Project title:	High-Performance Real-Time Architectures for Low-Power Embedded Systems
Acronym:	HERCULES
Project ID:	688860
Call identifier:	H2020 – ICT 04-2015 – Customized and low power computing
Project coordinator:	Prof. Marko Bertogna, University of Modena and Reggio Emilia

D5.3: Integrated Schedulability Analysis

Document title:	Integrated Schedulability Analysis
Version:	1.3
Deliverable No.:	D5.3
Lead task beneficiary:	UNIMORE
Partners involved:	UNIMORE, CTU, ETHZ
Author:	M. Solieri, T. Kloda, M. Bertogna, M. Sojka, M. Baryshnikov
Status:	Final
Date:	2018-12-30
Nature:	Report
Dissemination level:	PU – Public

Purpose & Scope

The purpose of this deliverable is to provide a description and analysis of the software techniques that allows predictable and efficient application hosting and scheduling on both CPU and GPU.

Revision History

Version	Date	Author/Reviewer	Comments
0.1	2018-05-30	CTU	Initial revision
0.2	2018-06-27	UNIMORE	Drafted sub-section on PREM implementation
1.0	2018-06-30	UNIMORE	Drafted sub-section on PREM schedulability
1.1	2018-12-10	UNIMORE	Added sub-section on schedulability of PREM and GPREM
1.2	2018-12-29	CTU	Integrated reference to FPGA-based PREM
1.3	2018-12-30	UNIMORE	Final revision

HERCULES project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement: 688860.

This document reflects only the author's view and the EU Commission is not responsible for any use that may be made of the information it contains.

Contents

Acronyms 1

1 Introduction 2

2 Online PREM Co-Scheduling 3
- 2.1 Sotfware-Based Scheduling with Jailhouse Hypervisor . . . 3
  - 2.1.1 PREM model and MEmory under Mutual EXclusion (memex) . . . 3
  - 2.1.2 Interface and Integration . . . 3
  - 2.1.3 Limited Unpredictability . . . 4
  - 2.1.4 Accelerator . . . 4
  - 2.1.5 Memex Spinlock Implementation . . . 5
  - 2.1.6 MemGuard Implementation . . . 8
- 2.2 Memguard Schedulability analysis . . . 10
  - 2.2.1 Model . . . 10
  - 2.2.2 Related Work . . . 11
  - 2.2.3 Schedulability Analysis Selection for Hercules . . . 11
  - 2.2.4 Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems 11
- 2.3 Integrated PREM Schedulability analysis . . . 16
  - 2.3.1 Task Model . . . 17
  - 2.3.2 Platform Model . . . 17
  - 2.3.3 Related Work about Memory-Computation Tasks . . . 18
  - 2.3.4 Schedulability Analysis for PREM with priority-queue spinlock and memguarding . . . 19

3 FPGA-based approach to PREM scheduling 21

4 Off-line scheduling and analysis of PREM execution 22
- 4.1 Algorithm overview . . . 22
- 4.2 Off-line scheduling evaluation . . . 25
  - 4.2.1 Kernels . . . 25
  - 4.2.2 Use-case scenarios . . . 25
  - 4.2.3 Experimental results . . . 28

5 Conclusion 32

Acronyms

ADAS	Advanced Driver-Assistance Systems	25
API	Application Programming Interface	3
AXI	Advanced eXtensible Interface	21
CCR	Compute-to-Communication Ratio	25
COTS	Commercial Of The Shelf	25
CPU	Central Processing Unit	4
CTI	Cross-Trigger Interface	21
CTM	Cross-Trigger Matrix	21
DAG	Directed Acyclic Graph	27
DRAM	Dynamic Random Access Memory	3
ETM	Embedded Trace Macrocell	21
FFT	Fast Fourier Transform	25
FTM	Fabric Trace Macrocell	21
GEMM	General Matrix Multiplication	25
GPU	Graphics Processing Unit	4
iFFT	Inverse Fast Fourier Transform	27
ILP	Integer Linear Programming	22
OS	Operating System	3
PCI	Peripheral Component Interconnect	5
PL	Programmable Logic	21
PREM	Predictable Execution Model	3
PS	Processing System	21
RTOS	Real-Time Operating System	3
TX2	NVIDIA Tegra X2	4
WCET	Worst-Case Execution Time	25

1 Introduction

The predictable task execution model (PREM) [**PREMpaper**] and the MemGuard technique [Yun+13] are two ways to enhance predictability of task execution times and reduce the amount pessimism in worst-case execution times estimation that often impedes the integration of non-trivial systems. The former is stronger, but it imposes strict requirements on the memory accesses and the code structure, relying on non-trivial platform features like cache partitioning. The latter is weaker, but it has a wider applicability. This document presents the implementation and analysis of the schedulability of the combined approaches.

This deliverable is an ideal continuation of [**D5.2**; D5.1], which presented software techniques for real-time scheduling and schedulability on, respectively, system host, i.e. cores of the central processing unit, and accelerators, i.e. graphics processing unit.

2 Online PREM Co-Scheduling

2.1 Sotfware-Based Scheduling with Jailhouse Hypervisor

In this section, we describe the design and implementation of software mechanisms that realize Predictable Execution Model (PREM) co-scheduling between hypervisor guests.

We start by recalling the key concepts of PREM introducing the goal (Section 2.1.1) and the software architecture (Section 2.1.2) of the Jailhouse extension authored by UNIMORE and CTU. We then present a relaxed model with limited unpredictability (Section 2.1.3), and discuss how to support the presence of computing accelerators on board. Finally, we illustrate and motivate the internal details of the implementation (Section 2.1.5).

A detailed evaluation of the provided solution is provided in [**D1.4**].

2.1.1 PREM model and MEmory under Mutual EXclusion (memex)

The PREM approach imposes tasks to ideally alternate between two kinds of operating modes named *intervals*:

Computation Execution performs no access to central DRAM, hence relying only on caches that are dedicated to the core.

Memory Cache writeback to Dynamic Random Access Memory (DRAM) is operated, followed by a cache prefetch for the next computation phase.

The key point is that memory phases of different tasks running on a multi-core system are required to never overlap, eliminating any competition for accessing DRAM.

Since PREM models the memory as a system resource that cannot be accessed simultaneously by different cores, it appears natural to implement an online co-scheduling mechanism by means of a *memex*, i.e. a memory mutex or lock. The transition of a task from a computation interval to a memory one thus requires the task to ask, and obtain, the lock on the memex. Conversely, the transition from memory to computation interval unlocks the memex for other tasks.

2.1.2 Interface and Integration

Hypervisor Support The memex implementation has been integrated in the Jailhouse hypervisor, by an extension provided and documented in [D4.5]. Externally, i.e. at the level of Application Programming Interface (API), memex lock and unlock operations are exposed in a dedicated Jailhouse hypercall that is made available to any guest, i.e., the Linux root-cell or any other Operating System (OS) or bare-metal application hosted as an inmate.

Runtime Integration This hypervisor-level API is also lifted to the application level, to allow compatibility with the PREM-ising compiler optimisation [**ETH PREM compiler**], whose interface definition called `libpremnotify` is defined in [D3.5] (where also a reference implementation for the Linux OS is provided). Whilst the hypervisor version of `libpremnotify` suffices for a bare metal application or lightweight Real-Time Operating Systems (RTOSs), more complex OSs typically restrict hypercall invocation from a privileged kernel space. To demonstrate integration with a general-purpose OS, a Linux kernel module has been implemented to proxy Jailhouse hypercalls through the IOCTL interface.

Predictable Caches Recall that for PREM to be effective, a form of cache partitioning is required. Otherwise, predictability of computation phases would be jeopardised by contention on the shared L2 cache. For this reason, the Jailhouse PREM support is integrated with another modification providing cache colouring — the HERCULES extension of Jailhouse [D4.5] includes this coloured lock-down mechanism, whose dedicated evaluation is provided in [D5.1]. Again in [D5.1] another PREM-enabler technique is presented and evaluated — it is called preventive invalidation and allows the prefetch operations performed during memory phases to be executed deterministically in spite of the pseudo-random replacement policy, which would instead expose to self-eviction of useful data.

2.1.3 Limited Unpredictability

We now present two additional techniques widening the applicability of PREM.

Compatible Intervals Predictable intervals are executed in a non preemptive manner, and no system call can be invoked during them. For the sake of flexibility, the PREM definition is therefore relaxed, by allowing a third kind of interval without the strict dichotomy between memory and computation. A *compatible* interval thus enables execution of foreign calls, such as system-calls, and any other situation where memory activity cannot be reordered and grouped, or when the obtained prefetch or write-back sequences have insufficient length (i.e., causing the PREM approach benefits to be frustrated by its own synchronization latency). This compromise is made possible by MemGuard [**MemGuard**], which limits the memory bandwidth attainable by a core in compatibility mode.

Timeout A PREM system should be protected by misbehaving tasks or OSs, because a complete deadlock can be caused by simply obtaining and not releasing the memex. A *watchdog* controller is thus introduced, to force task/OS interruption and compulsory unlock, after a given amount of time. In this case, the guilty task/OS activity is demoted from memory interval to the compatible one, hence throttling its memory usage. Technically, the hypervisor is equipped with a private timer that is set at the beginning of every memory interval. The watchdog, exposed as the interrupt service routine linked to such timer, forcibly unlocks the memex for the inmate and changes its current PREM interval.

In conclusion: MemGuard gives an upper bound to the amount of DRAM interference possibly measured by a CPU core; the PREM watchdog detects excessively long memory phases exposing system cores to starvation, and activates MemGuard limitation.

2.1.4 Accelerator

In order for a PREM implementation to be implemented completely, each memory client, and in particular each DMA subsystem, needs to adhere to the model. For the sake of simplicity, we focused on the most problematic kind of devices — external computing accelerators with a dynamic programming model, i.e. Graphics Processing Unit (GPU). Our reference architecture NVIDIA Tegra X2 (TX2) features a GP10B GPU.

Beside the many challenges, we observe two features allowing instead a key design choice. Firstly, the GPU operation entirely depends on the controlling host that submit jobs, i.e. the Central Processing Unit (CPU) core complex. Secondly, the Jailhouse design follows a static resource partitioning approach, accelerators are thus not shared amongst several guests. It looks therefore quite natural and elegant to assume that the responsibility of memex management for any given accelerator is encapsulated in the unique controlling host.

In practice, this means that the GPU-controlling host and its GPU are externally perceived as a whole by other hypervisors guests, whilst internally the Linux module is the only entity in charge of performing the Jailhouse

Listing 1: Memex structure

```
1 typedef struct {
2         u16 owner;
3         u16 queue;
4 } memexlock_t __attribute__((aligned(4)));.
```

hypercalls to ask and release the memex. This means that, once the memex lock is granted on the GPU-equipped inmate, parallel CPU and GPU programs can mutually exchange the lock without necessarily involving the hypervisor.

2.1.5 Memex Spinlock Implementation

Being a featherweight hypervisor, Jailhouse does not include (at the moment of writing) any feature supporting concurrency or synchronisation between guests — the only inter-guests services available merely implement communication, e.g. via shared memory and Peripheral Component Interconnect (PCI) emulation. The memex lock implementation thus required a custom design.

Power vs Performances The key trait of our solution regards the synchronisation primitive used to implement the memex request operation. A passive kind of wait employing interrupts tends on the one hand to slightly reduce power consumption, because a waiting core can be implicitly put in low-power mode by WFI or WFE instructions. On the other hand, passive waiting considerably increase the response latency, because the basic mechanism it employs is an inter-processor interrupt to signal lock release, and this is an order of magnitude slower than the exclusive load and store instructions on a the shared L2 cache, which are the basic building block of active waiting. Moreover, there is additional delay caused by frequency and/or voltage adjustment caused by power saving mechanisms. The trade-off between power saving and performance is clearly in favour of the latter.

Fixed-Priority Spinlock The lock has thus been implemented as a spinlock, and in particular the ticket spinlock has been selected as a starting point, a well-known kind of spinlock that guarantees fairness by maintaining a waiting queue (tickets). More specifically, we opted to deliver the simplest and most efficient memex implementation, and selected a fixed priority scheduling policy.
As previously mentioned, exclusive loads and stores are a way to execute semaphores, where the load and store operations are non-atomic. The ISA guarantees that if previously loaded data is modified by another CPU, then the store fails and the load-store sequence must be retried. Algorithm 1 and 2 implement primitives of memex, a spinlock for memory mutual exclusion with fixed priority queue. The memex is non-preemptive—once a core holds the spinlock, the arrival of a demand from a higher priority core does not cause the former to release it.
Memex is a 32-bit long structure composed of two fields: queue [31:16] and owner [15:0]. `cpu_id` is a bit mask with the logical CPU identifier: only one bit is set and its position identifies the CPU. Implementation is in Listing 1.

Function *spin_lock* (Algorithm 1) tries to acquire the spinlock: if the spinlock is free, it writes its cpu_id in the owner field and goes directly to the critical section; otherwise, if another core is holding the spinlock, then it inserts its cpu_id into the queue and waits for its turn until the spinlock's owner field is set to its `cpu_id`. Function *spin_unlock* (Algorithm 2) releases the spinlock: it removes the id of the CPU core that is holding the spinlock from the queue and sets the owner field to the identifier of the core with the highest priority in the queue.

```
input : a lock id l, a CPU core id c
result: l is exclusively assigned to c
1  prefetchL1(l);
2  do
3      s ← loadExclusive(l);
4      if s = 0 then
5          s ← c;
6      else
7          s ← s | (c << 16);
8      end
9      ret ← storeExclusive(l, s);
10 while ret = failed;
11 if s = c then
12     return;
13 else
14     do
15         s ← loadExclusive(l);
16         if c = (s & 0x0000FFFF) then
17             return;
18         end
19         waitForEvent;
20     while 1;
21 end
```

Algorithm 1: Fixed priority spin lock algorithm

```
input : a lock id l, a CPU core id c
result: l is not exclusively assigned to c
1  do
2      s ← loadExclusive(l) s ← s | 0xFFFF0000    // clear owner
3      s ← s & not(c)    // remove c from queue
4      if s ≠ 0 then
5          s ← s | (1 << (16 › countLeadingZeros(s)))    // set new owner
6      end
7      ret ← storeExclusive(l, s)
8  while ret = failed;
```

Algorithm 2: spin_unlock.

Listing 2: Memex lock procedure

```
1  static inline void memex_lock(memexlock_t *lock, unsigned int cpu_id)
2  {
3         unsigned int tmp;
4         memexlock_t lockval;
5
6         asm volatile("
7         prfm   pstl1strm, %2
8  1:     ldaxr  %w0, %2
9         cbnz   %w0, 2f
10        stxr   %w1, %w4, %2
11        cbz    %w1, 4f
12        b      1b
13 2:     orr    %w0, %w0, %w4, lsl #16
14        stxr   %w1, %w0, %2
15        cbnz   %w1, 1b
16        sevl
17 3:     wfe
18        ldaxrh %w0, %3
19        eor    %w0, %w4, %w0
20        cbnz   %w0, 3b"
21        /* We got the lock. Critical section starts here. */
22 "4:"
23        : "=&r" (lockval), "=&r" (tmp), "+Q" (*lock)
24        : "Q" (lock->owner), "r" (cpu_id)
25        : "memory");
26 }
```

Listing 3: Memex unlock procedure

```
1  static inline void memex_unlock(memexlock_t *lock, unsigned int cpu_id)
2  {
3      unsigned int tmp1, tmp2;
4      memexlock_t lockval;
5
6         asm volatile("
7         lsl    %w4, %w4, #16
8  1:     ldaxr  %w0, %3
9         and    %w0, %w0, #0xFFFF0000
10        bic    %w0, %w0, %w4
11        cbz    %w0, 2f
12        clz    %w1, %w0
13        mov    %w2, #0xF
14        sub    %w1, %w2, %w1
15        mov    %w2, #1
16        lsl    %w1, %w2, %w1
17        orr    %w0, %w0, %w1
18 2:     stxr   %w2, %w0, %3
19        cbnz   %w2, 1b"
20        : "=&r" (lockval), "=&r" (tmp1), "=&r" (tmp2), "+Q" (*lock)
21        : "r" (cpu_id)
22        : "memory");
23 }
```

2.1.6 MemGuard Implementation

MemGuard is an extension of the Jailhouse hypervisor that can limit memory bandwidth consumed by a CPU core. As outlined above, it is used mainly in compatible PREM intervals to limit memory interference caused by running non-PREMized code. At the same time, it is also used in predictable intervals as a protection against misbehaving applications. The mechanism is controlled by a hypercall interface described in D4.5, which allows setting **memory budgets** and **timer periods**. Additionally, it allows obtaining statistics from application execution (profiling).

MemGuard uses the following features of ARMv8 hardware:

1. Performance Monitoring Unit (PMU) is used to monitor/count memory accesses (cache misses).
2. Hypervisor timer is used for replenishing memory budgets (resetting performance counters) and for blocking the execution in case of memory budget overruns.
3. Interrupt controller is optionally used to support non-preemptive execution of predictable PREM intervals. When requested, MemGuard masks most of the interrupts (besides those that are needed for MemGuard functionality and a few that cannot be masked).

MemGuard activity is not coordinated between individual CPU cores, as this would result in unexpectedly high overhead. However, even with independent per-CPU MemGuards, there is some overhead, which needs to be taken into account for schedulability analysis.

MemGuard overhead evaluation In this section, we experimentally validate MemGuard operation and evaluate its overheads. The experiments were run on an NVIDIA TX2 board. We ran several synthetic benchmarks and varied configurable parameters (budget and period), using MemGuard profiling facility to obtain execution time and the number of memory accesses (cache misses). The results can be seen in Figures 1 and 2.

Figure 1 shows a single experiment where MemGuard did non perform any throttling, therefore uniquely showing the overhead. MemGuard was called with different timer period before and after reading 10 000 cache lines (ca. 640 KB of memory). The call at the beginning sets the parameters, while the call at the end obtains the statistics.

As expected, the cache miss statistics (Fig. 1, top line) reported almost 10 thousands cache misses in each execution. However, when the timer period is shorter, one can see that a certain number of cache misses is "lost". This is caused by the fact (documented in ARM TRM) that PMU events can be counted with a delay and if the CPU enters the hypervisor in between, the event is not counted. The shorter the timer period, the more often the CPU enters the hypervisor (to handle the timer interrupt) and more events are lost. From the graph, one can see that even with extremely short period of 1 µs the difference from correct value is at most 5%.

The execution time statistics (Fig. 1, bottom line) show the overhead caused by the periodic timer interrupt. For periods greater than 150 µs the timer did not interrupt the execution. For smaller times, execution time is extended by the extra interruptions. For the extreme timer period of 1 µs, the overhead is about 30%. For more realistic periods, such as 16 µs, the overhead is only 2%.

Figure 2a shows the same results as Figure 1, but for different memory budgets. It can be seen that when the budget is smaller than 10 000, the execution is throttled. The amount of throttling depends almost linearly on the available memory bandwidth, i.e. the budget/time ratio.

Figure 2b shows a similar experiment, varying the memory budget in the horizontal axis, and using different lines to show different timer periods.

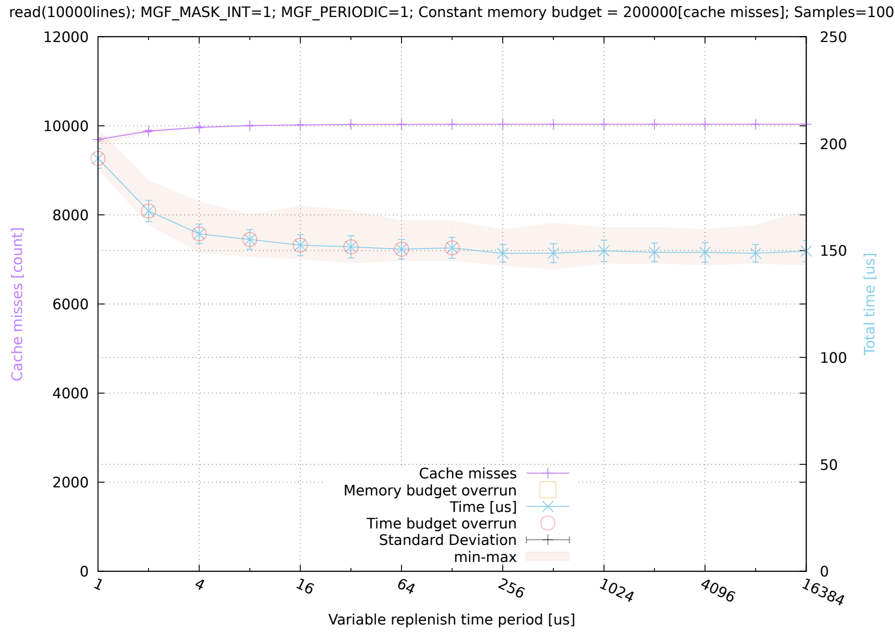
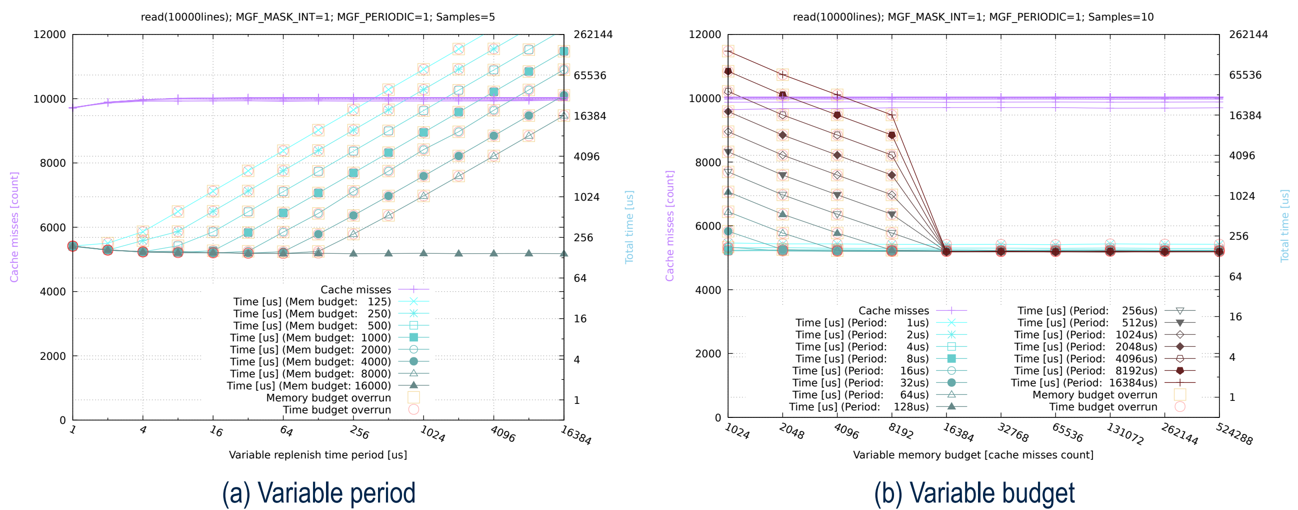


Figure 1: The effect of different MemGuard replenish time period



(a) Variable period (b) Variable budget

Figure 2: Effect of varying MemGuard parameters

2.2 Memguard Schedulability analysis

HERCULES framework implemented MemGuard, a memory bandwidth regulation mechanism for the multi-core platforms. Each core is guaranteed a given memory budget within a predefined time period. The software bandwidth controller monitors the number of memory accesses for each core and if any core exhausts its budget within its current period, the memory accesses from this core are throttled. Regulation period is the same for all the cores and the budgets are replenished synchronously at the beginning of the period. The tasks are statically partitioned among the processors.

In what follows, we introduce the formal model of the memory bandwidth regulation mechanism for real-time task. We identify two existing approaches in the literature for guaranteeing the schedulability of tasks executed under the HERCULES framework and specify the limitations of each approach.

2.2.1 Model

Task A system consists of a finite set of sporadic real-time tasks. Each task gives rise to a potentially infinite sequence of jobs. Task τ_i releases jobs sporadically after some minimum inter-arrival time T_i and each job of τ_i must be completed within a fixed time interval from its release given by a relative deadline D_i. We assume that tasks have constrained deadlines, i.e. $D_i \leq T_i$. The worst-case execution time C_i is the longest time needed to complete task τ_i running in isolation on a single core (no other tasks, all other cores are idle). The task alternates its status between memory accesses and computational operations. During the computational operations, each task performs the computation on the prefetched data without accessing main memory. During its execution, the task can have at most C_i^m memory accesses and C_i^e computational operations. Since the memory and computation phases are supposed to be non-overlapping, $C_i \leq C_i^e + C_i^m$ ($C_i = C_i^e + C_i^m$ for single path task). We also assume that the positions of the particular instructions requiring computation or memory are unknown (the order and the length of the particular memory and computation phases is unknown). All the above parameters are positive integers.Tasks are assumed to be independent and do not share resources other than processors and main memory. Preemption and context switch costs are assumed to be already accounted for in execution times.

As previously mentioned, two software mechanisms are implemented within the HERCULES framework to guarantee that the data prefetched during memory phases is persistently available during the subsequent computational phases. The first mechanism is the cache-coloring: it divides the L2 cache into multiple non-overlapping regions that are assigned to the particular tasks for their exclusive use, avoiding the problem of inter-task eviction (the preempting tasks or the tasks running on other cores evict the data being used by some other tasks). The second mechanism is the preventive invalidation: for L2 cache controllers with pseudo-random replacement policy, it guarantees that all the data requested during memory phase is actually fetched into the L2 cache, avoiding self-evictions.

We assume that tasks are statically partitioned among n processors and task-to-processor assignment is already given. Without loss of generality, we limit our attention to the set of n tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ assigned to the processor under analysis and scheduled by a fixed-priority preemptive scheduler. Tasks are indexed in priority order with τ_1 having the highest priority and τ_n the lowest priority. We denote as hp(i) and lp(i) the set of tasks with priorities higher (resp. lower) than the priority of task τ_i. Further, we denote as hep(i) and lep(i) the set of tasks with priorities higher than or equal (resp. lower than or equal) to the priority of task τ_i.

Platform Memory is a globally shared resource that can by accessed by all cores. All cores incur the same memory access latencies. We assume that each processor synchronously waits for every prefetch instruction caused either by cache miss or by explicit prefetch instruction (in reality it may concurrently execute out-of-order instructions). L is the maximum delay of a DRAM transaction for the core under analysis.

Each core has a reserved memory bandwidth guaranteeing a budget of Q_i memory accesses every period P. Regulation period P is a system-wide parameter that is the same for all the cores. The budgets are replenished synchronously at the beginning of the period. The sum of all cores budgets is less than the regulation period:

$$\sum_{i=1}^{n} Q_i \leq P. \tag{1}$$

2.2.2 Related Work

In [Yun+12], Yun et al. propose a schedulability analysis for a setting where a critical core executing hard real-time tasks can always access memory, i.e., with an unbounded memory budget. On the interfering cores, the number of memory accesses is throttled with a budget Q_i over a period P, which is the same for all the cores. Bus arbitration is based on a round-robin policy: each memory access can be delayed by one memory access from each other core. The authors propose a Response Time Analysis for the real-time tasks executed on the critical core with a deadline monotonic scheduler. Each task is characterized by its worst-case execution time and the total number of memory accesses. Furthermore, a method for finding the throttling parameters to satisfy the schedulability of hard real-time tasks is proposed.

Yao et al. generalize the framework proposed by Yun et al. Memory accesses from all the cores, including the critical one, can be regulated. The cores budgets are replenished synchronously on all cores at every period (synchronized regulation). The budgets of the cores interfering with the core under analysis are supposed to be evenly distributed, as this is shown to represent the worst-case scenario.

2.2.3 Schedulability Analysis Selection for Hercules

The schedulability analysis proposed by Yun et al. [Yun+12] can be applied for the Hercules framework if the following conditions are fulfilled.

- The core under analysis, $i = 1$, has an infinite memory budget: $Q_1 \to \infty$, i.e., it cannot be throttled. The other cores, $i \neq 1$, do not execute hard real-time tasks, as the method does not allow verifying the task schedulability of tasks whose memory accesses are throttled.

To apply the schedulability analysis proposed by Yao et al. [Yao+16b], the following conditions must be met:

- The core under analysis, $i = 1$, has a finite memory budget, Q_1, and the other cores must have identical budgets $Q_2 = Q_3 = \ldots = Q_n$. If the requirement of the equal budgets cannot be satisfied, then it can be assumed in the analysis that for $i \neq 1$, $Q_i' = \sum_{i=2}^{n} Q_i/(n-1)$. As proved by Yao et al. [Yao+16b], a safe upper bound is found under the assumption of equally distributed budgets for the interfering cores, as this always represents the worst-case configuration. If the hard real-time tasks are executed on more than one core, then the analysis should be separately applied to each core.

2.2.4 Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems

We recapitulate the schedulablity analysis for memory bandwidth regulated multicore real-time systems proposed by Yao [Yao+16b].

The main assumptions are in line with those stated in Section 2.3.1. The memory accesses are arbitrated according to the Round Robin policy. Each core has its own last level cache partition. All the cores have the same regulation period P. The positions of the memory accesses and computations within the task and their

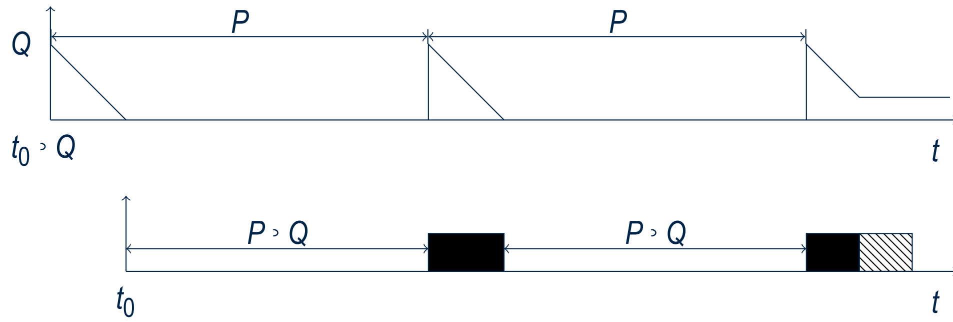


Figure 3: Task worst-case arrival. Black rectangles - task memory phase, hatched rectangles - computation phase.

particular order are not known a priori. Memory and computation phases of the task are supposed to be non overlapping. Tasks are periodic. Tasks releases are not synchronized with the memory regulation period. The core under analysis has budget Q (maximum allowed memory access time within one period).

Definition 1. *Stall time is the core's waiting time due to the interference from the other cores or the regulatory mechanism.*

Two separate cases are considered in the next paragraphs: (i) core stalls due to memory bandwidth regulation alone: no interference from the other cores; and (ii) core stalls due to inter-core contention alone: with interference from the other cores. For the sake of simplicity and ease of presentation, it will be considered that only one task is running on the analyzed core.

Core stalls due to memory bandwidth regulation alone Suppose that the other cores do not generate memory accesses. The core under analysis is the only one to access the main memory. Once the core has used up all its budget, the core is stalled and must wait until the end of the period to have its budget replenished.
It is shown that the worst-case is when: (i) the task is released Q time units after the budget replenishment and the budget has been just exhausted, (ii) all the memory accesses are grouped at the beginning of the task.
Every Q accesses the task is stalled for time $P - Q$. Figure 3 shows this case. The detailed proof can be found in [Yao+16b] Section 3.1.

Core stall due to inter-core contention alone Now suppose that the other cores do generate memory accesses and the budget of the analyzed core cannot be fully depleted within a single regulation period. Under Round Robin policy, a single memory access of the analyzed core can be delayed at worst by $(m - 1) \cdot L$ time units. It happens when all the other cores access the memory at the same time as the analyzed core does. We introduce *Remaining Budget Share* as an equal share of the remaining budget $P - Q$ over $m - 1$ remaining cores:

$$RBS = \frac{P - Q}{m - 1} \quad (2)$$

The worst-case is when: (i) the maximal amount of the remaining budget is allocated to the interfering cores, i.e., $P - Q$, (ii) the remaining budget is distributed evenly among the interfering cores, i.e., $\forall j \neq i : Q_j = RBS$.
The property stated in ()i) is trivial: decreasing the budget of the interfering cores cannot increase the interference. The key observation behind the property stated in (ii) is following: the excessive part of the budget on the interfering core (i.e., which is greater than Q) cannot cause interference on the core under analysis, assigning

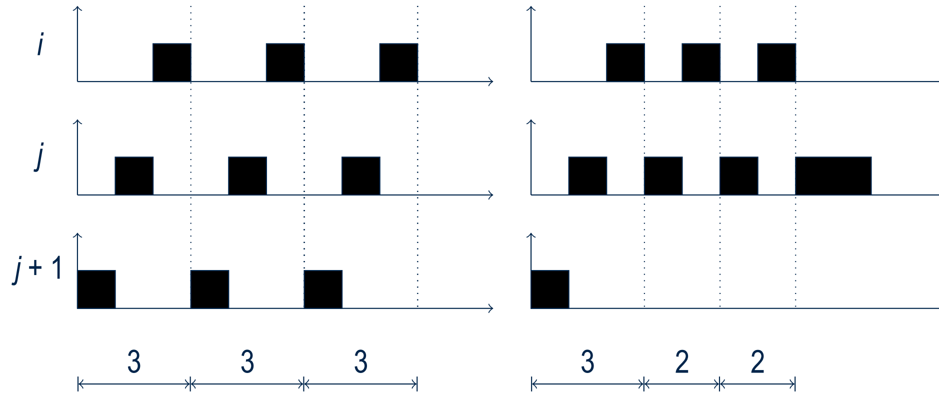


Figure 4: On the left: evenly distribute memory budgets of the interfering cores j and $j+1$; the total interference on the core i is maximal. On the right: the core $j+1$ has more memory budget than the core j; the total interference on the core i is reduced.

this excessive part to the core that have less budget than Q can increase the number of memory accesses in conflict with those from the core under analysis. Figure 4 illustrates this concept. If the core generates RBS memory accesses, then every access can be delayed by $(m-1)$ memory accesses from the other cores causing the total stall of $(m-1) \cdot RBS = P - Q$. The stall due to the memory contention cannot be greater than that.

Single task stall analysis In what follows, an analysis for a single task is presented. The analysis is based on the following observations:

- At any time instant, a task can either request a memory access or a computation.
- Within a single period, Q memory accesses cause the stall of $P - Q$ due to the memory regulation mechanism (Subsection 2.2.4).
- Within a single period, the RBS memory accesses suffer the delay $(m-1) \cdot RBS = P - Q$ from the other interfering cores, the later accesses do not suffer any interference (Subsection 2.2.4). The remaining time to the end of the period is $Q - RBS$.

We consider two cases depending on the relationship between the memory budget of the analyzed core and the memory budgets of the interfering cores.

1. Suppose that $Q < RBS$. Figure 5 illustrates this case. If the analyzed core generates Q memory accesses within a single memory regulation period then the core will be suspended by the memory bandwidth regulator for time $P - Q$. The worst-case is when the task in every regulation period generates Q consecutive memory accesses and, once there are no more memory accesses, it does the computation.

   The memory stall $stall$ can be computed as follows:

$$\begin{aligned} stall(C^m, Q, P, m) = & initial_stall(Q, P) + regulation_stall(C^m, Q, P) \\ & + interference_stall(C^m, Q, m) \end{aligned} \quad (3)$$

   where:

$$initial_stall(Q, P) = P - Q \quad (4)$$

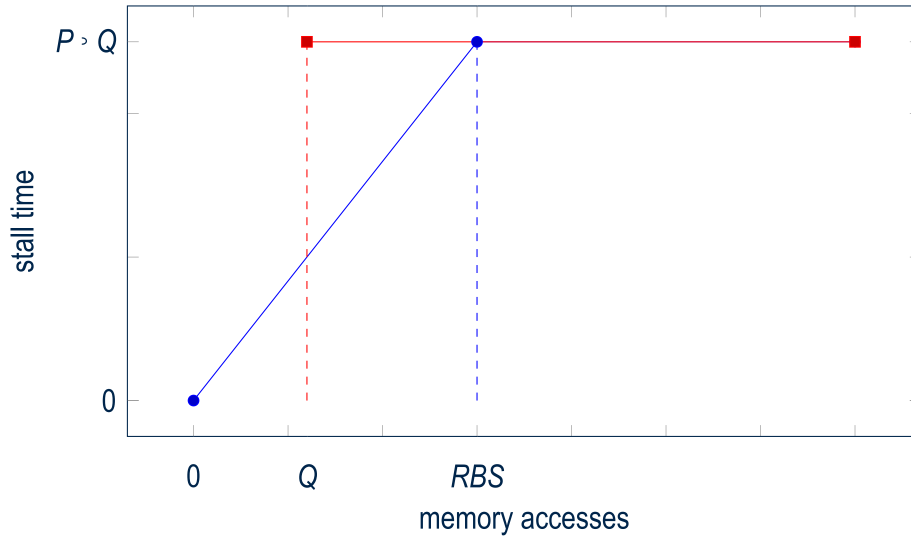


Figure 5: Stall time as a function of memory accesses. Red line - stall time due to the memory regulation. Blue line - stall time due to the inter-core memory contention.

is the initial stall due to the task arrival after the budget replenishment (see Figure 3 and Subsection 2.2.4),

$$regulation_stall(C^m, Q, P) = \left\lfloor \frac{C^m}{Q} \right\rfloor \cdot (P - Q) \tag{5}$$

is the regulation stall of $(P - Q)$ time units due to the budget depletion in $\left\lfloor \frac{C^m}{Q} \right\rfloor$ regulation periods, and

$$interference_stall(C^m, Q, m) = C^m \mod Q \cdot (m - 1) \tag{6}$$

is the interference stall from other cores that occurs when the task has no sufficient number of memory accesses to fully deplete the memory budget.

2. Now suppose otherwise that $Q \geq RBS$. Figure 6 illustrates this case. The worst-case is when RBS memory access (that are causing $RBS \cdot (m - 1)$ delay due to the inter-core contention) are at the beginning of every regulation period and later, if possible, the time from $RBS \cdot m$ to the end of the period is for the computation. By allocating memory and execution in such manner, the memory accesses are always exposed to the highest interference from the other cores. However, there may not be enough computation time to fill all the spaces between the subsequent regulation periods. In such case, some of the memory must be placed in the intervals in which there is no interference from the other cores (i.e., from $RBS \cdot m$ to the end of the regulation period).

   We introduce two coefficients:

$$\alpha_e = \frac{C^e}{Q - RBS}, \quad \alpha_m = \frac{C^m}{RBS} \tag{7}$$

   2.1. First, suppose that there is enough computation to place the memory accesses in such a way that every single access is exposed to the maximal interference. For every RBS memory accesses, apart the last regulation period, we need $P - m \cdot RBS = P - (m - 1) \cdot RBS - RBS = Q - RBS$ computation time. In the last regulation period we can have RBS memory accesses with full interference without the need of computation time.

$$\lceil \alpha_m \rceil - 1 \leq \lfloor \alpha_e \rfloor \tag{8}$$

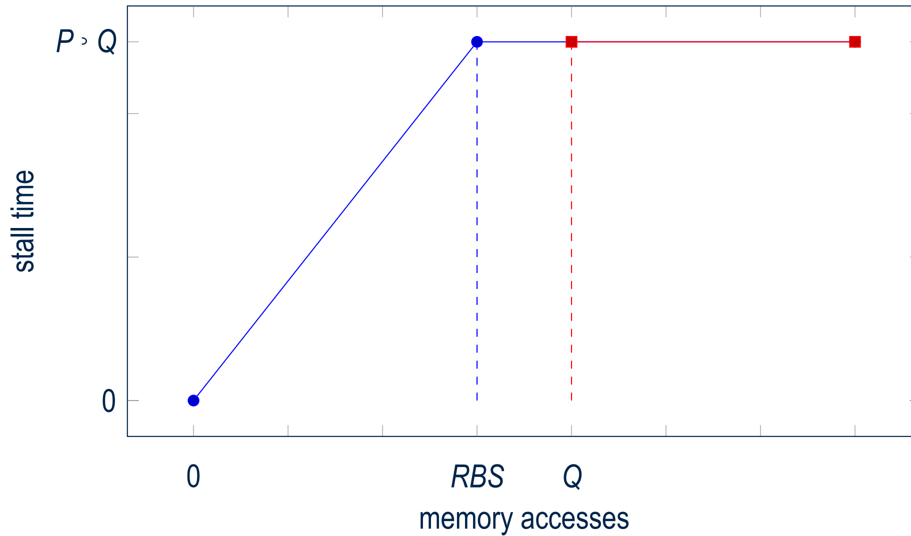


Figure 6: Stall time as a function of memory accesses. Red line - stall time due to the memory regulation. Blue line - stall time due to the inter-core memory contention.

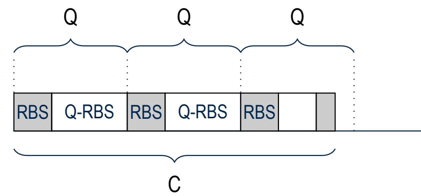


Figure 7: $\left\lceil \frac{C}{Q} \right\rceil = \lceil \alpha_m \rceil$, memory and computation finish in the same regulation period.

In every regulation period, each memory access has the greatest interference:

$$stall(C^m, Q, P, m) = initial_stall(Q, P) + C^m \cdot (m - 1) \tag{9}$$

2.2. Now, suppose that there is not enough of the computation, and some of the memory accesses will not suffer the maximal interference from the other cores:

$$\lceil \alpha_m \rceil - 1 > \lfloor \alpha_e \rfloor \tag{10}$$

In the first $\lfloor \alpha_e \rfloor$ regulation periods, there is enough computation to fill the time from $m \cdot RBS$ to the end of the period.

2.2.1. Consider the case in which the task terminates in the $(\lfloor \alpha_e \rfloor + 1)$-th regulation period (see Figure 7): the computation and memory accesses are used up in the last regulation period. Each of the first $\lfloor \frac{C}{Q} \rfloor$ regulation periods is filled with RBS memory accesses and $Q - RBS$ computation causing the stall of $P - Q$. In the last regulation period, only RBS memory accesses are interfered. The remaining part of the task $C \bmod Q - RBS$ may be both, computation or memory accesses, but will cause no additional interference. The total stall is given by:

$$stall(Q, P, C) = initial_stall(Q, P) + \left(\left\lfloor \frac{C}{Q} \right\rfloor + 1 \right) \cdot (P - Q) \tag{11}$$

2.2.2. Consider the case in which the task terminates after the $(\lfloor \alpha_e \rfloor + 1)$-th regulation period (see Figure 8): there is enough computation and memory accesses to continue the task execution

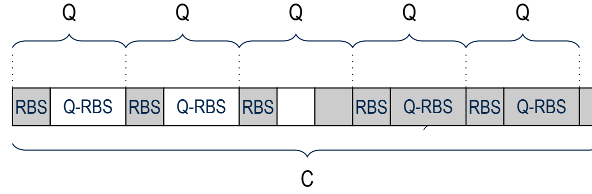


Figure 8: $\left\lceil \frac{C}{Q} \right\rceil > \lceil \alpha_m \rceil$, memory finishes in the later regulation period than the computation does.

over to the next regulation period. In the subsequent regulation periods there are only memory accesses. The task will be stalled after every Q memory accesses causing the delay of $P - Q$. Hence, the stall in the first $\left\lfloor \frac{C}{Q} \right\rfloor$ regulation periods is $P - Q$. In the last regulation period, the stall is $C \bmod Q(m-1)$ but cannot be greater than its maximal value $P - Q$. The total stall can be given as:

$$stall(Q, P, C, m) = initial_stall(Q, P) + \left\lfloor \frac{C}{Q} \right\rfloor \cdot (P - Q) + last_period_stall(Q, P, C, m) \quad (12)$$

where

$$last_period_stall(Q, P, C, m) = \min\left(\left(m - 1\right) C \bmod Q, P - Q\right) \quad (13)$$

Note that Equation (12) is equivalent to Equation (11) since it is supposed in Equation (11) that in the last regulation period the task generates *RBS* memory accesses causing the stall $P - Q$.

Task set analysis The previously obtained results are now applied for a task set of n tasks. The analysis for a whole task set is done by considering the continuous execution as one single task instance with the following parameters:

$$C^m = \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j^m \quad (14)$$

$$C^e = \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j^e \quad (15)$$

The response time of task τ_i is given by:

$$R_i = C_i + \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + stall(C^m, C^e, Q, P, m). \quad (16)$$

The formula can be solved iteratively.

2.3 Integrated PREM Schedulability analysis

Modern embedded platforms are composed of multiple active components, such as CPU cores and I/O devices, sharing among them various hardware and software resources. In particular, read and write requests issued by these components contend for the main memory. When each device accesses main memory at the same time,

system performance and data latency times can be heavily affected, leading to significant timing delays for the executing tasks.

HERCULES framework provides two key mechanisms to guarantee exclusive memory accesses for each component in the multi-core platforms: priority-queue based spinlock and memguard. Controlling the accesses to the main memory allows avoiding the bottleneck problems that may cause a significant slowdown to memory intensive tasks. The spinlock is used to guarantee that only the core that holds it is allowed to access the main memory in an unrestricted manner. Priorities are used to decide the next core from the waiting queue to acquire the spinlock. The other cores may have a limited number of accesses to the main memory over a specified period regulated by the memguard mechanism.

We present a simple timing analysis for tasks split into memory and computation phases running on a multiprocessor platform with the HERCULES framework. In our analysis, we take into account the waiting time incurred by the core when getting access to the main memory through the above-referred spinlock mechanism and the interference caused by the memguarded cores.

2.3.1 Task Model

A system consists of a finite set of sporadic real-time tasks. Each task gives rise to a potentially infinite sequence of jobs. Task τ_i releases jobs sporadically after some minimum inter-arrival time T_i and each job of τ_i must be completed within a fixed time interval from its release given by a relative deadline D_i. We assume that tasks have constrained deadlines, i.e. $D_i \leq T_i$. The worst-case execution time C_i is the longest time needed to complete task τ_i running in isolation on a single core (no other tasks, all other cores are idle). The task executes first its memory phase (M-phase) and then its computational phase (C-phase). During memory phase, task reads and writes its data to the main memory. During computational phase, task performs the computation on the prefetched data without accessing main memory. The maximum time to complete the memory phase with exclusive access when there is no interference from other cores is C_i^m. The worst-case execution time for the computation phase is instead C_i^e. Since the memory and computation phases are supposed to be non-overlapping and the task to follow a single path: $C_i = C_i^e + C_i^m$. All the above parameters are positive integers. The tasks are assumed to be independent and do not share the resources other than processors and main memory. The preemption and context switch costs are assumed to be already included in the worst-case execution times.

We suppose that the tasks are statically partitioned among m processors and task-to-processor assignment is already given. The set of n tasks $\mathcal{T}^k = \{\tau_1, \ldots, \tau_n\}$ assigned to the k-th processor is scheduled by a fixed-priority preemptive scheduler. Tasks are indexed in priority order with τ_1 having the highest priority and τ_n the lowest priority.

2.3.2 Platform Model

Also in this section we adopt the mechanisms implemented in the HERCULES framework to achieve the spatial and temporal isolation of the main memory on the multicore platforms, as described in Section2.3.1. Namely, cache-coloring and preventive invalidation.

Exclusive memory access To grant an unrestricted access to the main memory for only one core at a time, we adopt a *spinlock* with priority-based queue. Before starting a memory phase, a task must first obtain the spinlock. The spinlock is non-preemptive: the core that holds the spinlock does not release it when a higher priority task arrives. The requests from cores trying to acquire the spinlock while it is held by another core are stored in the queue. Once the spinlock is released, it is granted to the core with the highest priority in the queue. The k-th core is attributed a distinct priority level P_k. We introduce notation $hep(P_k)$ and $lep(P_k)$ for the set of cores with priorities higher than or equal (resp. lower than or equal) to the priority of core P_k.

When the spinlock is in use, the other cores can do only a limited number of memory accesses, regulated by a Memguard mechanism where each core has a memory access budget Q_i for every period P. Regulation period P is a system-wide parameter which is the same for all the cores. The budgets are replenished synchronously at the beginning of the period. As in the previous section, the sum of all cores budgets is less than the regulation period time:

$$\sum_{i=1}^{n} Q_i \leq P. \tag{17}$$

Also, all cores incur the same memory access latency L, and there is no out-of-order execution.

2.3.3 Related Work about Memory-Computation Tasks

Yao et al. [Yao+16a] proposed a task model in which each job consists of exactly three phases: memory - execution - memory. The tasks are scheduled on a multicore platform with a global scheduler that promotes the memory accesses, i.e., tasks in a memory phase have a higher priority than tasks in an execution phase. A number of tasks running on different cores can simultaneously access the memory bus incurring the same memory access latency. In [Sha+16], the notion of Single-Core Equivalence is proposed integrating PALLOC (OS-level memory allocator), MemGuard (OS-level) memory bandwidth manager, colored lockdown with offline profiling (fetching into the cache only the hot pages) and algorithms for application partition assignment for integrated modular avionics.

Melani et al. [Mel+17; Mel+15] proposed a new task model composed of two consecutive phases: a memory phase in which the task prefetches the required data into the local memory and a computation phase in which the task executes the instructions without memory contention. The exact response time analysis is proposed for a single-core system. Memory phase is fully preemptable. Memory and computation phases of two different tasks may overlap (a task can access memory while another one is executing). The memory accesses are assumed to be scheduled according to task priorities.

In [Pel+11; Pel+12], the Predictable Execution Model (PREM) is proposed, dividing jobs into sequences of non-preemptive scheduling intervals. A scheduling interval can be compatible or predictable. Compatible intervals may have cache misses, since they are compiled without any special provision. Instead, predictable intervals start with a memory phase, where the task prefetches data into the local memory (cache), followed by an execution phase, where the task computes cached data without incurring any cache miss. The length of the predictable interval is constant. The peripheral traffic is scheduled during the execution phase when tasks do not modify the main memory (I/O flows are scheduled both during execution phases and while the CPU is idle). The tasks are scheduled according to the Deadline Monotonic policy on a single processor. The schedulability of the task is verified with a response time analysis, taking into account the blocking time that the task may suffer due to lower priority tasks executing their non-preemptive regions. Another analysis is proposed for peripheral scheduling. It takes into account that the I/O flows can access the main memory only when the tasks are into their execution phases. The first result is obtained for periodic tasks, while the second one for sporadic tasks.

gPREM [AP14] is a global scheduling algorithm for PREM tasks executed on multicore systems. It extends global non-preemptive fixed priority scheduling by co-scheduling memory accesses and CPU execution. Only one task in the entire system is allowed to run a memory phase. Other tasks can run in parallel on the remaining cores if and only if they are running in their computation phases. A new task can start to execute if the memory is available and at least one processor is idle.

2.3.4 Schedulability Analysis for PREM with priority-queue spinlock and memguarding

For simplicity, we present a schedulability analysis for the case where each core has one running task. The analysis can be easily extended to multiple tasks settings. The approach is similar to the feasibility analysis for tasks with fixed preemption points proposed by Bril et al. [BLV09] (see also [BBY13] for a survey).

First, we consider the task execution with memguard budgets set to zero (i.e., task in the memory phase cannot be interfered by tasks running on the other cores).

We introduce the blocking factor B_i for the i-th core as the maximal time that a lower priority core can hold the spinlock:

$$B_i = \max\left\{c_j^m \mid j \in lp(i)\right\}. \tag{18}$$

In the worst-case scenario, the job running on core i may be blocked before acquiring the spinlock by only one job from a lower priority core. It may also be interfered by all jobs from the cores with higher priority. The start time $s_{i,k}$ of the k-th instance of task τ_i running alone on the core i is given as:

$$s_{i,k} = B_i + (k - 1) \cdot c_i^m + \sum_{j \in hp(i)} \left\lceil \frac{s_{i,k}}{T_j} \right\rceil c_j^m \tag{19}$$

Since, once started, the memory phase cannot be preempted and the computation phase starts right after the end of the memory phase, the finishing time $f_{i,k}$ is:

$$f_{i,k} = s_{i,k} + C_i \tag{20}$$

Due to the self-pushing effect described in [BLV09], the worst-case response time is not given in the first instance after a critical instant, but one needs to check all τ_i's instances within a level-i busy interval L_i, whose length can be computed as

$$L_i = B_i + \sum_{j \in hep(i)} \left\lceil \frac{L_i}{T_j} \right\rceil c_j^m \tag{21}$$

We now consider the task execution with non-null memguard budgets (i.e., task in the memory phase can be interfered by the tasks running on the other cores).

Each memory access from the analyzed core under Round Robin discipline can be delayed by at most one access from every interfering core. Yao et al. [Yao+16b] proved that the worst-case configuration is given when the budgets of the interfering cores are equally distributed.

We define q_i as the number of the memory accesses from any single interfering core (other than the i-th core) when the memguard budgets of the interfering cores are equally distributed:

$$q_i = \frac{\sum_{j \neq i} Q_j}{n - 1} \tag{22}$$

Given any contiguous time interval $t > 0$, the total number of memory accesses from the interfering core is given by:

$$\beta(t) = \left\lfloor \frac{t}{P} \right\rfloor q_i + \min\left\{(t \bmod P)/L, q_i\right\} \tag{23}$$

The memory accesses issued during the time interval $s_{i,k}$ (see Equation (19)) can be delayed by at most $n - 1$ interfering cores. The start time $s'_{i,k}$ of the k-th instance of task τ_i running alone on the core i interfered by the

memguarded cores is given as:

$$s'_{i,k} = \beta \left(B_i + (k-1) \cdot c_i^m + \sum_{j \in hp(i)} \left\lceil \frac{s'_{i,k}}{T_j} \right\rceil c_j^m \right) L + B_i + (k-1) \cdot c_i^m + \sum_{j \in hp(i)} \left\lceil \frac{s'_{i,k}}{T_j} \right\rceil c_j^m \tag{24}$$

In the same way, we can rewrite Equation (21) for the busy interval:

$$L'_i = \beta \left(B_i + \sum_{j \in hep(i)} \left\lceil \frac{L'_i}{T_j} \right\rceil c_j^m \right) + B_i + \sum_{j \in hep(i)} \left\lceil \frac{L'_i}{T_j} \right\rceil c_j^m \tag{25}$$

Equation (20) still holds for the computation of the task finishing time.

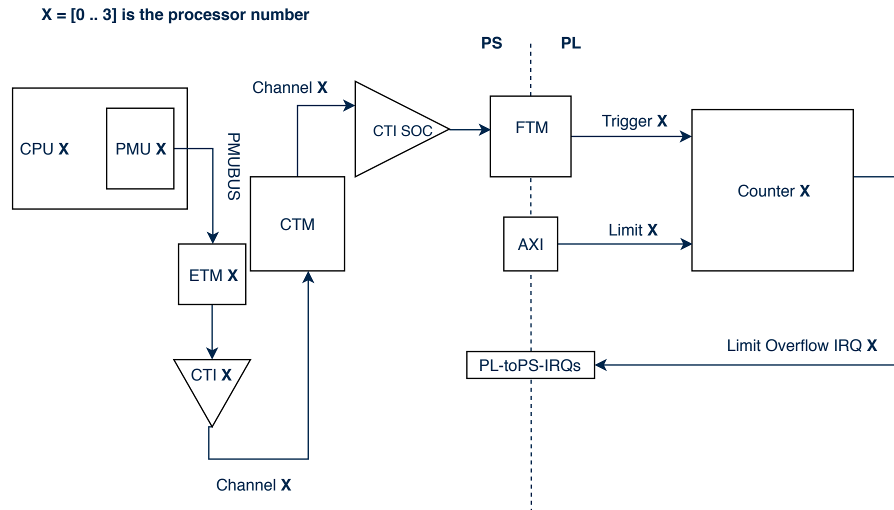


Figure 9: The PREM hardware watchdog concept overview

3 FPGA-based approach to PREM scheduling

The PREM implementations based on Memex Spinlock and MemGuard (described in previous sections) involve various overheads. While those overheads are in a range of a few percent, some performance sensitive applications may not want to pay this price. For that reason, we have started investigation of FPGA-based hardware acceleration of PREM scheduling and memory access monitoring. Here, we compare the prototype implementation with the hypervisor-based solution.

The stated FPGA-based hardware acceleration was successfully implemented on Xilinx Zynq Ultrascale+ MPSoC platform using the abilities of ARM's CoreSight Debug & Trace system. The results show that in case of using FPGA-based memory watchdog maintenance takes 2.88 times less than what the hypervisor-based solution requires in average (the hypercall time).

Figure 9 provides an overview of the final experimental design. It illustrates that every CPU's core has its private counter in Programmable Logic (PL) (**X** denotes the CPU ID: e.g., the CPU**3** is connected to Embedded Trace Macrocell (ETM) **3**, and Cross-Trigger Interface (CTI) **3** is connected to Cross-Trigger Matrix (CTM) at the **third** channel and so on). Counter X counts and acknowledges triggers through Fabric Trace Macrocell (FTM) Trigger interface. Counter **X** is an Advanced eXtensible Interface (AXI) slave, so the message about Limit for processor **X** is delivered from SW through memory mapped interface. Finally, every Counter sends an interrupt through the provided PL to Processing System (PS) interrupt delivery interface in case of the given limit is overflown.

More details about the implemented prototype and its evaluation can be found in [Bar18].

4 Off-line scheduling and analysis of PREM execution

In contrast to on-line scheduling, the use of off-line (time-triggered) scheduling ensures that all timing parameters are known beforehand and no schedulability analysis is necessary. Our experiments with real hardware show that the off-line computed worst case completion time matches measured completion times.

We proposed an Integer Linear Programming (ILP) formulation of the scheduling problem in [D5.1]. To create efficient schedules for large-scale instances, we extended a state-of-the-art heuristic algorithm proposed by Hanzalek [HŠ17]. As we already shown in [D5.1], the off-line (static) scheduling in combination with PREM significantly improves execution time jitter. The core of the heuristic is a priority-based list scheduling algorithm with unscheduling step, which can remove already scheduled tasks conflicting with the task currently being scheduled. To achieve near-optimal solutions on large-scale instances, this basic algorithm uses a few additional techniques that improve the results. One such a technique is the time symmetry mapping, which allows construction of schedules in both forward and backward time orientation. Others are propagation of information about conflicting tasks into subsequent iterations, and parallelization of the algorithm that enables reduction of solving time. Our main contribution in this work is the extension of the algorithm to support multi-capacity take-give resources (take-give resources are described in [D5.1]).

4.1 Algorithm overview

The proposed heuristic is illustrated in Algorithm 3. After initialization (lines 1–6), a bounded amount of while loop iterations (from line 7 on) is performed. The goal of the loop is to iteratively tighten the bounds maintained by the algorithm. In each iteration a decision problem "does a solution with these bounds exist?" is solved and new instances for next iterations are created. The best solution found during the iterations is the output of the heuristic. The algorithm consists of the following building blocks:

Initialization During the initialization phase (lines 1–6), the algorithm precalculates maximal distances $d_{i,j}$ between all activities in a graph and sets initial lower bound $LB = d_{0,n+1}$ and upper bound $UB = \sum_{i \in \mathcal{V}} p_i$. First item is enqueued into the solution queue.

Solution queue Input to each iteration is a tuple (I^C, C, *priority*), where I^C is the current instance, C is the requested maximal schedule length and *priority* is a vector of priorities of length $n + 2$. These tuples are stored in a queue denoted as *schedulingParameters*. Although the algorithm could be formulated in a more compact and elegant way using a recursion, we use the queue because it enables easy parallelization of the algorithm. The queue-based formulation also allows the possibility that multiple tuples in queue result in the same solution after several iterations. To avoid solving of redundant instances, the algorithm computes and stores a hash of the priority vector in a hash table, and skips future queue entries with the same hash.

Find schedule The core of the algorithm is the function *findSchedule* called at line 14 and shown in Algorithm 4. It tries to solve the decision problem mentioned in the overview by transforming the vector *priority* into a schedule S that has its completion time ($C_{\max}(S)$) smaller than C.

The function iterates until a feasible schedule is found or the *budget*, which depends on the number of activities, is depleted. In each iteration, an activity with the highest priority which is not scheduled yet is chosen and the earliest possible start time ES_i and the latest possible start time LS_i of the activity i is calculated. Then the *findTimeSlot* function tries to fit the activity into the current schedule at an earliest possible time slot with respect to already scheduled activities and dependencies. If a free slot is found, the start time s_i of the activity is set

```
    input  : an instance I
    output: best schedule S^best
1   Calculate d_{i,j} ∀(i, j) ∈ V^2;
2   Calculate LB and UB; C = (LB + UB)/2;
3   S^best = ∅; priority(i) = d_{i,n+1}   ∀i ∈ V;
4   failureCounter = 0; priorityHashSet = ∅;
5   I^forward = I; I^backward = TimeSymmetryMapping (I);
6   schedulingParameters.Enqueue({I^forward, C, priority});
7   while LB < UB and |schedulingParameters| > 0 do  // the stopping condition
8       {I^c, C, priority} = schedulingParameters.Dequeue();
9       hash = hash (priority);
10      if hash ∈ priorityHashSet and C > UB then
11          continue;  // this tuple was already processed
12      end
13      priorityHashSet.Add(hash);
14      S = findSchedule(I^c, C, priority);
15      if S is feasible then
16          if UB > C_max(S) then  // a new better solution was found
17              UB = C_max(S);
18              LB = min(LB, ⌈0.8 · UB⌉);
19              S^best = S;
20          end
21          C^new = UB − 1;
22          priority = (C − s_i)_{i∈V};  // get new priority from the start time
23      else
24          failureCounter + +;
25          if failureCounter ≥ log_2(n) then
26              LB = LB + (UB − LB)/4;
27              failureCounter = 0;
28          end
29          C^new = min(UB, ⌈1.1 · C⌉);
30      end
31      priority^1 = (C − priority_i)_{i∈V};  // reverse priority
32      priority^2 = modifyPriority (priority, S);  // modify original priority
33      if I^c == I^forward then  // change orientation of the instance
34          I^{c1} = I^backward;
35      else
36          I^{c1} = I^forward;
37      end
38      I^{c2} = I^c;
39      schedulingParameters.Enqueue({I^{c1}, C^new, priority^1});
40      schedulingParameters.Enqueue({I^{c2}, C^new, priority^2});
41  end
```

Algorithm 3: Iterative Resource Scheduling algorithm

so that the activity fits into the schedule, and the activity is added into the set of scheduled tasks (no conflicting activity exists). Otherwise, the task is forced into schedule with start time s_i and all conflicting activities (tasks that potentially block the insertion of the current task into the schedule) are unscheduled. When the finding of the free time slot is unsuccessful for the first time, the start time is set to ES_i, otherwise $s_i = s_i^{prev} + 1$ where s_i^{prev} is the previous start time.

```
findSchedule(I, C, priority)
s_i = −∞ ∀i ∈ V;
scheduled = ∅;
budget = budgetRatio · n;
while budget > 0 and |scheduled| < n + 2 do
    i = arg max_{∀i∈V:i∉scheduled} (priority_i);
    ES_i = max_{∀j∈V:j∈scheduled} (s_j + d_ji);
    LS_i = C − p_i;

    {slotFound, s_i} = findTimeSlot(i, ES_i, LS_i);
    if !slotFound then
        s_i = s_i^prev + 1;
    end

    unscheduled = unscheduleConflictingActivities(i, s_i);
    scheduled = scheduled \ unscheduled;
    scheduleActivity(i, s_i);
    scheduled = scheduled ∪ {i};

    budget = budget − 1;
end
return (s_i)_{i∈V};
```

Algorithm 4: Priority-rule based function with an unscheduling step

Schedule evaluation If the feasible schedule is found, new bounds based on $C_{max}(S)$ and new priorities are calculated. The C^{new} is decreased to $UB - 1$ and when the schedule is the best solution so far, it is stored in S^{best}, the UB is updated to C_{max} and LB is decreased to $\min(LB, \lceil 0.8 \cdot UB \rceil)$ which gives the algorithm more time to find a better solution. In the second case, the algorithm increases C^{new} to $min(\lceil 1.1 \cdot UB \rceil, \lceil 1.1 \cdot C \rceil)$ in order to allow the escaping from the current local optimum. The algorithm also counts unsuccessful iterations and when the *failureCounter* exceeds a given threshold, the LB is increased to $LB + (UB - LB)/4$ in order to fulfill the stopping condition after limited amount of unsuccessful iterations.

The *priority* vector is changed in every algorithm iteration and it progressively converges to the priorities, which allow to find a better solution in latter iterations. If the schedule S is feasible, the priorities are updated according to the start times of activities in the schedule – the latter start time, the lower priority.

It does not matter whether the *findSchedule* function finds a feasible schedule or not. In both cases two new tuples are generated and inserted into the queue. The first tuple contains always the time symmetric instance and appropriately reversed vector of priorities. In the second tuple is an instance with preserved orientation and modified priorities with respect to the most frequently conflicting activity during the previous schedule construction. During the schedule creation, the *findSchedule* function registers conflicts between activities, which are subsequently evaluated. For the activity couple with the maximum number of conflicts, the priorities are swapped and higher priority is propagated backward into priorities of previous activities so that while previous activities have lower priority, the priorities are being increased by a difference between the swapped priorities.

Time symmetry mapping Time symmetry mapping transforms input instance so that activities are scheduled in reverse time orientation (from activity $n + 1$ to 0). The process of conversion is explained more in detail in [HŠ17].

4.2 Off-line scheduling evaluation

The evaluation section begins with an description of our Advanced Driver-Assistance Systems (ADAS) inspired PREM kernels, from which we composed several batches of experiments. Such batches are instantiated with different parameters to create a number of use-case scenarios. Following this, we validate the correctness and evaluate the performance of the heuristic algorithm and the whole toolchain. For given scenarios, we generated PREM compliant code by using the Hercules compiler, profiled the resulting code to get execution times of generated PREM intervals, generated a schedule by using the heuristic, and run experiments on NVIDIA Jetson TX1 board (based on ARM Cortex A57 processor). We use Linux 3.16 to run the experiments, and to establish the predictable behavior required for the PREM model, we implemented system calls for temporary disabling / enabling of interrupts on the selected core and for flushing and invalidating the entire cache. We measured execution times and the numbers of cache misses in particular intervals by using the performance monitor unit (event `L2D_CACHE_REFILL` and `PMCCNTR` register) in user space. The heuristic algorithm was implemented in `C#` and gives start times which define sequencing of memory intervals in our test bed similarly as the ILP solver for small instances.

4.2.1 Kernels

For the evaluation, five kernels are used, from which different scenarios based on real use cases are constructed. The first three kernels were already described in [D5.1] (General Matrix Multiplication (GEMM), Fast Fourier Transform (FFT), and binary tree search). In addition to this 2D convolution (2DConv) and 2D Jacobi stencil computation (2DJacobi) were adopted from PolyBench/ACC benchmark suite [Gra+12]. The 2D convolution is widely used in signal processing and machine learning, and the 2D Jacobi stencil can be used for instance to solve a system of linear equations. 2D Jacobi consists of two kernels, and it is important to note that the second kernel (2DJacobi-2) is strictly a data copy kernel, i.e., it has no computation. Overall, these kernels have different memory access patterns, Compute-to-Communication Ratios (CCRs), which allows us to draw further conclusions on the effects of PREM.

Kernel Characterization As the current Commercial Of The Shelf (COTS) processors are optimized for average-case performance, it is difficult to obtain realistic estimation of worst-case execution time by sequential execution of a scenario. To underline the importance of scheduling of memory accesses and to get closer to realistic Worst-Case Execution Times (WCETs) of our scenarios, we provide evaluation of sensitivity of our kernels to a memory interference in [D3.5].

In this work, we use only the CPU cluster of the TX1 platform for our experiments. In the future work we consider an extension of the PREM to both CPU and GPU clusters, and as was shown in [**Forsberg18**], slowdowns on the GPU side can be even much worse than slowdowns on CPUs.

4.2.2 Use-case scenarios

We evaluate the effectiveness of our toolchain on several application execution scenarios, composed of different combinations of the previously presented kernels. Four small scenarios (each scenario having only sixteen

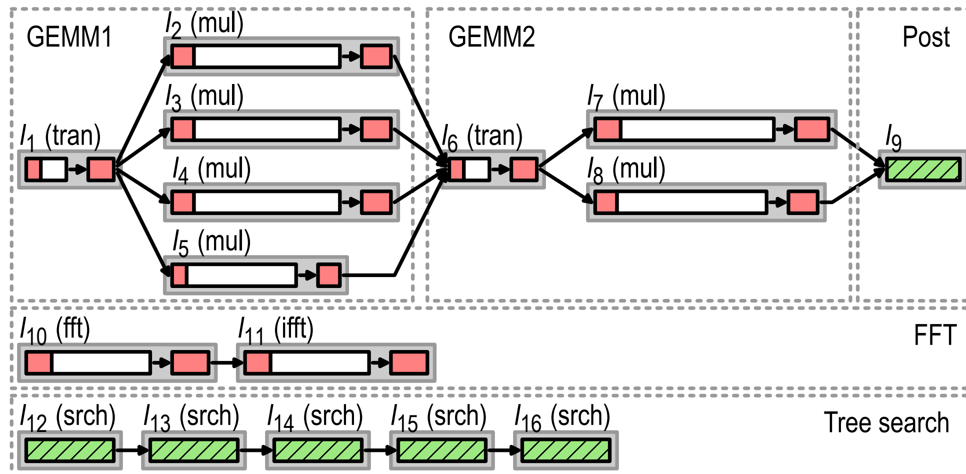


Figure 10: An example of PREMized ADAS scenario. Red rectangles are memory phases, white computations and hatched green are compatible intervals.

intervals), and two large scenarios (each having up to five hundred intervals) are used. We differentiate between small and large by the feasibility to solve the scheduling problem with the ILP approach.

The test scenarios are constructed and compiled to generate the following control flows: For each GEMM, one chain of transpositions and subsequent multiplications that can run in parallel are generated. For the 2D convolution, several parallel intervals are generated and similarly for 2D Jacobi, where two subsequent parallel intervals (2DJacobi-1 and 2DJacobi-2) are generated. For these intervals, the quantity of generated intervals depends of the data size. The FFT and tree traversing are fixed-size and stand-alone intervals in our scenarios. All instances are solved by the heuristic algorithm, but only small instances are solved by the ILP solver, as it is not able to solve large instances in reasonable time.

Four small instance were already described in [D5.1] however, we duplicate the description here for an easy understanding. Table 1 describes compositions of small scenarios. Each application of a scenario is described by two numbers – count and parallelism. We explain the meaning of the numbers on *Scn. 1*, which is the scenario from Figure 10. The first application consists of two subsequent GEMMs ($C = \alpha A \times B + \beta C$) where I_1 and I_6 are transpositions of the matrix B and $I_{2,3,4,5}$ and $I_{7,8}$ are actual multiplications that can run in parallel (GEMM count 2, parallelism 4 and 2). The second application is FFT followed by inverse FFT (FFT count 2, parallelism 1), and the third application is binary search tree algorithm divided into multiple intervals (Search count 5, parallelism 1). The four selected scenarios are:

1. the scenario in Figure 10, explained above,
2. the second scenario is composed of exactly the same applications, the only difference is a division of binary searches into two parallel chains of intervals (I_{12}, I_{13}, I_{14} and I_{15}, I_{16}),
3. the third scenario has only one multiplication divided into seven parallel intervals and
4. the fourth scenario has the same two GEMMs as in Scn. 1, two independent FFTs followed by inverse FFTs and only three graph traversal intervals.

The number of parallel multiplications was automatically generated by the compiler which converted all scenarios into PREM-compliant code. The amount of data processed by FFT was selected such that FFT completely fits into core-local memory. Binary tree search intervals cannot be efficiently converted into predictable intervals, therefore we marked them for transformation into compatible intervals. The compiler also generated scenarios with uncontrolled access to main memory by taking the same dependency graphs and intervals without prefetch and write-back phases. As before, we call these codes *Legacy*.

Scenario	Scn. 1	Scn. 2	Scn. 3	Scn. 4
GEMM count	2	2	1	2
GEMM parallelism	4, 2	4, 2	7	4, 2
FFT count	2	2	2	4
FFT parallelism	1	1	1	2
Search count	5	5	5	3
Search parallelism	1	2	1	1

Table 1: Composition of selected small scenarios

Two large scenarios are used to evaluate the heuristic. The first scenario is the same as Scenario 1, duplicated eight times in sequence. This makes the scenario too large to solve with the ILP solver, but since we can use the ILP solver to find an exact solution for each part, we know the optimal completion time of the sequential composition into a large scenario. This allows us to evaluate the heuristic scheduler performance on large scenarios against a known optimal solution.

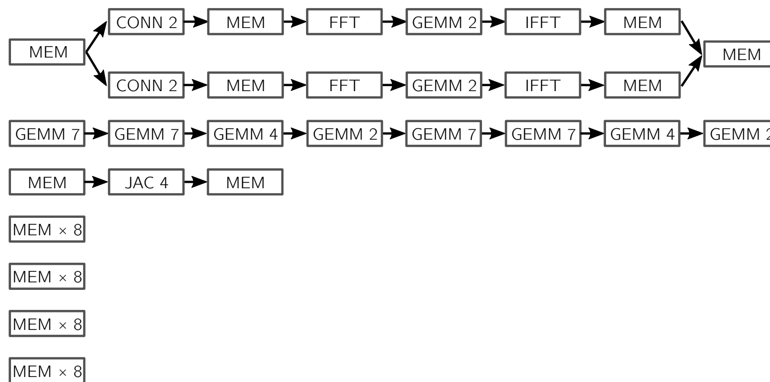


Figure 11: An example of a scenario with 110 intervals.

The second large scenario is inspired by real-world applications, that might be executed together in practice. In detail, we take inspiration from a KCF tracker [**Henriques14**] (*tracker*), convolutional neural networks (*neural*), control tasks (*control*), and image processing pipelines (*image*). An overview of the components of this scenario is shown in Figure 11. The parallelism of the kernels is expressed as a number after the name (e.g. GEMM 4 contains one transposition and four parallel multiplications kernel). On the top of the scenario are two parallel chains inspired by the *tracker*. These chains start and end with a memory intensive task, e.g. opening an image file. Each parallel chain consists of a convolution, a memory intensive interval, an FFT, a matrix multiplication, a Inverse Fast Fourier Transform (iFFT) and another memory intensive interval. In parallel with the *tracker*, an interval chain inspired by the *neural* application is executed, and depicted below the *tracker* chains. This application consists of a single chain of matrix multiplications, representing the evaluation of a neural network. The next chain, below the neural network, represents the *control* application, and consists of Jacobi kernels representing the solving of a system of linear equations. At the bottom of the scenario are 4 chains of memory intensive tasks, such as graph traversing or binary tree searches, which represent the *image* application. Overall, this task combines the components of a possible ADAS-style system, where *image* represents the aquisition of image data, *tracker* and *neural* represent the processing of this data, and lastly *control* represents the actuation on the system. Full Directed Acyclic Graph (DAG) of this scenario is shown in Figure 12.

Execution times of particular PREM phases were obtained by taking the worst-case execution time from 100 executions on a single core. Then we solved the ILP and executed the heuristic with the obtained execution times.

We evaluate our PREM compliant scenarios executed according to the solved schedules on 100 000 runs

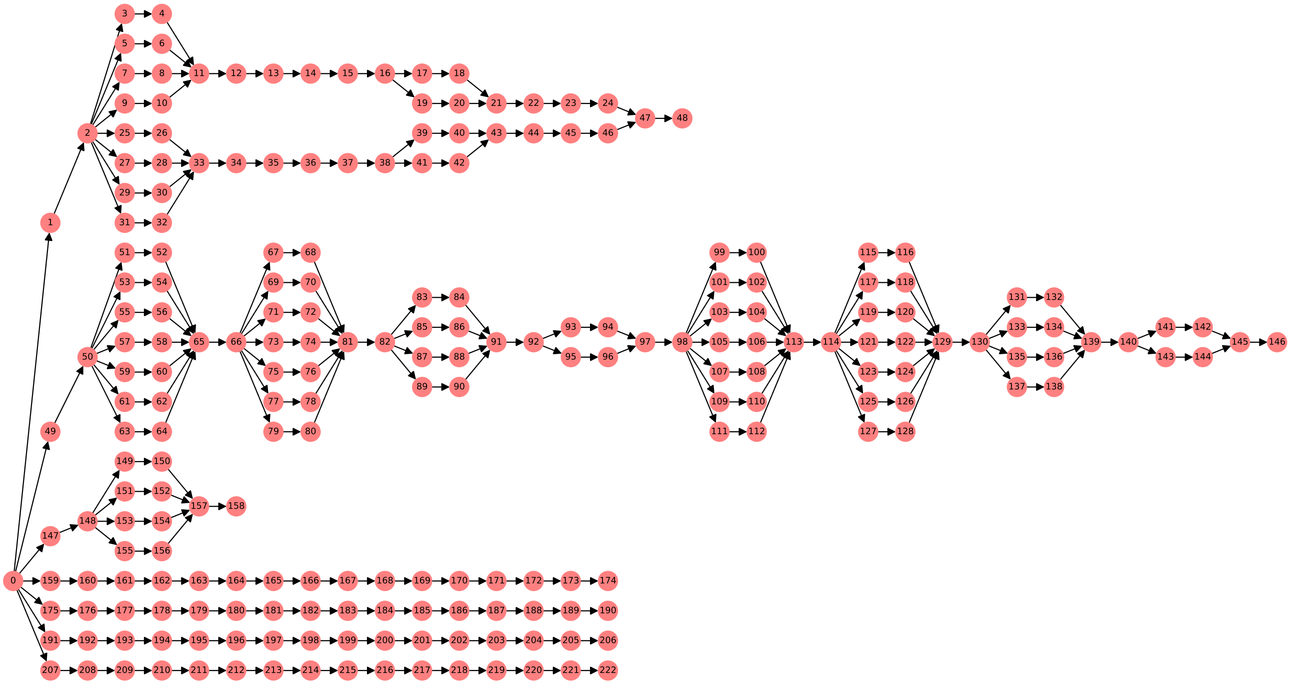


Figure 12: Full DAG of the scenario from Figure 11. It can be seen, which jobs can be executed in parallel. Red dots are the activities representing memory phases of predictable or compatible intervals.

and compare that with an implementation with uncontrolled access to main memory. Both implementations are based on a thread pool in order to minimize overheads for creating new threads. Jobs to be executed by the threads are picked from a queue. In PREM execution, the pool has a thread for each CPU core and the queue is ordered according to the schedule. When a PREM phase finishes earlier than expected, the subsequent phase is executed immediately once all dependencies are satisfied. In *Legacy* executions, the queue is dynamically filled based on the DAG and the jobs are executed by threads whose number equals to the maximum parallelism achievable in the application. The threads are scheduled by the Linux `SCHED_FIFO` scheduler and all have the same priority.

4.2.3 Experimental results

In Table 2 the measured worst-case execution times (WCET) and mean execution times are shown for each of the four small scenarios, both for PREM and *Legacy* executions. Furthermore, the schedule completion times C_{MAX} calculated by the ILP solver and the heuristic are shown for the PREM schedules (*Legacy* schedules are based on best-effort and have no pre-determined schedules). Lastly, the time required to find the optimal and the heuristic schedule for each of the scenarios is provided.

The measured execution times of all 100 000 runs of our small scenarios are presented in logarithmic scale histograms in Figures 13a–13d. The PREM schedules completion times C_{MAX} are shown as a dashed black lines.

There are three main findings in the results of the experiments. First, in every scenario, the variance of completion times under PREM is small (max 6.1%) in comparison to *Legacy* executions (up to 52.4%). We calculate the variance P for PREM as $P = 100 \times (WCET_{PREM}/BCET_{PREM} - 1)$ where $WCET_{PREM}$ and $BCET_{PREM}$ are the measured worst and best case execution times of the PREM compliant execution and analogously $L = 100 \times (WCET_{Legacy}/BCET_{Legacy} - 1)$ for the *Legacy* execution. Higher variances of *Legacy* executions are caused by non-optimal schedules resulting from dynamic scheduling algorithm as well as by competition for

		Small				Large	
Scenario		Scn. 1	Scn. 2	Scn. 3	Scn. 4	Scn. 5	Scn. 8*1
Number of intervals		16	16	16	16	111	128
ILP PREM	C_{MAX} (ms)	7.92	7.92	8.42	8.10	–	–
	WCET (ms)	7.79	7.79	8.40	8.09	–	–
	Mean (ms)	7.63	7.63	8.32	7.87	–	–
Heuristic PREM	C_{MAX} (ms)	8.07	8.40	9.08	8.43	79.20	73.40
	WCET (ms)	8.03	8.15	8.56	8.32	79.10	73.24
	Mean (ms)	7.77	8.00	8.36	8.07	78.19	70.27
Legacy	WCET (ms)	9.75	11.27	11.09	10.79	96.30	88.27
	Mean (ms)	7.77	9.11	8.92	8.93	72.31	60.46
ILP solving time (s)		41	73	7 908	60	–	–
Heuristic solving time (s)		0.9	1.6	0.8	0.6	53	46

Table 2: Scheduled completion time and measured execution times for the scenarios, as well as the time required to find schedules.

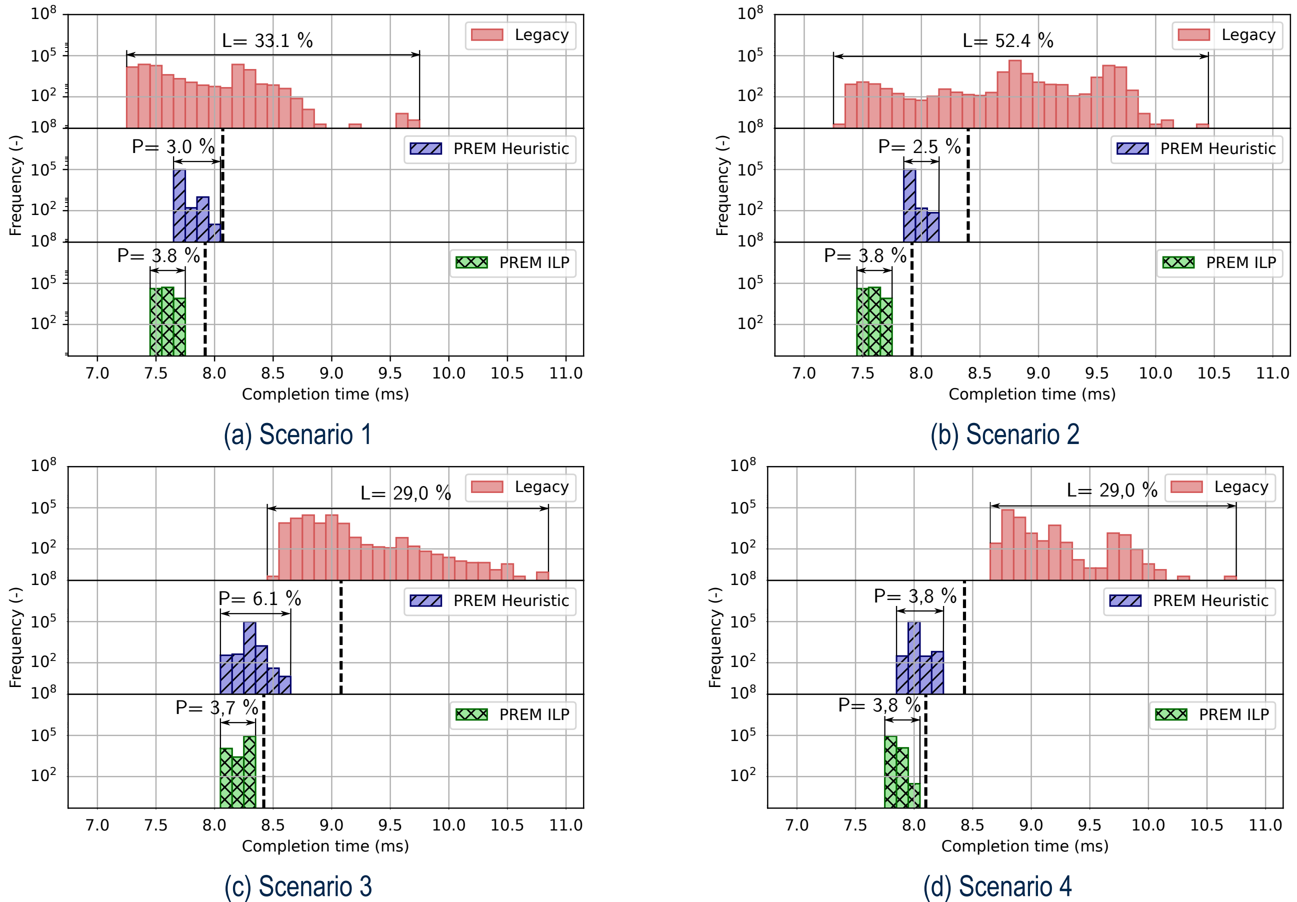
the shared memory. For example, we can see in Figure 13d that the histogram of the *Legacy* executions has three major peaks which correspond to three different schedules and selection of particular schedule depends on actual execution times of preceding intervals. If an interval is delayed, then a different schedule is selected at runtime. We can clearly see the positive impact of PREM in combination with static scheduling on the variance of completion times. The variance could be even smaller if we strictly followed start times of the generated schedule.

Second, the measured WCET of the PREM schedule is always smaller than calculated schedule completion time. Since we allow execution of intervals as soon as they are ready (we do not wait for the corresponding start time when all dependencies are satisfied and requested resources are available), the whole scenario can finish earlier. The fact that all executions finish before estimated WCET shows that our WCET estimations of particular tasks acquired by single core profiling are sufficient and are not affected by the execution of multiple intervals on a multi-core system at the same time.

Third and most important, the measured WCET of PREM executions is always smaller than the WCET of *Legacy* executions (at least by 25.1%, and up to 44.7% for exact solutions, and at least by 21.4%, and up to 38.3% for solutions produced by the heuristic). We calculate the WCET difference as $LP = 100 \times (WCET_{Legacy}/WCET_{PREM} - 1)$. The WCET of *Legacy* executions is strongly affected by dynamic scheduling algorithm which does not understand the structure of the scenario. For example scenarios 1 and 2 are composed of the same intervals, the only difference is that Scenario 2 enables execution of two memory intensive intervals at the same time. Concurrent execution of the intervals (I_{12} and I_{15}) prolongs both of them up to three times, and therefore subsequent tasks are significantly delayed. The delay influences the WCET of the *Legacy* execution which is 11.27 ms instead of 9.75 ms as well as the mean time which is 9.11 ms instead of 7.77 ms while the optimal static schedule for PREM model is the same in both scenarios.

We can also observe that on our small instances, the heuristic perform well, and produces schedules only about 5 to 10 % slower than exact solutions (note in Table 2 that computing the schedule with the heuristic is $45 - 10000\times$ faster than the ILP approach). Also, average execution times as well as worst-case execution times are always better than *Legacy* executions.

For larger scenarios, where the exact solver was not applicable, the heuristic comes into play most usefully. The PREM benefit on large scenarios depends on many factors. The sensitivity to memory interference plays an important role, which was already discussed in Section 4.2.1, as does the structure of the scenario and the efficiency of the heuristic. We show results of our heuristic on two different large scale scenarios, of which we show histograms in Figure 14. However, it can not be easily shown how far the generated solution is from the optimal solution. For this reason, we use Scenario 8*1, with the histogram depicted in Figure 14b, consisting



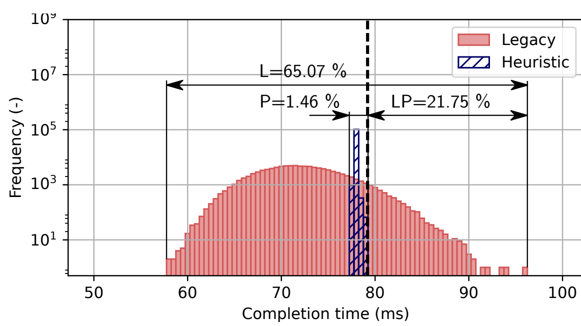
(a) Scenario 1
(b) Scenario 2
(c) Scenario 3
(d) Scenario 4

Figure 13: Histograms comparing completion times of small scenarios with and without PREM applied

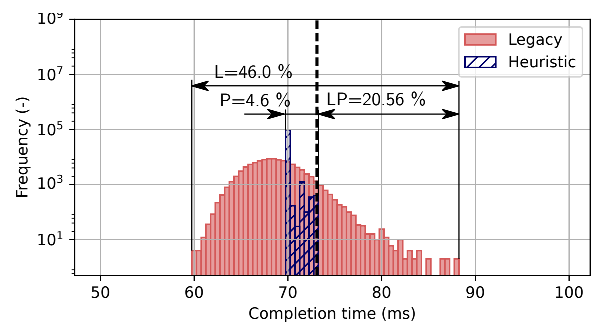
of eight times sequentially duplicated Scenario 1, for which we already know the optimal solution. Therefore, for this particular scenario, we can compare generated schedule with the optimal one. Optimal C_{MAX} for the scenario equals $8 \times 7.9 = 63.2$ while the heuristic generated schedule with $C_{MAX} = 73.4$. This represents 15.5 % precision.

The histogram of execution times of Scenario 5 is shown in Figure 14a. While the average execution time of PREM is higher than that of *Legacy*, the measured WCET of *Legacy* is much larger than in the PREM execution ($LP = 21.75$ %). The reason for the higher average execution is that this scenario includes several kernels that were already shown to be less performant under PREM (see Section 4.2.1), however, even in light of this, PREM provides tighter WCET bounds in this scenario. This is a good outcome, as the WCET is the limiting factor in how many tasks that can be successfully scheduled in a system. Also, we can see that the execution time variance is much lower in the PREM execution (1.5 % vs 65.1 %). From this we see that PREM successfully reduces the execution time jitter, greatly improving the predictability of the system.

To sum up the evaluation of the off-line scheduling section, we show that for small as well as large-scale instances, the jitter of PREM executions is very small compared to *Legacy* executions, and measured WCETs of PREM execution are always smaller than calculated completion times. Therefore, no schedulability analysis for off-line scheduling is required.



(a) Scenario 5



(b) Scenario 8*1

Figure 14: Histograms comparing completion times of large-scale scenarios with and without PREM applied

5 Conclusion

This deliverable documented an integrated schedulability analysis for the memory-aware real-time scheduling setting addressed by the HERCULES framework. In the first part of the document, we outlined the system-level mechanisms that have been implemented to enable a memory-aware co-scheduling of tasks on the available computing engines. A hypervisor-level Memguard support has been implemented for memory bandwidth regulation of general-purpose tasks. Then, a memory spinlock (called Memex) has been detailed to enable a privileged access to main memory without experiencing memory-intensive interference from other cores. This latter mechanism is particularly useful for tasks that can be PREM-ized with the compiler provided in deliverable D3.3.

In the second part of the deliverable, we then detailed the schedulability analysis for tasks partitioned to cores, whose memory accesses are co-scheduled using the proposed approaches. Recent results in the literature have been summarized to provide the basis of the integrated analysis. Individual analyses for Memguarded systems and simple PREM-ized settings have been recalled, before providing an integrated analysis for a hybrid setting that merges both approaches, as assumed in the HERCULES framework.

The last part of the document described how the FPGA engine could be conveniently used to decrease the system-level overhead of the proposed memory-aware arbitration mechanisms. Finally, an alternative method based on a statically computed off-line schedule is proposed, and validated on practical settings.

References

[AP14] A. Alhammad and R. Pellizzoni. "Schedulability analysis of global memory-predictable scheduling". In: *2014 International Conference on Embedded Software (EMSOFT)*. Oct. 2014, pp. 1–10. DOI: 10.1145/2656045.2656070.

[Bar18] Maxim Baryshnikov. "FPGA-based support for predictable execution model in multi-core CPU". https://dspace.cvut.cz/handle/10467/77607?locale-attribute=en. MA thesis. Czech Technical Univeristy in Prague, 2018.

[BBY13] G. C. Buttazzo, M. Bertogna, and G. Yao. "Limited Preemptive Scheduling for Real-Time Systems. A Survey". In: *IEEE Transactions on Industrial Informatics* 9.1 (Feb. 2013), pp. 3–15. ISSN: 1551-3203. DOI: 10.1109/TII.2012.2188805.

[BLV09] Reinder J. Bril, Johan J. Lukkien, and Wim F. J. Verhaegh. "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption". In: *Real-Time Systems* 42.1 (Aug. 2009), pp. 63–119. ISSN: 1573-1383. DOI: 10.1007/s11241-009-9071-z. URL: https://doi.org/10.1007/s11241-009-9071-z.

[D3.5] Hercules consortium. *D3.5: Optimized runtime for parallel heterogeneous platforms*. Deliverable of the HERCULES project. June 2018.

[D4.5] Hercules consortium. *D4.5: Multi-OS Integration and Virtualization*. Deliverable of the HERCULES project. June 2018.

[D5.1] Hercules consortium. *D5.1: Power-Aware Scheduling Algorithm for Host*. Deliverable of the HERCULES project. June 2018.

[Gra+12] S. Grauer-Gray et al. "Auto-tuning a high-level language targeted to GPU codes". In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pp. 1–10. DOI: 10.1109/InPar.2012.6339595.

[HŠ17] Zdeněk Hanzálek and Přemysl Šůcha. "Time symmetry of resource constrained project scheduling with general temporal constraints and take-give resources". In: *Annals of Operations Research* 248.1 (Jan. 2017), pp. 209–237. ISSN: 1572-9338. DOI: 10.1007/s10479-016-2184-6. URL: https://doi.org/10.1007/s10479-016-2184-6.

[Mel+15] Alessandra Melani et al. "Memory-processor Co-scheduling in Fixed Priority Systems". In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. RTNS '15. Lille, France: ACM, 2015, pp. 87–96. ISBN: 978-1-4503-3591-1. DOI: 10.1145/2834848.2834854. URL: http://doi.acm.org/10.1145/2834848.2834854.

[Mel+17] A. Melani et al. "Exact Response Time Analysis for Fixed Priority Memory-Processor Co-Scheduling". In: *IEEE Transactions on Computers* 66.4 (Apr. 2017), pp. 631–646. ISSN: 0018-9340. DOI: 10.1109/TC.2016.2614819.

[Pel+11] R. Pellizzoni et al. "A Predictable Execution Model for COTS-Based Embedded Systems". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2011, pp. 269–279. DOI: 10.1109/RTAS.2011.33.

[Pel+12] Rodolfo Pellizzoni et al. "Predictable Execution Model: Concept and Implementation". In: (May 2012).

[Sha+16] L. Sha et al. "Real-Time Computing on Multicore Processors". In: *Computer* 49.9 (Sept. 2016), pp. 69–77. ISSN: 0018-9162. DOI: 10.1109/MC.2016.271.

[Yao+16a] G. Yao et al. “Global Real-Time Memory-Centric Scheduling for Multicore Systems”. In: *IEEE Transactions on Computers* 65.9 (Sept. 2016), pp. 2739–2751. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2500572.

[Yao+16b] G. Yao et al. “Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems”. In: *IEEE Transactions on Computers* 65.2 (Feb. 2016), pp. 601–614. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2425874.

[Yun+12] H. Yun et al. “Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality”. In: *2012 24th Euromicro Conference on Real-Time Systems*. July 2012, pp. 299–308. DOI: 10.1109/ECRTS.2012.32.

[Yun+13] H. Yun et al. “MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms”. In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2013, pp. 55–64. DOI: 10.1109/RTAS.2013.6531079.