Optimal Performance Tuning in Real-Time Systems using Multi-Objective Constrained Optimization

Stefano Di Alesio

Certus Centre for Software Verification & Validation, Simula Research Laboratory, Norway
stefano@simula.no

Abstract. Real-Time Embedded Systems (RTES) in safety-critical applications have to meet strict performance requirements to be deemed safe for operation. The satisfaction of these requirements at runtime often depends on configuration parameters that regulate how software tasks interact with hardware sensors and actuators. Tuning performance-related parameters is usually a manual, time-consuming, and error-prone process. This is because these parameters and their values define a large space of system configurations, and evaluating how each configuration affects the performance often requires executing the whole system. In this paper, we express RTES performance tuning as a multi-objective Constrained Optimization Problem (COP) over the configuration space that captures the dependencies between configuration parameters and performance requirements. In this way, the COP solutions characterize configurations predicted to maximize the satisfaction of performance requirements, and can in turn be used as guidelines for optimal performance tuning. We develop the COP as an OPL model for IBM ILOG CP OPTIMIZER, and validate our approach on a safety-critical I/O drivers system from the maritime and energy domain. The validation shows that our approach identifies within half an hour configurations characterized by tasks delay times that minimize deadline misses, response time, and CPU usage.

1 Introduction: Performance Tuning in Safety-Critical Systems

Failures in safety-critical systems, such as those in the energy, transport, and healthcare domains, could result in catastrophic consequences [17]. Therefore, the safety-related software components of these systems are usually subject to strict performance requirements involving real time and resources utilization constraints [26]. In particular, three performance requirements that are commonplace in safety-critical systems concern *task deadlines*, *response time*, and *CPU usage* [24]. Specifically, task deadlines state that the system tasks should always terminate before a given completion time, entailing that even a single deadline miss severely compromises the system operational safety. Response time requirements specify that, in order for the outputs to be valid, the system should react to external inputs within a specified time. Finally, CPU usage constraints state that the system should always keep a certain percentage of free CPU time, to avoid that high computational load prevents the system from timely responding to safety-critical alarms.

However, safety-critical systems are progressively relying on Real-Time Embedded Systems (RTES), where software applications interact with the environment through sensors and actuators [22]. In complex RTES, the software components communicate with a large number of different devices. In particular, RTES have to ensure a smooth data transfer between hardware devices and software components. This is especially true in safety-critical systems, where external data should always be processed in brief time to guarantee a prompt reaction to critical events [34]. Therefore, the timing of RTES tasks can be configured to correctly operate with the specific devices connected. Nonetheless, tuning these timing properties without violating performance requirements is complicated by two main factors [16]. First, the task parameters related to temporal properties, such as delay times, offsets, and periods, range in a large domain of values. Second, the impact specific parameter values have over the system performance is hard to evaluate without executing the whole system. This is because RTES typically run on a preemptive Real-Time Operating System (RTOS) which may preempt a task execution in order to run an incoming higher priority task. Therefore, a minimal variation in a single task timing may trigger unpredictable interactions between other tasks [10].

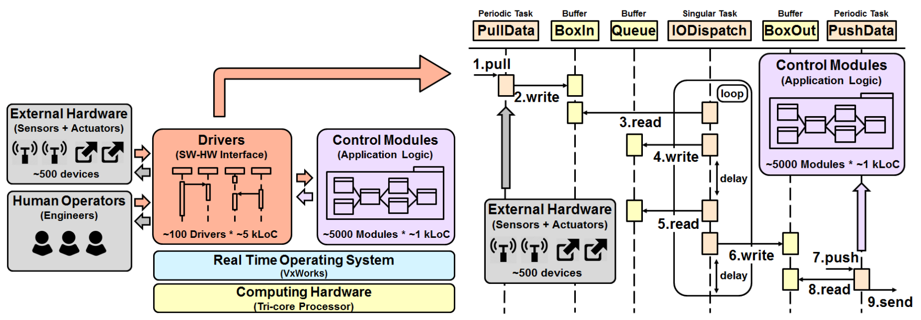
As a consequence, it is often practice in industry to tune parameters related to RTES performance manually, based on the engineers expertise and knowledge of the system. This renders the process of tuning these performance-related parameters significantly time-consuming and error-prone [41]. Traditionally, systematic approaches for analyzing the RTES performance properties rely on Scheduling Theory [36], which is often based on unrealistic assumptions on the target system [3]. On the other hand, Model Checking [1] has been successfully proposed as an alternative for performance analysis and tuning [9], even though its scalability has to be further investigated due to the well-known state explosion problem [8]. Approaches based on metaheuristic search have also been proposed for identifying configuration parameters likely to satisfy performance requirements on CPU usage [28]. However, previous work in the field of software stress testing [13] suggests that complete search strategies, such as those based on Constraint Programming (CP), can potentially find solutions closer to the global optimum than meta-heuristics such as Genetic Algorithms (GA), and hence are worth being considered also in the context of performance tuning.

In this paper, we propose a methodology, based on Constrained Optimization, to help engineers tune performance-related parameters of RTES. The key idea behind our work is to identify scenarios where tasks finish their execution as far as possible from their deadlines, and exhibit low response time and CPU usage. Such scenarios are determined by the way tasks are scheduled to execute at runtime by the RTOS. The task schedules depend in turn on the value of system timing parameters, on constraints derived from software design, and on the execution platform. Therefore, we propose a strategy to find combinations of timing properties that maximize the satisfaction of performance requirements on deadline misses, response time, and CPU usage. We characterize each of these combinations by a set of task *delay times*, and we refer to each set of delay times as a *configuration*.

2 Motivating Case Study: The Fire and Gas Monitoring System

The motivation behind our work originates from a case study in the maritime and energy domain concerning a Fire and Gas Monitoring System (FGMS). The system monitors potential gas leaks in off-shore oil extraction platforms, displaying to human operators data coming from smoke, heat, and gas flow sensors. In case a fire is detected, the FGMS triggers audio/visual alarms, activates the fire sprinklers, shuts down ongoing processes, and isolates electrical equipment. The software architecture of the system consists of drivers and control modules, as shown in Figure 1a. Drivers support I/O communication between the software components and the operating environment, which consists of hardware sensors and actuators. Control modules implement the application logic of the FGMS, processing operational commands coming from the environment and accordingly deciding the actions to perform. The FGMS software components are executed by the RTOS VxWorks[1], which is configured with a fixed-priority preemptive scheduling policy on a tri-core computing platform.

We point out three main context factors that influence our formulation of the performance tuning problem in the FGMS. (1) In this paper, we do not consider FGMS-level performance requirements, which require considering interactions between drivers, control modules, and external hardware. On the other hand, we limit our scope to driver-level requirements, for which it is only necessary to consider the drivers subsystem. To avoid confusion, in the rest of this paper we will refer to the FGMS drivers as the *system* under investigation. (2) Different instances of a given driver are independent, i.e., they do not communicate with one another and do not share memory. For this reason, in this paper we focus on a single driver instance and do not consider interactions between them. (3) The FGMS performance profiling logs indicate that task deadlines, response time, and CPU usage of the drivers are not significantly affected by memory allocation activities such as garbage collection and data transfer operations on storage peripherals. For this reason, we do not consider the impact of memory usage on the drivers performance.



(a) Software architecture of the FMS

(b) Typical drivers data transfer scenario

Fig. 1: Description of the Fire and Gas Monitoring System

Drivers in the FGMS share the same design pattern, consisting of two periodic tasks, (*PullData* and *PushData*), and one singular task (*IODispatch*), which is executed only once during the drivers execution. The tasks communicate through three fixed capacity buffers with mutually exclusive access, namely *BoxIn*, *Queue*, and *BoxOut*. Figure 1b shows how the three tasks collaborate in the typical operating scenario, that is a unidirectional data transfer between hardware sensors and control modules.

[1] `http://www.windriver.com/products/vxworks`

(1) *PullData* periodically receives data from sensors or human operators through the *pull* signal, formats the data in an appropriate command form, and (2) writes it in *BoxIn*. (3) *IODispatch* reads the buffer, extracts the commands from the data, and (4) stores them in the priority *Queue*. After a given delay time, (5) *IODispatch* reads the highest priority command and (6) writes it to *BoxOut*. When the periodic *push* signal (7) activates *PushData*, the task (8) reads the commands from *BoxOut* and finally (9) sends them to the control modules for processing. Note that *IODispatch* is executed when the drivers are initialized, and encloses within an infinite loop four sequential read/write *activities*. In particular, the two activities writing in *Queue* and *BoxOut* are followed by delay times, implemented as *sleep* calls in the drivers source code. In particular, the delay times may vary during the *IODispatch* loop iterations, allowing the drivers to send data to the control modules at a variable rate. This design is meant to ensure that the FGMS data flow is slow enough to avoid overloading the computational resources, but fast enough to ensure prompt reaction to critical events.

The data transfer functionality is subject to strict performance requirements. Specifically, in each driver instance, (1) no task should miss its deadline, (2) the response time should be less than one second, and (3) the average CPU usage should be below 20%. The main variables determining whether or not these requirements will be satisfied at runtime are the delay times after the write activities of *IODispatch*. Indeed, if the delay times are too short, *IODispatch* is running continuously and keeps the CPU busy, eventually exceeding the 20% usage threshold. On the other hand, if the delay times are too large, *PullData* may fill up *BoxIn*, and be blocked waiting for *IODispatch* to empty the buffer. As a result, *PullData* is not able to terminate before the next *pull* signal arrives, missing its deadline. In general, it is hard to predict whether a set of delays in *IODispatch* will break deadlines in other tasks, or will make the driver exhibit high response time or CPU usage at runtime. This is because the delay determines the arrival time of the activities in *IODispatch*, which in turn can preempt or be preempted by other tasks. Note that, however, the delay times of the *IODispatch* iterations are tunable parameters that engineers can set when configuring the drivers.

3 Related Work

The increasing complexity in RTES software and hardware architecture renders analyzing and estimating performance properties in RTES increasingly challenging [26], especially in safety-critical domains, where performance issues can impact the system behavior more than incorrect functionality [39]. In RTES, performance properties have been traditionally analyzed through verification approaches, such as Scheduling Theory [36] and Model Checking [1]. Theorems from Scheduling Theory are limited to providing sufficient or necessary conditions for a set of tasks to be schedulable, and are often based on unrealistic assumptions on the target system [3]. On the other hand, Model Checking approaches analyze time-related properties, such as task deadlines and resource usage, by proving reachability properties in state machines [9]. However, Model Checking requires complex formal modeling of the system, which is not always available for large systems and often leads to the state explosion problem [8]. Our experience suggests that, in several industrial contexts, RTES are developed by relatively small-sized teams consisting of developers with several years of expertise in their domain. This development strategy increases productivity by reducing the communication overhead [15], but can potentially come at the cost of overlooking the need of systematic performance analysis. This usually happens in systems with long lifespan, whose core functionalities undergo only minor updates over the years and are hence deemed stable. In such systems, performance tuning is mostly addressed by human expertise, which in turn relies on profiling and benchmarking tools that dynamically analyze performance properties [21]. Such tools, however, can only provide a rough performance assessment limited to a small number of system executions, which have to be manually investigated [41].

The use of search-based approaches to find optimal configuration parameters originates from the domain of control systems [38], where Genetic Algorithms (GA) have been applied to tune the performance of Proportional-Integrative-Derivative (PID) controllers [19]. In the context of Real-Time Systems, GA have been used to generate scenarios characterized by reproducible environmental conditions that push the system to break task deadlines [5]. These approaches have inspired the use of metaheuristic search to derive configuration parameters that are predicted to minimize the CPU usage [28]. In particular, Non-dominated Sorting GA (NSGA) have been used to identify task offsets that yield an optimal trade-off between CPU usage and requirements specifying groups of tasks that have to be executed within short time [29]. Even though these approaches have only been applied in the context of Static Cyclic Scheduling (SCS), which is non-preemptive, they represent the closest related work to that presented in this paper.

Previous work [13] in the area of stress testing suggests that, when compared to GA, Constraint Programming (CP) can find task schedules closer to the global optimum, and is hence worth investigating also in the context of performance tuning. For schedulability analysis, CP approaches [4] have been used since long time, especially in the domain of job-shop scheduling [25]. Among those, several approaches target task real-time constraints such as task deadlines [18], or timeliness [27]. Preemptive scheduling problems have also been approached with pure CP [6], and with hybrid approaches combining CP with GA [42]. The most recent implementations have successfully used both CP and Mixed Integer Programming (MIP) to solve priority-driven scheduling problems, albeit not addressing task preemption [23]. However, we are unaware of CP approaches targeted to the generation of software configurations predicted to satisfy performance requirements, and in particular of approaches addressing all the complexities of RTES such as multi-core architectures, task dependencies and triggering relationships, and priority-based preemptive scheduling policies.

4 Performance Tuning with Constrained Optimization

The approach presented in this paper extends earlier work [12, 30] for deriving test cases exercising CPU usage and task deadlines requirements of multi-core RTES. Specifically, the approach has been adapted to derive configurations characterized by task delay times that maximize the satisfaction of requirements on their deadlines, response time, and CPU usage. In particular, we cast the search for these delay times as a multi-objective Constraint Optimization Problem (COP). The COP models a preemptive priority-based task scheduler with fixed priorities, task triggering, and dependencies on shared resources. The COP is derived from earlier work on generating stress test cases for RTES [14], and the key idea behind its formulation relies on four main points. (1) We model the system design, which is static and known prior to the analysis, as a set of constants. The system design consists of the tasks of the real-time application, their dependencies, offsets, periods, durations, deadlines, and priorities. (2) We model the system properties that depend on runtime behavior, and those that are configurable parameters, as a set of variables. The main real-time property in the first category is the specific runtime schedule of the tasks. Configurable properties, which are the output variables of the model, are instead the delay times between task activities. (3) We model the RTOS scheduler as a set of constraints among such constants and variables. Indeed, the RTOS scheduler periodically checks for triggering signals of tasks and determines whether tasks are ready to be executed or need to be preempted. (4) We model the performance requirements the system must satisfy (i.e., task deadlines, response time, or CPU usage) as objective functions to be minimized. By design, each solution of our COP is a sequence of task delay times, which in turn characterizes a configuration.

Our analysis is subject to two assumptions. (1) The RTOS scheduler checks the running tasks for potential preemptions at regular and fixed intervals of time, called *time quanta*. Therefore, each time value in our problem is expressed as a multiple of a time quantum. (2) The interval of time in which the scheduler switches context between tasks is negligible compared to a time quantum. We found these two assumptions to be commonplace in several RTES, as the scheduling rate of operating systems varies in the range of few milliseconds, while the time needed for context switching is usually in the order of nanoseconds [32]. These assumptions allow us to consider time as discrete in our analysis, and model the COP as an Integer Program (IP) over finite domains.

We implemented the COP in the Optimization Programming Language (OPL) [37], and solved it with IBM ILOG CPLEX CP OPTIMIZER[2]. This choice was motivated by practical reasons, such as its extensive documentation, strong supporting community, and acknowledged efficiency to solve optimization problems. Note that we could not express a preemptive priority-driven scheduling problem in an effective way that exploited the solver capabilities of working with task intervals [7], and hence we implemented our COP as a traditional IP. In the following, we describe our constraint model (Section 4.1), and how to use it to model infinite loops of activities separated by a delay time (Section 4.2).

4.1 Description of the Constrained Optimization Problem

Constants. As explained before, we consider time as discretized in time quanta. Therefore, we define the *observation interval* T as an integer interval of length tq, i.e., $T \overset{\text{def}}{=} [0, tq-1]$, representing the time interval during which we observe the system behavior. Each time value $t \in T$ is a time quantum. We define c as the *number of cores* in the execution platform, representing the maximum number of tasks that can be executed in parallel, J as the *set of tasks* of the system, and R as the *set of resources* shared by such tasks. Each resource $r \in R$ is typically implemented as a buffer, and serves as a mechanism to store data for synchronous and asynchronous communication between tasks. Each task $j \in J$ has a set of static properties, whose values are part of system design and known prior to the execution of the system. These static properties are defined in Real-Time Scheduling Theory [33], and comprehend the task *priority* pr_j, *period*

[2] `http://www.ibm.com/software/commerce/optimization/cplex-cp-optimizer/`

pe_j, *offset* of_j, *deadline* dl_j, and *number of task executions* te_j. In particular, we refer to the k^{th} execution of task j as the couple $(j, k-1)$. In this way, the first execution of j is the couple $(j, 0)$. The offset and period determine the number of task executions so that $te_j = \left\lfloor \frac{tq - of_j}{pe_j} \right\rfloor$. For simplicity, we define the interval K_j of executions of task j as $K_j \overset{\text{def}}{=} [0, te_j - 1]$, so that, in the context of a given j, $k \in K_j$. We also consider the *duration* dr_j of tasks as a constant equal to the task Worst Case Execution Time (WCET), which can be estimated through different techniques both statically, using the system design, and dynamically, by measuring execution times [40]. In our context, the WCET is estimated by selecting the worst-case time across several executions of the system. Note that considering the duration of each task as its WCET is a common practice when analyzing task real-time properties [16]. We refer to the d^{th} time quantum of the task execution (j, k) as the triple $(j, k, d-1)$. For simplicity, we also define the interval D_j of duration time quanta of a task as $D_j \overset{\text{def}}{=} [0, dr_j - 1]$, so that, in the context of a given j, $d \in D_j$. Finally, we also define as constants the tasks *triggering relation* tg, and *read (write) dependency relation* rd (wr). The former is an irreflexive and antisymmetric binary relation over $J \times J$, where tg_{j_1,j_2} holds if the event triggering j_2 occurs when j_1 finishes its execution, plus a possible delay. The latter are binary relations over $J \times R$, where $rd_{j,r}$ $(wr_{j,r})$ holds if j reads data from (writes data to) r during its execution. Note that tasks in a dependency relation cannot be executed in parallel nor can preempt each other, but one can execute only after the other has released the lock on the resource.

Variables. Tasks in J also have a set of dynamic properties, whose values depend on the runtime behavior of the system, and hence are not known prior to the analysis. Indeed, the values for these variables are calculated during the search, and represent the output data of the COP. In the context of constraint solving, variables whose domain values define the search space are said to be *independent* or *decision variables* [20]. Indeed, the goal of a constraint solver is to assign values for the independent variables that satisfy all the constraints, optimizing an objective function when specified. In our model, the independent variables characterize configurations in terms of the delay time that trigger the task executions. The independent variables of our model, marked with a single dot $(\cdot)$, are the time quanta $\dot{ac}_{j,k,d}$ where the system tasks are active and executing, the arrival times $\dot{at}_{j,k}$ of triggered task executions, and their delay times $\dot{dy}_{j,k}$. All these variables have domain in T. In particular, we refer to the set of all $\dot{ac}$ variables as the *schedule* produced by the arrival times of tasks in J. Note that the arrival times of periodic tasks are constant, and determined by the task period and offset: $\dot{at}_{j,k} = of_j + k \cdot pe_j$. In addition to these independent variables, we also define *dependent* variables, whose value is defined by a mathematical expression of independent variables and constant values. Dependent variables, marked with a double dot $(\cdot\cdot)$, simplify our notation by allowing us to easily formulate constraints and objective functions. For example, we define as dependent variable the *start* and *end time* of tasks $\ddot{st}_{j,k}$ and $\ddot{en}_{j,k}$, i.e., the first and the last time quantum in which (j, k) is executing: $\ddot{st}_{j,k} \overset{\text{def}}{=} \dot{ac}_{j,k,0}$ and $\ddot{en}_{j,k} \overset{\text{def}}{=} \dot{ac}_{j,k,dr_j-1} + 1$. In particular, the end times of tasks allows to define the *deadline miss* $\ddot{dm}_{j,k}$ of a task execution as the amount of time by which (j, k) missed its deadline: $\ddot{dm}_{j,k} \overset{\text{def}}{=} \ddot{en}_{j,k} - (\dot{at}_{j,k} + dl_j)$. We also define the *system load* $\ddot{ld}_t$ as the number of tasks active at time t: $\ddot{ld}_t \overset{\text{def}}{=} \sum_{j,k,d}(\dot{ac}_{j,k,d} = t)$. Note that $(\dot{ac}_{j,k,d} = t)$ is a boolean variable that is evaluated to 1 when true, and to 0 when false. Furthermore, the dependent variables include the *preempted time quanta* $\ddot{pm}_{j,k,d}$ of task executions, defined as the number of time quanta for which (j, k) is preempted for the d^{th} time: $\ddot{pm}_{j,k,d} \overset{\text{def}}{=} \dot{ac}_{j,k,d} - \dot{ac}_{j,k,d-1} - 1$, and the *waiting time* $\ddot{wt}_{j,k}$ of task executions, defined as the amount of time for which (j, k) has to wait after its arrival time before starting its execution: $\ddot{wt}_{j,k} \overset{\text{def}}{=} \ddot{st}_{j,k} - \dot{at}_{j,k}$. The preempted time quanta and the waiting time allow us to easily formulate constraints specifying that tasks should only be preempted by higher priority tasks, and should postpone their starting time only when they are locked on a shared resource, or waiting for data to be written, or because there is no processing core available. Finally, we define the *resource status* indicator $\ddot{rs}_{r,t}$ as a binary variable indicating whether the resource r is full or empty at time t. For any pair of executions of two tasks j_1 and j_2 which respectively read and write r, $\ddot{rs}_{r,t}$ has value 1, i.e., the resource is full and ready to be accessed for read operations, between the end of j_2 and the end of j_1, and has value 0, i.e., the resource is empty and cannot be accessed for read operations, otherwise: $\forall j_1, j_2 \in J, k \in K_{j_1} \cap K_{j_2}, r \in R, t \in T \cdot rd_{j_1,r} \wedge wr_{j_2,r}$

$$\ddot{rs}_{r,t} \overset{\text{def}}{=} \begin{cases} 1 & \text{if } t \in [\ddot{en}_{j_2,k}, \ddot{en}_{j_1,k}] \\ 0 & \text{otherwise} \end{cases}$$

The resource status indicator allows us to easily formulate constraints specifying that tasks can only write to empty buffers, and read from full buffers.

Constraints. We define five sets of constraints which model task runtime interactions, i.e., locks and preemptions, and the way in which the RTOS scheduler executes these tasks based on their triggering and dependency relations. In this paper, we only report shortened expressions of constraints, labeled with the letter γ, as their rigorous mathematical formulation is part of previous work [14]. *Well-formedness constraints* specify relations among variables that directly follow from their definition in the schedulability theory. For example, well-formedness constraints state that each task execution starts after its arrival time, and ends after the task duration dr ($\gamma_1 : \dot{at}_{j,k} \leq \ddot{st}_{j,k} \leq \ddot{en}_{j,k} - dr_j$). Furthermore, note that resources may be shared between more than two tasks. This entails that more than one task execution can be locked on a given resource at any time. In RTES, task queues regulate the access of a resource by multiple locked tasks. In our COP, these task queues are modeled through a well-formedness constraint stating that if task j_2 is ready to be executed at the same time as a lower-priority task j_1, j_2 starts first: ($\gamma_2 : \dot{at}_{j_1,k_1} = \dot{at}_{j_2,k_2} \longleftrightarrow \ddot{st}_{j_2,k_2} \leq \ddot{st}_{j_1,k_1}$). *Temporal ordering constraints* specify the relative ordering of tasks based on their dependency and triggering relations. In particular, these constraints state that the a task j_2 triggered by j_1 arrives after the delay of j_1, counted from when j_1 ends ($\gamma_3 : \ddot{en}_{j_1,k} + \dot{dy}_{j_1,k} = \dot{at}_{j_2,k}$). Furthermore, temporal ordering constraints state that the executions of two dependent tasks j_1 and j_2 cannot overlap, i.e., that one can only start after the other has ended ($\gamma_4 : \ddot{en}_{j_1,k_1} \leq \ddot{st}_{j_2,k_2} \vee \ddot{en}_{j_2,k_2} \leq \ddot{st}_{j_1,k_1}$). Finally, these constraints state that (1) a task cannot write to a full buffer, i.e., that the start time of a task j writing on a resource r has to occur when r is empty ($\gamma_5 : \ddot{rs}_{r,\ddot{st}_{j,k}} = 0$), and that (2) a task cannot read from an empty buffer, i.e., that the start time of a task j reading from a resource r has to occur when r is full ($\gamma_6 : \ddot{rs}_{r,\ddot{st}_{j,k}} = 1$). *Multi-core constraints* capture the concurrent nature of the computing platform, stating that no more than c tasks can be active at any time ($\gamma_7 : \ddot{ld}_t \leq c$). *Preemption constraints* capture the priority-driven preemptive scheduling of the RTOS, stating that each task should be preempted when a higher priority task is ready to be executed and no cores are available. Finally, *scheduling efficiency constraints* ensure that tasks are not preempted unnecessarily and are executed as soon as possible.

Objective Functions. We formalize three objective functions representing task deadlines, response time, and CPU usage. The functions are minimized in a multi-objective optimization problem, in a way that solutions of the COP characterize scenarios approaching optimal tradeoffs between the objective values. Note that, even though the performance requirements specify a maximum threshold on the value of task deadlines, response time, and CPU usage, the value of these properties is not bound by any constraint in the COP. Therefore, the search process might initially find solutions that satisfy the constraints, but whose objective value is greater than the threshold expressed by the requirements. However, our COP is based on estimates of the tasks WCET, which might be over-pessimistic. For this reason, the delay times characterized by these solutions might not violate the performance requirements at runtime, and hence are worth looking at during configuration. Nevertheless, the COP minimizes the objective functions representing the performance requirements, because the lower the objective values, the higher the confidence that the system achieves a satisfactory performance. We define the *CPU usage function* F_{CU} that models the system CPU usage: $F_{CU} \overset{\text{def}}{=} \sum_{t \in T} (\ddot{ld}_t > 0) \big/ tq$. F_{CU} measures the average CPU usage of the system over T as the percentage of T where at least one core is busy. We define the *deadline misses function* F_{DM} that models the requirement on task deadlines: $F_{DM} \overset{\text{def}}{=} \sum_{j,k} 2^{\ddot{dm}_{j,k}}$. To ensure that tasks completing in short time do not overshadow deadline misses, F_{DM} assigns to $\ddot{dm}$ an exponential contribution towards the sum [13]. Recall that $\ddot{dm}_{j,k}$ is positive if the task execution (j, k) misses its deadline, and negative otherwise. Finally, we also define the *response time function* F_{RT} that models the system response time. In traditional scheduling, the response time measures the maximum length in time quanta of the task schedule restricted to a single execution. This means that the response time is the maximum time between the k^{th} arrival time of a task, and the k^{th} end time of a possibly different task. The response time is also traditionally known as *makespan* [31].

$$F_{RT} \overset{\text{def}}{=} \max_{j_1, j_2 \in J,\ k \in K_{j_2} \cup K_{j_2}} \left(\ddot{en}(j_1, k) - \dot{at}(j_2, k) \right)$$

4.2 Modeling Task Activities and Infinite Loops

In task scheduling, a task j consists of a vector $[a_0, a_2, \ldots a_{n-1}]$ of n activities a_i executed sequentially. At the lowest level of abstraction, an activity is a single statement in a task source code. For this reason, several task properties defined in Section 4.1 can also be considered at activity-level. For example, the duration of an activity is its WCET, while its priority is equal to the priority of its task. In particular, the delay time of an activity a_i is the minimum time that has to elapse, not considering preemptions, between the completion of a_i and the start of a_{i+1}. Since activities are executed sequentially, the arrival time of an activity a_i is the time when the preceding activity a_{i-1} finishes executing, plus the delay of a_{i-1}. Task interactions can also be considered at activity-level, as activities may depend on, or trigger other activities in different tasks. For instance, an activity may trigger another activity of a waiting task by sending a specific message to that task, or can launch a new task by triggering its first activity. Therefore, a task $j = [a_0, a_1, \ldots a_{n-1}]$ with priority p consisting of n activities a_i can be considered for scheduling purposes as a vector $[j_0, j_1, \ldots j_{n-1}]$ of n tasks with priority p, where the duration of j_i is equal to the duration of a_i, and where j_i triggers j_{i+1}. In this case, each task j_i inherits the dependencies and triggering relationships of the corresponding activity a_i. Note that this property holds under the assumption that the RTOS overhead for managing tasks in negligible compared to their execution and interarrival times. This assumption has proven to be realistic in most RTES [35].

Given this mapping between activities and tasks, we can model tasks enclosing activities in infinite loops, such as *IODispatch* (Figure 1b). Consider the task $j = [a_0, \ldots a_{n-1}]$, where the n activities are enclosed in an infinite loop. j can be modeled through a vector of $n+1$ tasks $[j'_0, j_0 \ldots j_{n-1}]$. In the vector, j'_0 and j_0 both correspond to a_0, and each other task j_i corresponds to the activity a_i. Each task in the vector has the same duration, priority, and dependencies of its corresponding activity. Each task triggers the following one forming a *triggering chain*, with the exception of j'_0 that triggers j_1, and j_{n-1} that triggers j_0. Note that, if all the activities in j are enclosed in an infinite loop, the task j'_0 is necessary in order to ensure that the COP is feasible. Consider indeed the alternative of modeling j through the tasks $j_0 \ldots j_{n-1}$, with each task triggering the following one and j_{n-1} triggering j_0. Recall that a triggered task execution arrives after when its triggering execution finishes, plus a possible delay. This is specified by the temporal ordering constraint γ_3 introduced in Section 4.1. Therefore, a circular dependency of tasks triggering each other would render the model infeasible, because the first arrival time of j_0 would depend from a previous execution of j_{n-1} that never happened. This means that the temporal ordering constraint above would result in a *non well-defined* recursion, i.e., a recursion with no base case. To overcome this issue, we model the first execution of a_0 as a separate task, namely j'_0. j'_0 is a *singular* task, i.e., a periodic task whose period is equal to the observation interval T, and hence is only executed once during the system execution. After finishing, j'_0 triggers j_1, emulating a_0 triggering for the first time a_1 in j. Fixing the first arrival time of the first activity executed during the loop allows the solver to find the arrival times of subsequent activities by unrolling the task executions in the triggering chain. Figure 2 shows how the loop in *IODispatch* is modeled through five tasks, namely *Init*, *IOBoxRead* (*IOBR*), *IOQueueWrite* (*IOQW*), *IOQueueRead* (*IOQR*), and *IOBoxWrite* (*IOBW*). In the figure, the numbers within the rectangles on the lifeline show the correspondence between activities and tasks.

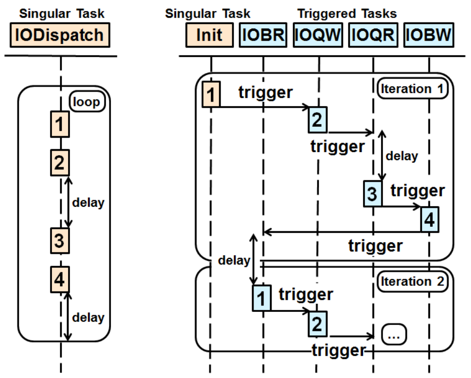


Fig. 2: Emulation of the loop in *IODispatch* through five tasks

5 Industrial Experience: Context, Process, Results, and Discussion

The work reported in this paper originates from the collaboration over the years with Kongsberg Maritime (KM)[3], a leading company in the production of systems for positioning, surveying, navigation, and automation of merchant vessels and offshore installations. When developing the software components of their real-time systems, KM faces significant challenges which have motivated our research. Therefore, the main goal of our industrial evaluation is to investigate whether CP can effectively support performance tuning in an industrial context. This aspect depends on whether we can conveniently use the output of our analysis, i.e., the values for the delay time variables in the COP, to derive configurations that satisfy the system performance requirements. In particular, we investigate this practical usefulness through two main factors. First, we note how, for practical use, performance tuning has to accommodate time and budget constraints. Therefore, we analyze the *efficiency* of our approach, i.e., the time needed to generate delay times predicted to satisfy the performance requirements. Second, recall from Section 2 that requirements on task deadlines and response times often conflict with thresholds on the CPU usage, because it is hard to achieve shorter task completion times without over-utilizing the CPU. In practice, the goal of performance tuning in the FGMS is finding safe margins in which delay times yield a trade-off between conflicting performance requirements without violating them. For this reason, we also analyze the *effectiveness* of our approach, i.e., the capability of the generated delay times to lead to scenarios achieving such satisfactory trade-off between performance requirements.

Experimental Design. Recall from Section 2 that we characterize system configurations by delay times between activities in the *IODispatch* task of the FGMS drivers. Therefore, such delay times are the main independent variables in our constraint model (Section 4). We performed an experiment with the FGMS drivers using an observation interval T of five seconds, assuming, in accordance with the specification of the RTOS executing the FGMS, time quanta of 10 ms. The search for optimal solutions was driven by a lexicographic multi-objective optimization. In lexicographic ordering, the first criterion is considered as the most important one, so that its improvement is worth any loss on the other criteria. The second criterion is the second most important, so that only losses on the first criterion are not allowed for its improvement, and so on. Using multi-objective optimization allows us to identify a Pareto-optimal frontier of solutions that are *non-dominated*, i.e., solutions x^* for which no other solution x exists such that x has a better objective value than x^* for *all* the criteria. The solutions in the frontier achieve an optimal trade-off between the search criteria, because any solution with a better objective value for one criterion has a worse objective value for at least another criterion. Investigating solutions in the Pareto frontier is particularly useful to evaluate trade-offs of conflicting optimization criteria, such as F_{RT} and F_{CU}.

We run our model for six times, one for each lexicographic permutation of F_{DM}, F_{RT}, and F_{CU}. Each run was performed on an Amazon EC2 *m2.xlarge* instance[4] with a timeout of two hours, after which the solver was instructed to terminate. We also recorded the computation times of the first solutions predicted to satisfy F_{DM}, F_{RT}, and F_{CU}. Consistent with the terminology used in Integer Programming, we refer to

[3] `http://www.km.kongsberg.com` [4] `http://aws.amazon.com`

these (sub)optimal solutions as *incumbents* [2]. The COP consisted of approximately 500 variables and one million constraints, and used up to 10 GB RAM during resolution.

Results and Discussion — Efficiency. Figure 3 shows 18 graphs reporting the experimental results for the six runs. The graphs are organized in a matrix, where each row corresponds to an objective function (F_{CU}, F_{DM}, and F_{RT}, respectively), and each column corresponds to a run. Runs are reported in the format *XX-YY-ZZ*, where each group of two letters corresponds to an optimization criterion, with *XX* being the most important, *YY* being the second most important, and *ZZ* the least. In each graph, the x-axis reports the incumbent computation times in the format *hh:mm:ss*, and the y-axis reports the corresponding objective value. The graphs related to F_{CU} and F_{RT} also report an horizontal line representing the maximum threshold on the performance requirement. Note that, being defined as an exponential function of task deadline misses, F_{DM} has no threshold on its value. In each graph, we also highlight in a circle (◯) the first incumbent predicted to satisfy the relative performance requirement, and in a square (□) the first incumbent predicted to satisfy *all* the requirements. For these incumbents, we report in a box their computation times and objective values. Finally, we report at the top right of each column the total number of solutions found in the run, and at the top center of each graph the number of incumbents satisfying the requirement. Recall from Section 4 that each solution of our COP is a sequence of task delay times.

To support engineers in configuring performance-related parameters of RTES, our approach should be able to efficiently produce usable results. In particular, engineers need to know for how long on average they should run our COP. The six runs found a total of 71 incumbents, terminating with proof of optimality in less than one hour when choosing F_{DM} as the primary optimization criterion (third and fourth column in Figure 3). With the exception of F_{CU} in the runs *CU-DM-RT*, *CU-RT-DM*, and *DM-RT-CU*, the first solution predicted to satisfy any of the performance requirements was found in less than a minute. In these three cases, the first solution predicted to exhibit a CPU usage less than 20% was found approximately after 28, 27, and 15 minutes, respectively. In each run, the incumbents found presented no deadline misses. Overall, our COP was able to find solutions predicted to satisfy *at least* one performance requirement in less than one minute, and *all* the requirements in less than half an hour. In particular, the runs *DM-CU-RT*, *RT-CU-DM* and *RD-DM-CU* found the first solutions satisfying all the requirements in approximately 30 seconds, while the other runs did so in approximately 28 minutes. The delay times characterizing these solutions can be used to derive and test initial system configurations while the search continues, because the lower the objective value, the more likely the solutions are to satisfy the systems performance requirements. In summary, it is sufficient to run our COP for half an hour on the FGMS I/O drivers in order to find solutions satisfying all the requirements.

Results and Discussion — Effectiveness. As explained above, engineers are particularly interested in finding ranges of delay times where conflicting performance requirements are close to their thresholds, but are not violated. To find these ranges, we first have to identify the conflicting requirements by analyzing the trend of the objective functions. We note how, in each run, the objective value over time related to the first optimization criterion presents a monotonic decreasing trend. This is expected because each run performs a lexicographic optimization, for which any gain on the primary criterion

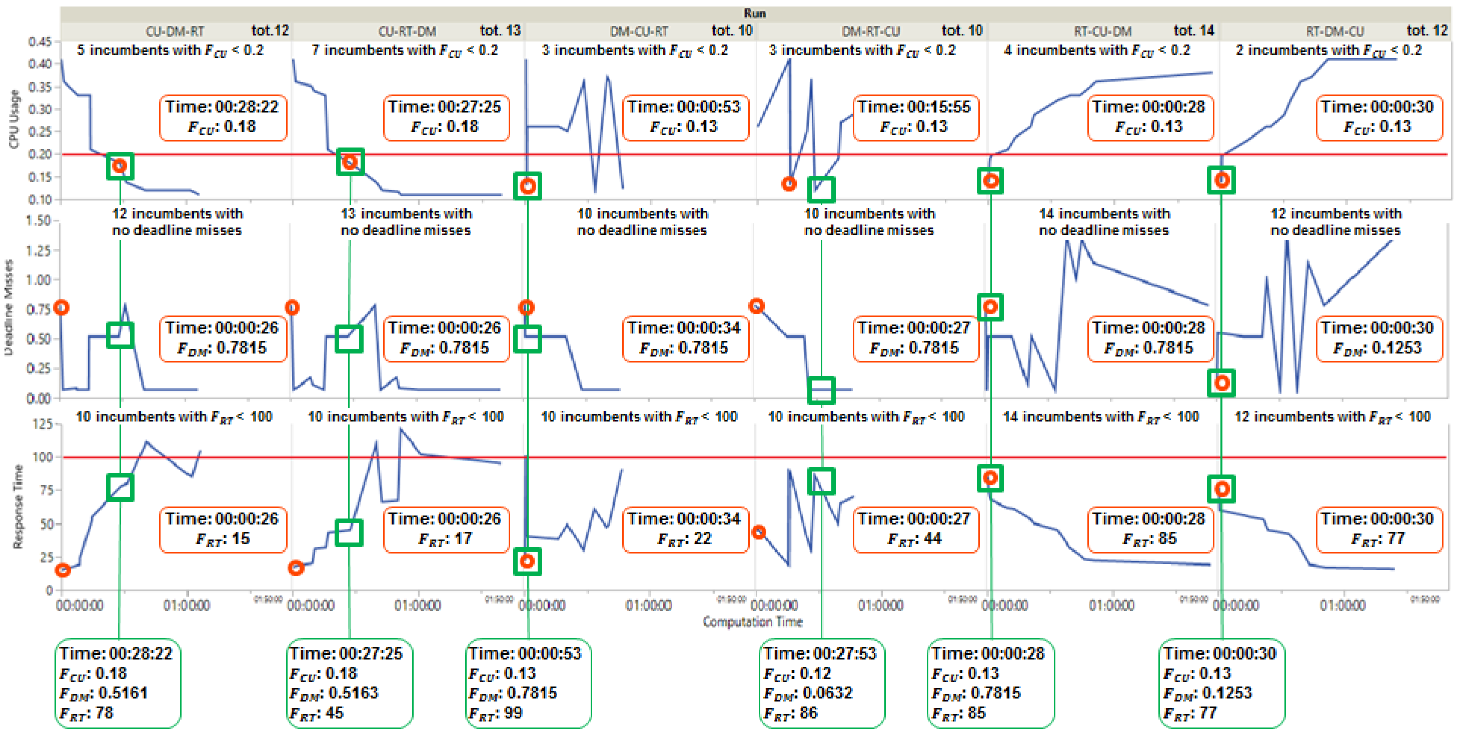


Fig. 3: Objective values of F_{CU}, F_{DM}, and F_{RT} over time, grouped by run

is worth loss on the others. When looking into the trend of specific criteria over runs, F_{DM} shows no significant correlation with F_{CU} and F_{RT}. This seems counterintuitive, because tasks are likely to miss their deadlines if the response time is too long, and when the response time is short, tasks are likely to complete long before their deadline. However, even though the exponential shape of F_{DM} is very sensible to variations in task deadlines, the fluctuations in the objective value are several orders of magnitude smaller than the size of the observation interval T. We also note how F_{CU} and F_{RT} present an inversely proportional trend. Indeed, in the runs where F_{CU} is the first optimization criterion (first two columns in Figure 3), F_{RT} tends to decrease over time, and vice versa (last two columns in Figure 3). This is also expected because, as explained in Section 2, short delay times make *IODispatch* keep the CPU busy, while long delay times are likely to block *PullData*, increasing the drivers response time. Therefore, when configuring delay times, it is necessary to analyze the trade-off between expected response time and CPU usage. Note that every incumbent found satisfies the requirement on task deadlines, and hence this trade-off analysis can ignore F_{DM}.

Figure 4 shows the Pareto-optimal frontier of F_{CU} and F_{RT}, whose solutions are highlighted with a solid bullet (●). The circle (◯) highlights the first solution found in the frontier that satisfies all the requirements, for which we report computation time and objective values. Similar to Figure 3, the two orthogonal lines represent the maximum threshold on F_{CU} and F_{RT}. Over the six runs, the search found ten solutions in the frontier, four of which satisfy the performance requirements. The first of such solutions was found in approximately 27 minutes, and corresponds to that highlighted in the *CU-RT-DM* run in Figure 3. By definition, the solutions in the frontier do not Pareto-dominate each other, entailing that for each solution in the frontier there does not exist any other solution with a lower CPU usage *and* a lower response time. Therefore, the solutions in the frontier achieve an optimal trade-off between CPU usage and response time, and can be used to derive configurations that are as likely as possible to exhibit low CPU usage and response time. Finally, recall that the six solutions in the frontier above the 20% CPU threshold might not violate such requirement at runtime due to pessimistic WCET estimates. These solutions are still worth being investigated, albeit with lower priority than the others. In particular, our experience suggests that the most valuable solutions lie in the extreme regions of the Pareto frontier, close to the highest value for a single objective function, and in the central part, where the performance requirements are equally far from their maximum values. In fact, solutions in extreme regions can be used to push the system performance to the limit, while solutions in the central area guarantee a balance between conflicting requirements.

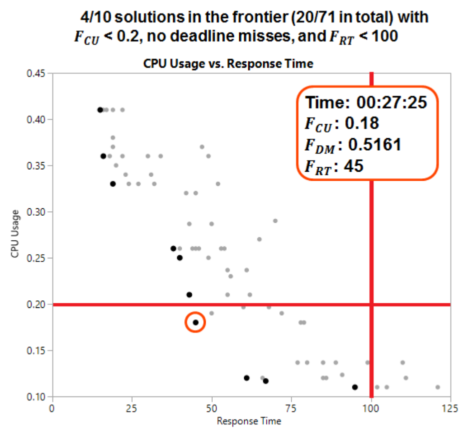


Fig. 4: Pareto-optimal frontier (solid bullets) of F_{CU} and F_{RT}

6 Concluding Remarks

In this paper, we presented a multi-objective Constrained Optimization Problem (COP) for generating RTES configurations characterized by values of configurable timing properties that satisfy a set of performance requirements. In particular, we presented a COP whose solutions are task delay times that characterize scenarios where tasks are as far as possible from their deadlines, and exhibit low response time and CPU usage. However, we note that casting the scheduling analysis of RTES as a COP is a flexible strategy that can be tailored to support activities in different phases of software development, such as stress testing and performance tuning, as well as to suit different application scenarios. For example, in order to generate task offsets that satisfy a requirement on minimal throughput, we would only need to modify the existing COP by (1) specifying task offsets as variables, rather than constants, and (2) defining a new throughput objective function. As another example, to target a system with a priority ceiling scheduling policy, we would have to modify only the preemption constraints by specifying that tasks locking a resource shared with a high priority task cannot be preempted. These adaptations would be similar to that done in this paper with respect to previous work in the area of stress testing [14].

We validated our approach on a RTES from the maritime and energy domain concerning safety-critical device drivers, showing that our approach is able to find Pareto-optimal solutions with respect to CPU usage and response time in less than half an hour. Recall that our approach builds also upon previous work in the context of performance analysis, which introduces a conceptual model to capture the timing and concurrency abstractions required to analyze response time and CPU Usage in RTES [30]. Those abstractions form the basis of both the COP presented in that work, and that presented in this paper. The effort to capture the input data for that approach was approximately 25 man-hours of effort [30]. This was considered worthwhile as drivers typically have a long lifetime and have to be certified regularly. We note how both COPs have the same set of constants, and are applied to FGMS I/O drivers having similar architectural design, and hence, the overhead for deriving the COP constant values is comparable in both cases. Furthermore, the design of our COP ensures that the final users, i.e., software engineers, can simply use it as a black box configuration generator, without having to be aware of the mathematical details of the COP. Currently, KM engineers spend several days simulating the FGMS behavior with manually tuned delay times, and monitoring its performance requirements. We expect that, by following our approach, they can configure the delay times in the FGMS drivers more conveniently, and ensure that no safety risks are posed by violating performance requirements at runtime.

Previous work in the field of software stress testing has shown that approaches based on complete search can potentially be more effective than metaheuristics in finding task schedules closer to the global optimum [13]. This aspect motivated us to investigate complete search strategies also in the context of performance tuning. However, we solve the COP with a off-the shelf solver that performs a deterministic complete search. This means that solving the COP multiple times within a time budget always finds the same set of solutions. To diversify the configurations found, we plan to combine complete deterministic search with randomized metaheuristics in hybrid strategies, which have already been successfully applied to stress test RTES [11].

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on. pp. 414–425. IEEE (1990)
2. Atamtürk, A., Savelsbergh, M.W.: Integer-programming software systems. Annals of Operations Research 140(1), 67–124 (2005)
3. Baker, T.P.: An analysis of fixed-priority schedulability on a multiprocessor. Real-Time Systems 32(1-2), 49–71 (2006)
4. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-based scheduling: applying constraint programming to scheduling problems, vol. 39. Springer (2001)
5. Briand, L.C., Labiche, Y., Shousha, M.: Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. Genetic Programming and Evolvable Machines 7(2), 145–170 (2006)
6. Cambazard, H., Hladik, P.E., Déplanche, A.M., Jussien, N., Trinquet, Y.: Decomposition and learning for a hard real time task allocation problem. In: Principles and Practice of Constraint Programming–CP 2004, pp. 153–167. Springer (2004)
7. Caseau, Y., Laburthe, F.: Improved clp scheduling with task intervals. In: ICLP. pp. 369–383. Citeseer (1994)
8. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Tools for Practical Software Verification, pp. 1–30. Springer (2012)
9. David, A., Illum, J., Larsen, K., Skou, A.: Model-based framework for schedulability analysis using UPPAAL 4.1. Model-Based Design for Embedded Systems p. 93 (2010)
10. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys (CSUR) 43(4), 35 (2011)
11. Di Alesio, S., Briand, L., Nejati, S., Gotlieb, A.: Combining genetic algorithms and constraint programming to support stress testing of task deadlines. ACM Transactions on Software Engineering & Methodology (2015)
12. Di Alesio, S., Gotlieb, A., Nejati, S., Briand, L.: Testing deadline misses for real-time systems using constraint optimization techniques. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. pp. 764–769. IEEE (2012)
13. Di Alesio, S., Nejati, S., Briand, L., Gotlieb, A.: Stress testing of task deadlines: A constraint programming approach. In: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on. pp. 158–167. IEEE (2013)
14. Di Alesio, S., Nejati, S., Briand, L., Gotlieb, A.: Worst-case scheduling of software tasks – a constraint optimization model to support performance testing. In: Principles and Practice of Constraint Programming (CP 2014) (2014)
15. Galorath, D.D., Evans, M.W.: Software sizing, estimation, and risk management: when performance is measured performance improves. CRC Press (2006)
16. Gomaa, H.: Designing concurrent, distributed, and real-time applications with UML. In: Proceedings of the 28th International Conference on Software Engineering. pp. 1059–1060. ACM (2006)
17. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: FM 2006: Formal Methods, pp. 1–15. Springer (2006)
18. Hladik, P.E., Cambazard, H., Déplanche, A.M., Jussien, N.: Solving a real-time allocation problem with constraint programming. Journal of Systems and Software 81(1), 132–149 (2008)
19. Huang, W., Lam, H.: Using genetic algorithms to optimize controller parameters for hvac systems. Energy and Buildings 26(3), 277–282 (1997)

20. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. The journal of logic programming 19, 503–581 (1994)
21. Jain, R.: The art of computer systems performance analysis. John Wiley & Sons (2008)
22. Kopetz, H.: Real-time systems: design principles for distributed embedded applications. Springer (2011)
23. Laborie, P.: IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 148–162. Springer (2009)
24. Lala, J.H., Harper, R.E.: Architectural principles for safety-critical real-time applications. Proceedings of the IEEE 82(1), 25–40 (1994)
25. Le Pape, C., Baptiste, P.: An experimental comparison of constraint-based algorithms for the preemptive job shop scheduling problem. In: CP97 Workshop on Industrial Constraint-Directed Scheduling. Citeseer (1997)
26. Lee, E.A., Seshia, S.A.: Introduction to embedded systems: A cyber-physical systems approach. Lee & Seshia (2011)
27. Malapert, A., Cambazard, H., Guéret, C., Jussien, N., Langevin, A., Rousseau, L.M.: An optimal constraint programming approach to the open-shop problem. INFORMS Journal on Computing 24(2), 228–244 (2012)
28. Nejati, S., Adedjouma, M., Briand, L.C., Hellebaut, J., Begey, J., Clement, Y.: Minimizing cpu time shortage risks in integrated embedded software. In: Automated Software Engineering (ASE), 2013 IEEE/ACM International Conference on. pp. 529–539. IEEE (2013)
29. Nejati, S., Briand, L.C.: Identifying optimal trade-offs between cpu time usage and temporal constraints using search. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 351–361. ACM (2014)
30. Nejati, S., Di Alesio, S., Sabetzadeh, M., Briand, L.: Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing. In: Model Driven Engineering Languages and Systems, pp. 759–775. Springer (2012)
31. Reza Hejazi, S., Saghafian, S.: Flowshop-scheduling problems with makespan criterion: a review. International Journal of Production Research 43(14), 2895–2929 (2005)
32. Singh, A.: Identifying Malicious Code Through Reverse Engineering. Springer (2009)
33. Sprunt, B., Sha, L., Lehoczky, J.: Aperiodic task scheduling for hard-real-time systems. Real-Time Systems 1(1), 27–60 (1989)
34. Storey, N.R.: Safety critical computer systems. Addison-Wesley Longman Publishing Co., Inc. (1996)
35. Tanenbaum, A.S.: Modern operating systems. Pearson Education (2009)
36. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocessing and microprogramming 40(2), 117–134 (1994)
37. Van Hentenryck, P.: The OPL optimization programming language. MIT Press (1999)
38. Varšek, A., Urbančič, T., Filipič, B.: Genetic algorithms in controller design and tuning. Systems, Man and Cybernetics, IEEE Transactions on 23(5), 1330–1339 (1993)
39. Weyuker, E.J., Vokolos, F.I.: Experience with performance testing of software systems: issues, an approach, and case study. Software Engineering, IEEE Transactions on 26(12), 1147–1156 (2000)
40. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problem – overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 7(3), 36 (2008)
41. Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Future of Software Engineering, 2007. FOSE'07. pp. 171–187. IEEE (2007)
42. Yun, Y.S., Gen, M.: Advanced scheduling problem using constraint programming techniques in SCM environment. Computers & Industrial Engineering 43(1), 213–229 (2002)