^{\mathrm{L}}_{P_k}$ (see Section 5). It can be observed that the maximum blocking imposed to a task τ_i with a priority within range B calculated by (16) results in term e in (12) when $\widehat{CP}$ is used and term d when CP is used. Thus, to compare the maximum blocking delay under CP and $\widehat{CP}$, we should compare (14) and (15). By looking at these two terms, it can be observed that using CP may cause an extra LBL term besides the LBG term compared to using $\widehat{CP}$. On the other hand, according to (41) the LBG term can be smaller under CP compared to $\widehat{CP}$ since $spin_{P_k,q}$ term is zero for a task τ_i in range B if CP is used. In other words, such a task has to wait for its lower priority task's spin-lock time under $\widehat{CP}$ but not under CP.

Followed by the discussion above, a task τ_i may experience one extra LBL if CP is used, whereas it may experience longer LBG if $\widehat{CP}$ is used since it has to wait for the spin-lock time of a lower priority task. These two parameters determine the trade-off factors of the two protocols. One conclusion from this discussion is that if the extra LBL is not imposed under CP, then CP outperforms $\widehat{CP}$ since under $\widehat{CP}$ a task may be delayed longer due to the spinning of lower priority tasks.

7.5 Intermediate Spin-Based Protocol

In Sections 5.2 and 7.3, we showed that CP and $\widehat{CP}$ are incomparable by means of both a trace example and analysis in Examples 35 and 52, respectively. In this section, we show for the same example using a third scenario (Scenario 3) through the analysis results that if CP and $\widehat{CP}$ cannot make a task set schedulable on a core, there may exist an intermediate spin-lock priority within the range $(CP, \widehat{CP})$ that can make the task set schedulable. Under a third scenario *(3)* we denote the blocking term and worst-case response time of a task τ_i under CP by $B_i^{sc_3-CP}$ and $WR_i^{sc_3-CP}$, under $\widehat{CP}$ by $B_i^{sc_3-\widehat{CP}}$ and $WR_i^{sc_3-\widehat{CP}}$ and under a third protocol which we call $\widetilde{CP}$ by $B_i^{sc_3-\widetilde{CP}}$ and $WR_i^{sc_3-\widetilde{CP}}$, respectively. Under scenario *(3)* we again use dedicated task specifications $(C_i, Cs_{i,l}, Cs_{i,g})$ for task τ_3 and τ_7, i.e., τ_3: (2, 2, 0) and τ_7: (7, 0, 5). $\pi_{P_k}^{\mathrm{spin}_{CP}} = 2$, $\pi_{P_k}^{\mathrm{spin}_{\widehat{CP}}} = 5$ and $\pi_{P_k}^{\mathrm{spin}_{\widetilde{CP}}} = 3$. The blocking terms and respective worst-case response times of τ_4 under this scenario are as follows: $B_4^{sc_3-CP} = 5$, thus $WR_4^{sc_1-CP} = 10$, $B_4^{sc_3-\widehat{CP}} = 8$, thus $WR_4^{sc_1-\widehat{CP}} = 13$, and $B_4^{sc_3-\widetilde{CP}} = 3$, thus $WR_4^{sc_1-\widetilde{CP}} = 9$. Therefore, since τ_4 misses its deadline under both CP and $\widehat{CP}$ but not under $\widetilde{CP}$, thus the task set is schedulable under $\widetilde{CP}$ only. To determine such spin-lock priority, if any, the priority levels between CP and $\widehat{CP}$ need to be explored linearly, applying worst-case response time calculations using (16). Please note that finding $\widehat{CP}$ limits the search. In general, spin-lock priorities in the range $(CP, \widehat{CP})$ do not necessarily dominate $HP.$, whereas $\widehat{CP}$ does. Moreover, whenever a task set is schedulable by CP and $\widehat{CP}$, this does not imply that the set is also schedulable by all, some or even any spin-lock priority in the range $(CP, \widehat{CP})$. We have performed an experiment which shows that out of 8000 task sets $\widetilde{CP}$ could only schedule 2 more task sets compared to both CP and $\widehat{CP}$.

8 Evaluation

In this section we present the experimental results of comparing HP, CP and $\widehat{CP}$. According to Corollaries 49, 50 and 51, it is enough to compare the worst-case response time of tasks of range B in Figure 3 when comparing CP and $\widehat{CP}$, range A when comparing $\widehat{CP}$ and HP and range

$A \cup B$ when comparing *CP* and *HP*. In our experiments, we therefore only consider tasks in the related range, effectively ignoring tasks that have identical results under the compared protocols. As we discussed in Section 6.4, using ILP cannot tighten the blocking bounds for this range of tasks. Therefore, for simplicity, we use the traditional worst-case response time analysis [4] for all tasks on a core. We run two types of experiments. In our first type of experiments we calculate the improvement in worst-case response time of tasks of one protocol compared to the other. We use $RTI(a, b)$ to denote the *response time improvement* under protocol a compared to protocol b. Since the response jitter of a task is bounded by the difference of the worst-case and best-case response time of that task [25, 11] and the best-case response time being independent of a global resource sharing protocol, the bound on the response jitter decreases when the worst-case response time decreases. Therefore, response time improvement of tasks are directly correlated with response jitters. We denote $RTI_i(a, b)$ for a task τ_i as $\frac{(WR_i^b - WR_i^a)}{max(WR_i^a, WR_i^b)} \times 100$, where WR_i^a and WR_i^b denote the worst-case response time of task τ_i under protocols a and b, respectively. For a randomly generated task set, we show the percentage of tasks as a function of $RTI(a, b)$. We have performed the experiments for how different system parameters as the number of processors, task set utilization, number of tasks per core and local and global critical section lengths can affect the $RTI(a, b)$. Due to space constraints, we only illustrate *RTI* for changing the numbers of cores as well as local critical section lengths here. Further results can be found in [2]. In the second type of experiments, we compare the system schedulability under *HP*, *CP* and $\widehat{CP}$. We perform the response-time analysis [4] to check the schedulability of task sets according to [18]. For this experiment we have generated 200,000 task sets for 4-processors. We denote $PS_{(C)}$ as the percentage of the schedulable systems under condition C, e.g., $PS_{(HP \wedge CP \wedge \neg \widehat{CP})}$ denotes the percentage of systems that are schedulable under both *HP* and *CP* but not under $\widehat{CP}$. This percentage is calculated based on the number of systems that are schedulable under any of the three aforementioned spin-based protocols.

8.1 Experimental Setup

In each experiment we randomly generate task sets for each processor. The number m of processors is selected from the set {4, 8, 12, 16}. For each experiment 1000 schedulable task sets under the considered protocols are generated. The task set size is the same for each processor and is selected from the set {20, 40, 60}. Tasks are randomly selected to be dedicated to ranges A, B and C in Figure 3 such that at least one task is dedicated to priority ranges A and B each in order to implement $\widehat{CP}$ and *CP* protocols for the sake of comparison. The task set utilization is also the same for each processor and is selected from the set {0.4, 0.6, 0.8}. The UUnifast algorithm [7] is used to generate the utilization of each task. The period of each task is randomly generated from the range [10, 150] ms with a granularity of 10 ms. The worst-case execution time of a task is calculated by $C_i = U_i \times T_i$. Deadlines of tasks are selected randomly according to a uniform distribution in the range $[C_i + \alpha \times (T_i - C_i), T_i]$ with $\alpha = 0.5$ as the default [14]. The maximum number of accesses to local and global resources for each task is 4. The local and global critical section lengths (lcs and gcs) are generated according to $Cs_q = \beta \times C_i$, where β is selected from the set $\{0.1, 0.2, 0.3\}$. The number of local resources per processor as well as number of global resources per task set is set to 3.

In our basic system configuration which is used for both types of experiments in Sections 8.2 and 8.3, the number of processors is set to $m = 4$, the task set utilization per core is 0.6, the number of tasks on each processor is 20, and $\beta = 0.2$.

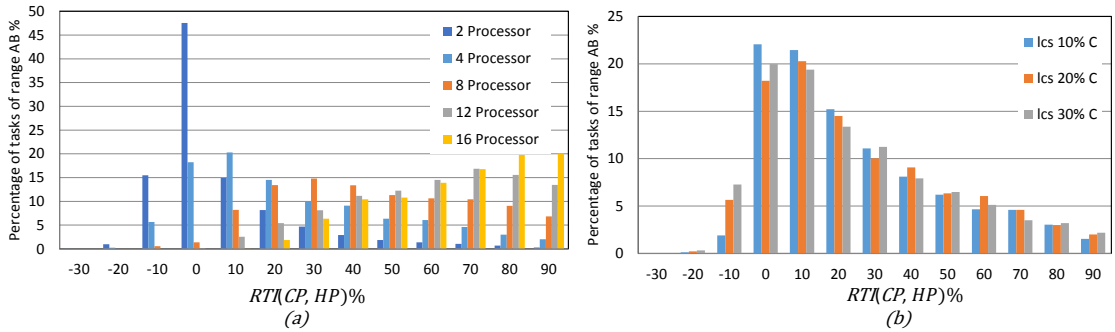

Figure 5 RTI for CP versus HP for (a) different number of processors m, and (b) different values of lcs.

8.2 Results for Response Time Improvements

For these experiments bar charts will be used to visualize the results, and the presented graphs show the distribution of the tasks for the calculated RTI. The X-axis in the graphs represents $RTI(a, b)$ and the Y-axis shows the percentage of the examined tasks that have that improvement. Note that, values in the X-axis present a non-continuous range. A bar in a graph that presents $RTI(a, b)$ with x_i as X value and y_i as Y value shows that $y_i\%$ of tasks have an improvement in the range $(x_{i-1}\%, x_i\%]$ in their response times under protocol b compared to protocol a. Note that a positive RTI value for a graph representing $RTI(a, b)$, shows that response times under protocol b are larger compared to protocol a. Similarly a negative RTI value shows that response times are smaller under protocol b compared to protocol a. The results in Figures 6, 5 and 7 show the variation in distribution of tasks for the calculated RTI values for different numbers of processors. More experimental results are available in [2] from which similar conclusions are derived as from the graphs presented here.

8.2.1 Evaluation Results of CP versus HP

Figure 5 shows that CP improves response time of tasks up to 90% compared to HP. In more detail, it can be observed from Figure 5.(a) that increasing the number of cores leads to more tasks having larger response time improvement under CP compared to HP. For $m = 2$ around 1% of tasks have up to 20% improvement. The same trend was also obtained by increasing the global critical section lengths for which the results can be found in [2]. This results are confirmed by revisiting (16) where increasing the number of cores and global critical sections is positively correlated with $spin_{P_k,q}$ included in $B_i^{\mathrm{G}}(\pi_{P_k}^{\mathrm{spin}})$ which is zero for the compared tasks under CP and not under HP. Moreover, Figure 6.(b) shows that by increasing the local critical section lengths, CP's performance decreases compared to HP with regard to response time improvement. This is due to the fact that by increasing the local critical sections $B_{i,j}^{\mathrm{L}}$ increases. The same trend was also obtained by increasing the number of tasks per core [2]. The reason is that by increasing the number of tasks on a core, the number of tasks in range B may increase as well, which in the worst-case leads to in an increase in the first $B_{i,j}^{\mathrm{L}}$ term in RHS of (16). This term is zero under HP. Increasing the task set utilization did not show a significant improvement. The reason is that by increasing the task set utilization the execution time of tasks are increased leading in an increase in both local and global critical section lengths which seems to nullify the effect of each other. The interesting observation is to have both positive and negative RTI values in the graphs which shows response time improvement under both CP and HP. This confirms the incomparability

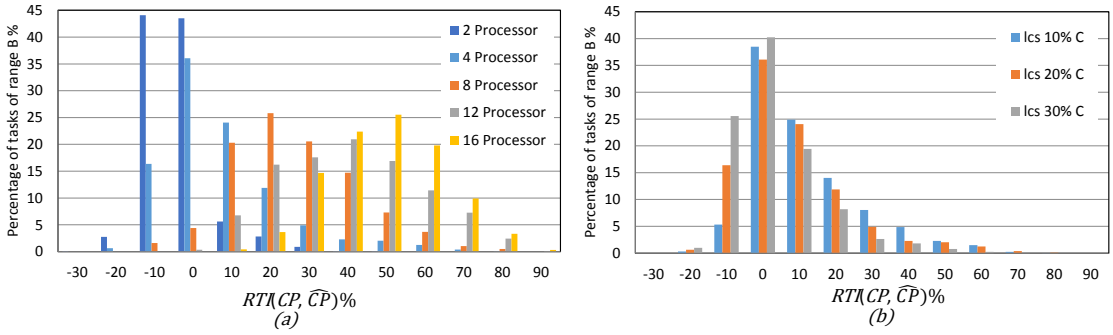

Figure 6 *RTI* for *CP* versus $\widehat{CP}$ for (*a*) different number of processors m, and (*b*) different values of *lcs*.

claim of *CP* and *HP* which we previously have shown by examples [1]. In conclusion, the sum of the percentages for the positive values is larger than the sum of the percentages for the negative values from graphs. Hence, overall with the given system configuration in these experiments, *CP* introduces less delays to tasks compared to *HP*.

8.2.2 Evaluation Results of *CP* versus $\widehat{CP}$

Figure 6 shows that, in general, *CP* improves response time of tasks compared to $\widehat{CP}$ which can reach up to 80%. However, it can be observed that when the number of cores are small $\widehat{CP}$ could improve response time of tasks up to 20%. In more detail, it can be observed in Figure 6.(a) that for $m = 2$, roughly 45% of tasks have up to 10% response time improvement under $\widehat{CP}$ compared to *CP* and around 3% have up to 20% improvement. Similar trend is achieved here as well by increasing the number of cores, global critical and local critical sections, task set size and utilization similar to when comparing *CP* and *HP*. The reason is that for all tasks of range *B* the spin-lock priority under $\widehat{CP}$ is higher than their priority similar to spin-lock priority under *HP*, when comparing *CP* versus $\widehat{CP}$ compared to when comparing *CP* and *HP*. The interesting observation here is that both positive and negative *RTI* values exist which confirms the incomparability of *CP* and $\widehat{CP}$ previously shown by the example in Section 5.2. In conclusion, overall with the given system configuration in these experiments as well, *CP* introduces less delays to tasks compared to $\widehat{CP}$.

8.2.3 Evaluation Results of $\widehat{CP}$ versus *HP*

Figure 7 shows that $\widehat{CP}$ improves response time of tasks up to 90% compared to *HP*. Figure 7.(a) illustrates that by increasing the number of cores $\widehat{CP}$ outperforms *HP* more, in terms of response time improvement. The reason is that for tasks of range *A*, $spin_{P_k,q} = 0$ under $\widehat{CP}$ and not under *HP*. The interesting observation here is that there are no negative *RTI* values in any of the related graphs meaning that response times cannot be improved under the *HP* compared to $\widehat{CP}$ which confirms dominance of $\widehat{CP}$ compared to *HP* proven by Lemma 33.

8.3 Schedulability Results

In the second type of experiments, the schedulability under *HP*, *CP* and $\widehat{CP}$ is investigated. The results show that in general from the schedulable systems, a higher percentage are schedulable under *CP* compared to the other two protocols, i.e., $PS_{(CP)} = 99.6\%$, $PS_{(\widehat{CP})} = 76.2\%$ and

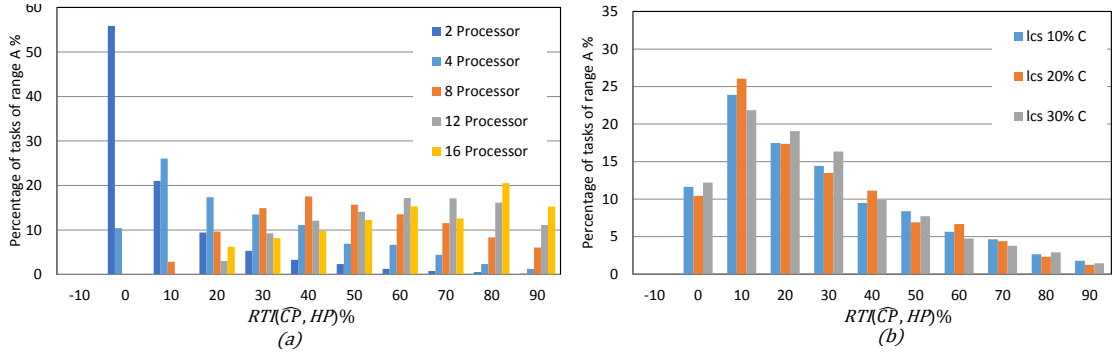

Figure 7 RTI for $\widehat{CP}$ versus HP for (a) different number of processors m, and (b) different values of lcs.

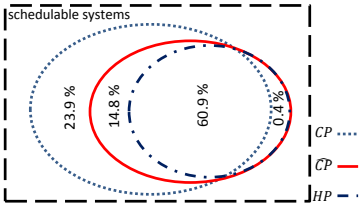

Figure 8 Schedulability percentage under HP, CP and $\widehat{CP}$.

$PS_{(HP)} = 61.4\%$. The results show that most of the schedulable systems were schedulable under all three protocols, i.e., $PS_{(CP \wedge \widehat{CP} \wedge HP)} = 60.9\%$. Moreover, this results also confirms that $\widehat{CP}$ dominates HP, i.e., all systems that were schedulable under HP were also schedulable under $\widehat{CP}$, however a percentage of tasks were only schedulable under $\widehat{CP}$ and not under HP which is presented by $PS_{(\widehat{CP}) \wedge \neg HP} = 14.8\%$. These schedulability results have also been illustrated in Figure 8. Note that the values in the graph, illustrate the schedulability of area in which the value is located.

9 Conclusion and Future Work

In this paper, we investigated spin-based protocols for resource and jitter constrained embedded multi-core platforms with the aim to improve the cost-efficiency and quality of control as well as schedulability performance. We have focused on fixed-priority partitioned scheduling which is the industry's preferred scheduling approach. For such systems, non-preemptive spin-based protocols have shown a good performance in improving the systems costs by offering use of one shared stack for running all tasks residing on a core. However, they have shown a poor efficiency in preserving the control and schedulability quality of those systems. To address these aspects, we have investigated preemptive spin-based protocols which give a better promise to pertain all these factors. Further, we showed that the selection of priority upon which a task spins is significantly important since it affects the blocking duration and hence the response time and response jitter of tasks. We have presented spin-lock priorities for preemptive-spin-based protocols for which the response time of tasks and hence the corresponding response jitters are decreased compared to when using the non-preemptive spin-based protocol.

We focused on spin-based protocols where a fixed spin-lock priority is used for spinning of any task on the core in combination with FIFO-ordering policy where under a classical technique tasks are kept in the queue upon preemption. From this type we focused on a special range that

offer attractive properties where spinning occurs at a priority at least the highest ceiling of any global resource which is the spin-lock priority of an existing spin-based protocol *CP*. Selecting from this range keeps the remote blocking bound as well as the resource queue size confined to a factor that is the number of cores in the system, similar to non-preemptive spin-based protocols. It also guarantees that the number of some resources such as stack used per core is not increased considerably compared to the non-preemptive spin-based protocols. In this paper we introduced a special spin-based protocol from this range, called $\widehat{CP}$ where we showed that it dominates the traditional non-preemptive spin-based protocol *HP*, and all spin-based protocols that use a spin-lock priority between those used by these two. Further, we showed that $\widehat{CP}$ and *CP* are incomparable, thus we have provided the blocking analysis for the considered range of spin-locks in order to enable the comparative evaluation of these incomparable protocols. The new analysis turns out to give tighter blocking bounds than those previously presented for the *CP* protocol.

Finally, we showed that if a task set is unschedulable under both *CP* and $\widehat{CP}$ on a processor, there may exist a spin-based protocol that uses a spin-lock priority in between of those used by *CP* and $\widehat{CP}$ which can make the task set schedulable. The complexity of finding such spin-based protocol, if any, is linear and can be a small value. The experimental results showed that, in general, *CP* can provide better schedulability results compared to *HP* and $\widehat{CP}$. Moreover, the results showed that although $\widehat{CP}$ and *CP* are incomparable, under specific system configurations tasks can obtain up to 70% improvement in their response times under *CP* compared to $\widehat{CP}$. Similarly, tasks can gain up to 90% improvements in their response times under *CP* and $\widehat{CP}$ compared to *HP*. It can be viewed from the evaluation results that in general, more tasks can have shorter response times under *CP* than under *HP* and $\widehat{CP}$.

Towards optimizing the spin-based protocols for tasks, we would like to look at the following steps: optimizing the spin-lock priority *(i)* per processor, *(ii)* per task, *(iii)* per resource and (iv) per resource access. In this paper we have focused on step *(i)* for a specific range of spin-based protocols. We leave the later steps as future work.

References

1 Sara Afshar, Moris Behnam, Reinder J. Bril, and Thomas Nolte. Flexible spin-lock model for resource sharing in multiprocessor real-time systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014, Pisa, Italy, June 18-20, 2014*, pages 41–51. IEEE, 2014. doi:10.1109/SIES.2014.6871185.

2 Sara Afshar, Moris Behnam, Reinder J. Bril, and Thomas Nolte. On per processor spin-lock priority for partitioned multiprocessor real-time systems. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, 2014. URL: http://www.es.mdh.se/publications/3766-.

3 James H. Anderson, Rohit Jain, and Kevin Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 346–355. IEEE Computer Society, 1998. doi:10.1109/REAL.1998.739768.

4 Neil C. Audsley, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. doi:10.1049/sej.1993.0034.

5 AUTOSAR release 4.1, specification of operating system, 2013. URL: http://www.autosar.org.

6 Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991. doi:10.1007/BF00365393.

7 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.

8 Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), 21-24 August 2007, Daegu, Korea*, pages 47–56. IEEE Computer Society, 2007. doi:10.1109/RTCSA.2007.8.

9 Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2011. AAI3502550.

10 Björn B. Brandenburg and James H. Anderson. An implementation of the pcp, srp, d-pcp, m-pcp, and FMLP real-time synchronization protocols in litmus$^{\mathrm{rt}}$. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*,

pages 185–194. IEEE Computer Society, 2008. doi:10.1109/RTCSA.2008.13.

11 Reinder J. Bril, Elisabeth F. M. Steffens, and Wim F. J. Verhaegh. Best-case response times and jitter analysis of real-time tasks. *J. Scheduling*, 7(2):133–147, 2004. doi:10.1023/B:JOSH.0000014069.63823.e7.

12 Alan Burns and Andy J. Wellings. A schedulability compatible multiprocessor resource sharing protocol - mrsp. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 282–291. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.37.

13 Travis S. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the Real-Time Systems Symposium. Raleigh-Durham, NC, December 1993*, pages 148–157. IEEE Computer Society, 1993. doi:10.1109/REAL.1993.393505.

14 Robert I. Davis and Marko Bertogna. Optimal fixed priority scheduling with deferred pre-emption. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS 2012, San Juan, PR, USA, December 4-7, 2012*, pages 39–50. IEEE Computer Society, 2012. doi:10.1109/RTSS.2012.57.

15 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, 2011. doi:10.1145/1978802.1978814.

16 UmaMaheswari C. Devi, Hennadiy Leontyev, and James H. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *18th Euromicro Conference on Real-Time Systems, ECRTS'06, 5-7 July 2006, Dresden, Germany, Proceedings*, pages 75–84. IEEE Computer Society, 2006. doi:10.1109/ECRTS.2006.10.

17 Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. The multiprocessor bandwidth inheritance protocol. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*, pages 90–99. IEEE Computer Society, 2010. doi:10.1109/ECRTS.2010.19.

18 Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, 2-6 December 2001*, pages 73–83. IEEE Computer Society, 2001. doi:10.1109/REAL.2001.990598.

19 Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Stack size minimization for embedded real-time systems-on-a-chip. *Design Autom. for Emb. Sys.*, 7(1-2):53–87, 2002. doi:10.1023/A:1019795414875.

20 Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), May 27-30, 2003, Toronto, Canada*, page 189. IEEE Computer Society, 2003. doi:10.1109/RTTAS.2003.1203051.

21 Theodore Johnson and Krishna Harathi. A prioritized multiprocessor spin lock. *IEEE Trans. Parallel Distrib. Syst.*, 8(9):926–933, 1997. doi:10.1109/71.615438.

22 Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Trans. Comput. Syst.*, 15(1):3–40, 1997. doi:10.1145/244764.244765.

23 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991. doi:10.1145/103727.103729.

24 Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

25 Ola Redell and Martin Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *14th Euromicro Conference on Real-Time Systems (ECRTS 2002), 19-21 June 2002, Vienna, Austria, Proceedings*, pages 165–172. IEEE Computer Society, 2002. doi:10.1109/EMRTS.2002.1019196.

26 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

27 H. Takada and K. Sakamura. Predictable spin lock algorithms with preemption. In *11th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS'94)*, pages 2–6, 1994. doi:10.1109/RTOSS.1994.292571.

28 Hideyuki Takada and Ken Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernel. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*, pages 134–143. IEEE Computer Society, 1997. doi:10.1109/REAL.1997.641276.

29 Alexander Wieder and Björn B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 45–56. IEEE Computer Society, 2013. doi:10.1109/RTSS.2013.13.

A Table of Notations

Table 1 Table of notations

Notations	Description
P_k	processor k
τ_i	task i
C_i	worst-case execution time of τ_i
$\acute{C}_i$	inflated execution time of τ_i
T_i	minimum inter-arrival time of τ_i
D_i	relative deadline of τ_i
π_i	priority of τ_i
$\mathcal{T}_{P_k}$	set of tasks allocated to processor P_k
R_q	resource q
$\mathcal{R}^{\mathrm{L}}_{P_k}$	set of local resources accessed by tasks on P_k
$\mathcal{R}^{\mathrm{G}}_{P_k}$	set of global resources accessed by tasks on P_k
$\mathcal{RS}^{\mathrm{L}}_i$	set of local resources accessed by jobs of τ_i
$\mathcal{RS}^{\mathrm{G}}_i$	set of global resources accessed by jobs of τ_i
$Cs_{i,q}$	worst-case execution time in all τ_i's requests on R_q
$n^{\mathrm{G}}_{i,q}$	maximum number of requests of a job of τ_i for a global resource R_q
$\mathcal{T}_{P_k,q}$	set of tasks on P_k that request R_q
WR_i	worst-case response time of τ_i
BR_i	best-case response time of τ_i
RJ_i	response jitter of τ_i
$\pi^{\max}_{P_k}$	highest priority level on P_k
$ceil_{P_k}(R_l)$	ceiling of R_l on P_k
$\pi^{\mathrm{L}}_{P_k}$	highest local ceiling of any local resource on P_k
π^{G}_{Pk}	highest local ceiling of any global resource on P_k
$\pi^{\mathrm{LG}}_{P_k}$	highest local ceiling of any (local or global) resource on P_k
$\pi^{\mathrm{spin}}_{P_k}$	an arbitrary fixed spin-lock priority for any task on P_k
$\pi^{\mathrm{spin}_\sigma}_{P_k}$	spin-lock priority of spin-based protocol σ of P_k
LBL	local blocking due to local resources
LBG	local blocking due to global resources
$spin_{P_k,q}$	maximum remote blocking (i.e. spin-lock time) for any task on P_k to acquire R_q
$spin_i$	maximum total remote blocking (i.e. spin-lock time) for τ_i to acquire all its resources
$R^{\mathrm{spin}}_{P_k}$	the "virtual" local spin resource on P_k
B^{L}_i	LBL imposed to a task τ_i under an arbitrary spin-based protocol
B^{G}_i	LBG imposed to a task τ_i under an arbitrary spin-based protocol
B_i	total pi-blocking imposed to a task τ_i under an arbitrary spin-based protocol
$B_i(\pi^{\mathrm{spin}}_{P_k})$	total blocking to a task $\tau_i \in \mathcal{T}_{P_k}$ under a spin-based protocol with spin-lock priority $\pi^{\mathrm{spin}}_{P_k}$