Integrated Development Framework for Safety-Critical Embedded Systems.

Luca Santinelli, Frédéric Boniol, Eric Noulard, Claire Pagetti, W. Puffitsch

Integrated Development Framework for Safety-Critical Embedded Systems

Luca Santinelli[1], Frédéric Boniol[1], Eric Noulard[1], Claire Pagetti[1] and Wolfgang Puffitsch[2]

[1] ONERA Toulouse, name.surname@onera.fr
[2] Technical University of Denmark, wopu@dtu.dk

Abstract. This paper presents an integrated framework for designing and implementing safety-critical embedded systems. The development begins with the specification of the system using the PRELUDE language. Then there is the compilation step, where the PRELUDE compiler translates the program into a set of communicating periodic tasks that preserve the semantics of the original program. The necessary schedulability analysis is performed with the SCHEDMCORE analyzer that explores the timing requirements of the periodic tasks concluding about the program schedulability. Finally, the task set can be executed on the single- or multi-core architecture target using the SCHEDMCORE execution environment. We outline the benefits of an integrated development framework by applying it to the task mapping problem, the functional requirement and non-functional requirement co-scheduling problem, and the measurement-based probabilistic timing analysis problem.

1 Introduction

The development of safety-critical embedded systems undergoes strict development processes. The main idea is to begin the development from the specification of the system, then to consider each step of the development as a refinement of the system itself. Each refinement is asked to produce a new system coherent with the previous: at each stage the overall requirements have to be preserved . The flight control system of an aircraft is an explicative example of a safety-critical embedded system that has to be developed following DO-178B [1].

1.1 Context

In this context, the purpose of the paper is to present an integrated development framework for safety-critical embedded systems that goes from the programming of high level specifications to the execution of automatically generated, semantically equivalent multithreaded code on both single-core and symmetric multi-core architectures. In between the programming and the execution there is the need for schedulability analysis of the task set over the architecture considered. The respect of the timing constraints is the safety level we are considering while presenting our integrated development framework. Other safety mechanisms can be easily applied at each step of the development chain.

A correct implementation of safety-critical embedded systems implies both functional and non-functional predictability. On the one hand, the functional determinism comes

from the guarantee that the outputs of the system are always the same for a given sequence of inputs. This implies fully deterministic task communications: for any execution of the system the same task instance of the communication producer must communicate with the same task instance of the communication consumer. On the other hand, the non-functional determinism (the considered timing determinism) is achieved by guaranteeing all the system timing constraints. The completion of task instances is guaranteed within a specified time window.

The integrated development environment has to take into account both problems passing through the deterministic modeling, the deterministic analysis and the deterministic verification of safety-critical embedded systems.

An example for the safety-critical embedded systems we consider is an adapted version of the Flight Application Software (FAS) of the Automated Transfer Vehicle (ATV) designed by Astrium for resupplying the International Space Station (ISS). Figure 1 provides an informal description of its software architecture. The FAS acquires several data treated by dedicated sub-functions: orientation and speed (Gyro Acq), position (GPS Acq and Str Acq) and telecommands from the ground station (TM/TC). The *Guidance Navigation and Control* function (divided into GNC_US and GNC_DS) computes the commands to apply, while the *Failure Detection Isolation and Recovery* function (FDIR) verifies the state of the FAS and checks for possible failures. Commands are sent to the control devices: thruster orders (PDE), power distribution orders (PWS), solar panel positioning orders (SGS) and telemetry towards the ground station (TM/TC).

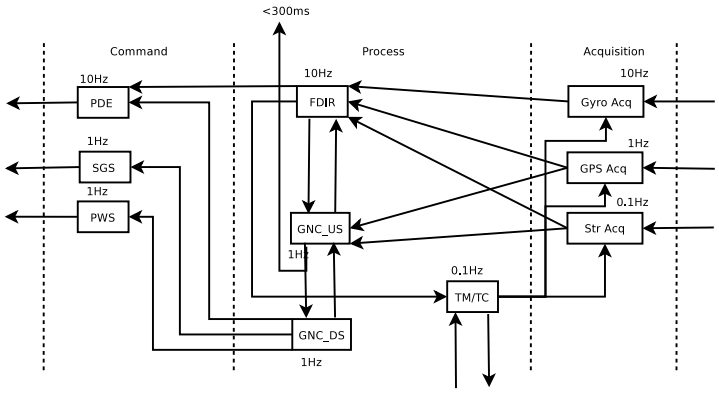


Fig. 1. FAS architecture

The specification of such a system passes through the definition of a) the functional behavior of each function, where each function can be programmed separately with existing languages, such as Simulink, Scade or directly in low-level code; b) the real-time characteristics of the system such as periods and deadlines, which depend on the physical characteristics of the system; c) the multi-rate communication patterns, which defines how data is exchanged between functions, accounting for the different periods at which they communicate.

1.2 Contribution

In this work we describe a complete operational toolset able to cope with safety-critical embedded systems. The integrated framework we have in mind is made up of three components:

1. the PRELUDE compiler, which generates code executable by the SCHEDMCORE environment;
2. the SCHEDMCORE multiprocessor schedulability analyzer;
3. the SCHEDMCORE environment, which is an extensible, easy-to-use and portable real-time scheduling framework for single- and multi-core architectures.

Each of those composing steps is detailed in the next section.

Another contribution from this paper is the illustration of how the PRELUDE-SCHEDMCORE framework is applied to three main problems for safety-critical embedded systems. They are a) the task mapping to multi-core architectures, b) the task grouping for a functional/non-functional requirement co-scheduling, and c) the execution time measurements and probabilistic worst-case execution time estimation. We show how the framework can be proficiently applied to approach them.

The organization of the paper is the following. Section 2 describes the parts composing our PRELUDE-SCHEDMCORE integrated development framework. Section 3 illustrates the task scheduling and task mapping problem and the PRELUDE-SCHEDMCORE solution proposed. Section 4 presents a grouping approach to the functional and non-functional co-scheduling. We outline possible ideas on how to integrate that within the PRELUDE-SCHEDMCORE framework. Section 5 describes the usage of SCHEDMCORE to measure task execution times. Thus a real-time execution environment developed to accurately measure execution times in different possible execution conditions. Finally, Section 6 sums up the PRELUDE-SCHEDMCORE integrated development framework and outlines future developments to it.

1.3 Related work

Development frameworks for multi-periodic systems. SIMULINK [2] is widely used in many industrial application domains and allows the description of communicating multi-periodic systems. Current code generators, such as Real-time workshop-embedded coder from the MathWorks or TargetLink from dSpace, provide a multi-threaded code generation on uniprocessor. Caspi et al. [3] translate a multi-periodic SIMULINK specification into a semantically equivalent synchronous program that is executed on a multiprocessor time triggered architecture (TTA) where no preemption is allowed.

Synchronous data-flow languages, such as LUSTRE [4] or SCADE [5], have been extended with operators that enable the specification of real-time constraints in order to program multi-threaded systems more easily [6]. Thread synchronization relies on a specific communication protocol initially defined by Sofronis et al. [7] for uniprocessors and later extended specifically for Loosely Time Triggered Architectures (LTTA) by Tripakis et al. [8].

Finally, automated distribution of synchronous programs has been studied by several authors [9,10]. All those works are considered references for PRELUDE-SCHEDMCORE, however, these studies are not dedicated to multi-periodic systems and thus scheduling policies are not considered, as indeed in PRELUDE-SCHEDMCORE.

Schedulability analysis. Lots of theoretical results on system schedulability are already available considering independent task sets [11,12]. However, there are not so many for dependent task sets and not so many tools (even for independent task sets) are available yet. Among them we mention STORM [13], which is a multiprocessor scheduling simulator, and MAST [14], to model system timing behaviors and perform schedulability analysis in presence of precedence constraints.

David et al. [15] propose a framework allowing the analysis of tasks configuration for multiprocessor machines with UPPAAL models. This framework supports rich task models including timing uncertainties in arrival and execution times, and dependencies between tasks (such as precedences and resource occupation). However, the task set should not exceed 30 tasks due to performance concerns.

Execution environments. Multiple real-time execution environments or operating systems are available, e.g., industrial ones such as VxWorks, LynxOS or PikeOS. There are also Linux variants like Xenomai, LitmusRT or SCHED_EDF for Linux (see [16] and references therein). Furthermore, academic operating systems like MARTE OS [17] and even user space runtimes like Meta-scheduler [18] exist. To covers our needs, we need an open environment that enables the implementation of user-specific scheduling policies, which rules out industrial solutions. Modified Linux kernels like LitmusRT or sub-kernel approaches like Xenomai are too closely tied to the kernel for our purpose, as this requires to patch and recompile the kernel when new releases of Linux occur. SCHEDMCORE can run on top of these environments but does not rely on them. MARTE OS [17] and the Meta-scheduler [18] answer our needs partially but, in both cases, those environments only support uniprocessor platforms. In the end, we reused the conceptual idea of a pluggable scheduler framework (of MARTE OS and the Meta-scheduler) but started with a fresh new source code.

2 Integrated Development Framework

The PRELUDE-SCHEDMCORE framework is composed by two main parts. PRELUDE is a synchronous data-flow language and thus shares similarities with LUSTRE [4], SIGNAL [19] or LUCID SYNCHRONE [20]. SCHEDMCORE [21][1] is an open-source set of tools for experimenting with real-time scheduling analysis and programming. Both have been developed at ONERA.

Figure 2 illustrates the development process within the PRELUDE-SCHEDMCORE framework. The PRELUDE specification describes functional relationships between tasks as well as the timing constraints that tasks subdue to. SCHEDMCORE is then used to check the schedulability and to execute the code on the target. In this section, we describe this tool chain. The three problems (task mapping, grouping, and execution time measurements) and their integration within PRELUDE-SCHEDMCORE are detailed in the next sections.

2.1 PRELUDE: a High Level Specification Language

PRELUDE[2] is a formal language designed for the specification of the software architecture of a critical embedded control system, [22,23]. It belongs to the family of synchronous

[1] http://sites.onera.fr/schedmcore/

[2] The PRELUDE compiler is available for download at http://www.lifl.fr/~forget/prelude.html

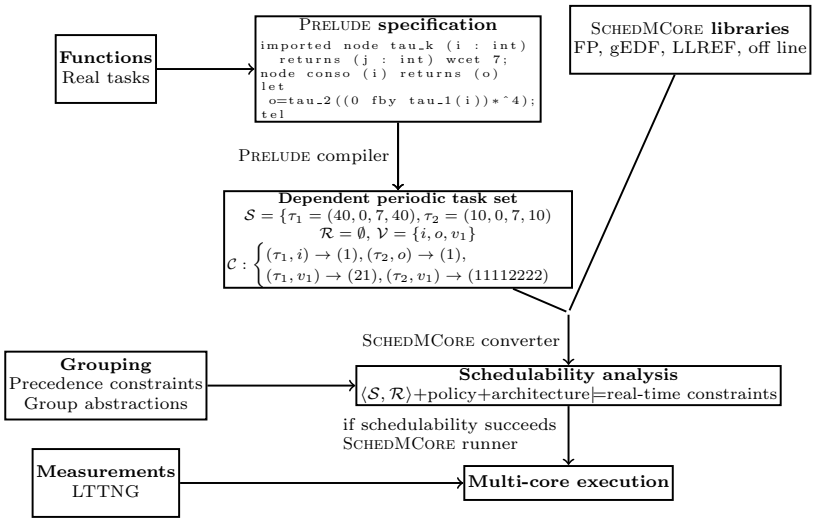


Fig. 2. Development process

data-flow languages [24] and focuses on dealing with the functional and real-time aspects of multi-periodic systems conjointly. From a PRELUDE program the compiler generates a program consisting of a periodic dependent task set. By dependent tasks we mean tasks that are linked by a functional relationship, such as precedence constraints. For each pair of producer job and consumer job of the task set the following properties have to be ensured. (1) The producer completes before the consumer starts. This is modeled by adding a precedence constraint from the producer to the consumer; the scheduler has to ensure that the schedule complies with the precedence constraint. (2) The produced data remains available until the completion of the consumer. This is fulfilled by using a specific communication protocol (directly generated by the compiler) derived from [7] and detailed in [23]. Task dependencies imply that functional constraints must be added to the classic non-functional constraints that real-time tasks subdue to.

For PRELUDE, variables and expressions are *flows*. A flow is a sequence of pairs $(v_i, t_i)_{i \in \mathbb{N}}$, where v_i is a value in some domain $\mathcal{V}$ and t_i is a date in $\mathbb{Q}$ $(\forall i, t_i < t_{i+1})$. The clock of a flow is its projection on $\mathbb{Q}$; it defines the set of instants during which the values of the flow are computed: value v_i must be computed during the interval (or instant) $[t_i, t_{i+1}[$. According to this relaxed synchronous hypothesis, different clocks have different durations for their instants. Two flows are synchronous if they have the same clock. It is possible to distinguish a specific class of clocks corresponding to periodic task activations, called strictly periodic clocks: the clock $h = (t_i)_{i \in \mathbb{N}}$ is strictly periodic if there exists some n such that $t_{i+1} - t_i = n$ for all i. n is the period of h and t_0 is its phase.

The PRELUDE language comes with a compiler that guarantees the semantic correctness of the whole compilation process and consequently of the compilation result. The initial release of PRELUDE [23] targeted uniprocessor platforms. The task set was executed with MARTE OS [17], using a scheduling policy derived from Earliest Deadline

First (EDF) [25] and from the work of Chetto et al. [26]. Cordovilla et al. [21] extended the compiler to enable the execution of PRELUDE programs on a multi-core architecture.

2.2 SCHEDMCORE: A Multiprocessor Schedulability Analyzer and Execution Environment

The task set generated by PRELUDE must be scheduled in a way that respects the real-time attributes of each task and the extended precedence constraints between the tasks. The schedulability analysis within the SCHEDMCORE framework determines whether a task set can be correctly scheduled by a given policy.

SCHEDMCORE is provided as a fully operational toolset with the SCHEDMCORE CONVERTER and the SCHEDMCORE RUNNER as the main components.

SCHEDMCORE CONVERTER takes as input a task model and generates a formally analyzable model in C or UPPAAL, which encodes all the possible execution sequences of the system as a finite automaton. Then, using either UPPAAL model checker or a generated custom C program, the automaton is explored in order to find executions that violate the system constraints. This automaton can also be used to generate off-line schedules, which can even satisfy optimality requirements for fixed-priority assignments. The scheduling policies SCHEDMCORE implements are able to cope with multiprocessor platforms and consider preemptive policies accepting full migration, non-preemptive and partitioned policies. In particular, the implemented policies are Fixed Priority (FP), global Earliest Deadline First (gEDF), global Least Laxity First (gLLF), and Largest Local Remaining Execution First (LLREF).

SCHEDMCORE RUNNER constitutes the execution environment (or runtime) tool which is capable of running a set of real-time tasks with the previously mentioned scheduling policies. It is realized as a pluggable scheduling framework which envisions precise real-time execution on a multi-core target architecture. With this tool, all the required real-time support can be provided: memory locking, switching to real-time scheduling policy, core affinity etc. The SCHEDMCORE RUNNER requires as inputs a) the task set description, which can be either a simple descriptive text file or a dynamic library produced by the compilation of a C file generated by PRELUDE [27]; b) the number of cores to be used, and, possibly, a core affinity mask; c) the scheduling policy to be used: besides the ones listed above, the user is allowed to create his own customized policy, pluggable into the runtime system.

The SCHEDMCORE runtime is implemented as a user-space library, in order to avoid the burden of going inside the OS kernel. In particular, its evolution does not need to follow the underlying operating system, allowing easy maintenance and portability, without the need of patching the Linux kernel. The same approach has been used in the past [18], and a recent study demonstrated that such an approach can be efficient [28].

3 The Task Scheduling and Task Mapping Problem

The purpose of an integrated framework for embedded systems is to help the designer finding an adequate solution for scheduling task sets using any scheduling policy.

In general, the scheduling problem involves functional and non-functional requirements. A task set is *feasible* for a given architecture if there exists a schedule that

respects the temporal and the precedence constraints of every task (regardless of any specific policy). A scheduling algorithm is *optimal*, with respect to an architecture and a class of policies (preemptive/non-preemptive, static/dynamic priority, etc.), if it can schedule all the feasible task sets.

The task set schedulability is fully integrated into the PRELUDE-SCHEDMCORE framework, and in the following we discuss solutions to the off-line scheduling problem as well as task mapping to multi-core architectures.

3.1 Off-line Scheduling of Dependent Periodic Tasks

An initial solution to the off-line scheduling, presented by Cordovilla et al. [29], was the brute force exploration of priority assignment as proposed by Cucu and Goossens [30]. Such a method quickly encounters the state space explosion problem, and there is a need for heuristics. Cordovilla et al. [29] proposed a sub-optimal heuristic. It offers good results in the sense that it finds solutions for task sets that are non-trivial both in size and "hardness". The heuristic is a modified Audsley-like algorithm based on a branch and bound algorithm with efficient cuts by using schedulability results. The implementation is in C.

Computing an off-line sequence requires to have a cycle of repetition. On uniprocessors, the feasible window [31] for independent constrained deadline synchronous task sets is known to be $[0, H]$, with H being the task set's hyperperiod. In case of independent constrained deadline asynchronous task sets on uniprocessors, it is $[0, max_i(O_i) + 2H]$. As soon as there are precedences, the window is more complex to determine [32]. In the multiprocessor case, Cucu and Goossens [30] have proved the feasible window to be $[0, H]$ in case of synchronous independent constrained deadline sets.

Xu and Parnas [33,34] have worked on *pre-run-time scheduling* which is equivalent to off-line feasible schedule. They propose an optimal scheduling method based on a branch and bound approach for synchronous dependent periodic task sets on uniprocessor and multiprocessor platforms with additional constraints such as mutual exclusion. Shepard and Gagné [35] extend the results of Xu and Parnas for multiprocessors without migration. The model the problem as bin packing problem that consists in finding an adequate partitioning of the tasks on the processors and then applying uniprocessor scheduling. Behrmann et al. [36] use priced timed automata for searching an optimal schedule for a set of jobs related by precedence constraints. They apply the model checker UPPAAL [37] for the effective search. Their modeling relies on several automata and states which is not efficient.

Cordovilla et al. [29] define an optimal search of off-line global preemptive schedules for asynchronous periodic dependent task sets. The general problem of finding such a schedule is NP-hard in the strong sense [38]. The search in [29] is implemented in UPPAAL due to the complexity of the configuration graph; model checkers that are dedicated to the efficient exploration of such data structure are better suited. In there it has also illustrated the integration within the PRELUDE-SCHEDMCORE framework.

3.2 Task Mapping Problem

The PRELUDE-SCHEDMCORE framework can also be used to execute real-time systems on a many-core platform [39]. The high-level system specification is translated in order to generate a mapping of real-time tasks to the underlying many-core platform. Then,

in [39] it has been addressed a many-core Single-chip Cloud Computer (SCC) architecture, which scheduler is not the scheduler on SchedMCore. The applicability to the SchedMCore scheduler is under investigation.

In the following, we briefly outline the main task mapping steps for many-core architectures. This gives us a further insight on how the scheduling/mapping problem is integrated into the Prelude-SchedMCore framework.

Given a periodic task set, the objective is to define a static mapping on the many-core platform. This consists in defining on which core each task executes and also mapping the communication buffers created by Prelude to the message passing infrastructure. Due to the large number of tasks and cores, performing this mapping manually would be time-consuming and error-prone. Therefore, the mapping process is required to be automated as far as possible.

Puffitsch et al. [39] developed a partitioning heuristic that takes into account the specifics of the real-time task set, the run-time system, and the underlying hardware. A cost model is detailed for the Intel SCC architecture, and the heuristic is based on that cost model. The partitioning heuristic relies on the schedulability test for non-preemptive EDF scheduling of Fisher and Baruah [40]. While their schedulability test does not consider dependencies as in our case, it is discovered that it approximates schedulability reasonably well when additionally bounding the load on each processor. To ensure the validity of the generated partitioning, the Prelude-SchedMCore tool chain integrates a schedulability analysis for non-preemptive partitioned EDF scheduling.

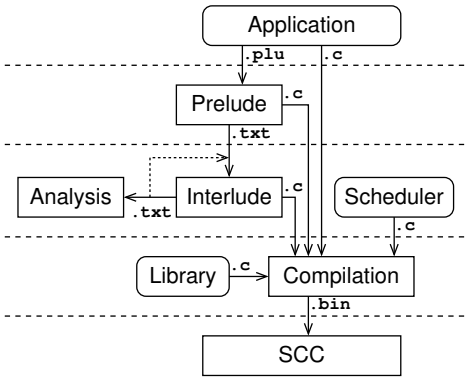


Fig. 3. Tool flow

Figure 3 shows the flow of information between the individual tools in the proposed framework. The top layer is the application, which comprises the implementations of the individual tasks in C, and a description of the communication between tasks in a Prelude source file (`.plu`). This description is translated by Prelude, which generates C code that implements the communication between tasks. The general communication mechanism is kept, and Prelude is modified to leave the allocation of communication buffers to later stages and to access them via functions instead of direct reads and writes. Furthermore, Prelude generates a description of the tasks properties, their dependencies, and the buffers required for communication to a `.txt` file. This task set description is mapped to the SCC by Interlude. Interlude emits a description of the task set

like PRELUDE, with additional information about the mapping of tasks to cores. This description is then passed on to the schedulability analysis. In case the generated mapping is unschedulable, the description can be edited to pre-map tasks to cores and avoid undesirable configurations. INTERLUDE also generates C code describing the mapping of tasks to cores, their dependencies, and the locations of communication buffers.

4 The Grouping Problem

Due to its complexity, the scheduling/mapping problem does not scale to huge task sets. That is the reason why Santinelli et al. [41] define an approach to group tasks in order to translate task scheduling into group scheduling. The idea of the grouping is to create groups of tasks and make the functional/non-functional co-scheduling problem tractable by reducing the number of elements to be scheduled. Some reference to the grouping comes form papers about clustering and functional partitioning, [42,43,44].

The grouping is a two-stage approach where first tasks are grouped according to their functional requirements, and then the groups of tasks are scheduled to guarantee the timing requirements.

4.1 Task Grouping

A real-time system can be seen as a task set $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$ where tasks are described through their parameters (worst-case execution time, period, deadline, etc.) and their mutual relationships. In particular, Santinelli et al. [41] considered the functional dependencies as precedence constraints or data-flows. The precedence constraints between tasks are described as a directed acyclic graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ where $\mathbb{V}$ is the set of tasks Γ, $\Gamma \equiv \mathbb{V}$, and $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is the set of edges, which represent the precedence constraints between tasks.

The grouping classification is according to the task dependencies and a grouping partition $\mathcal{G} = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$ divides the task set into disjoint subsets such that $\forall \mathcal{G}_i, \mathcal{G}_j \in \mathcal{G},\ \mathcal{G}_i \cap \mathcal{G}_j = \emptyset$ and $\bigcup_{i=1}^{n} \mathcal{G}_i = \mathbb{V}$.

The criteria for grouping may be chosen arbitrarily, and some grouping are more helpful with regard to scheduling than others. Santinelli et al. [41] focused on two classes of grouping: a) *independence grouping*, which exploits the notion of independence to partition the task set, b) *dependence grouping*, which creates groups of dependence tasks.

Independence Grouping. We call a grouping $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots\}$ an independence grouping if only independent tasks ($\overline{\rhd}$) belong to the same partition $\mathcal{I}_i$, $\forall\ \mathcal{I}_i \in \mathcal{I}, \quad \forall\ \tau_j, \tau_k \in \mathcal{I}_i, \quad \tau_j \overline{\rhd} \tau_k$.
$\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots\}$ partitions the task set into groups $\mathcal{I}_i$, where all the tasks within a group are mutually independent. Santinelli et al. [41] developed *forward and backward independence grouping algorithms* to partition the graph into independence groups, starting from the beginning or the end of the graph, respectively.

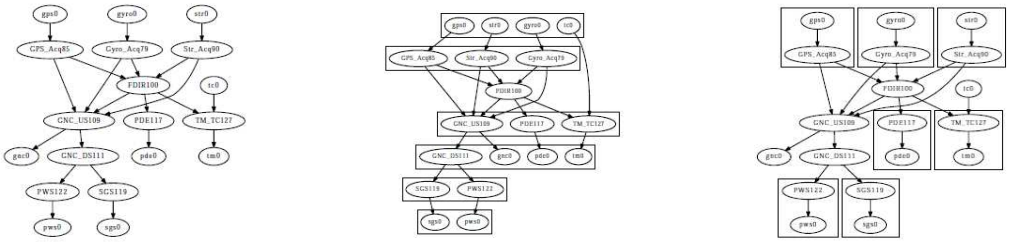
Dependence Grouping. Instead of grouping independent tasks, we can group tasks that form chains of dependent tasks. Two tasks τ_i and τ_k form a chain if τ_k is the only successor of τ_i and τ_i the only predecessor of τ_k, or if there exists a sequence of chains between τ_i and τ_k through intermediate tasks. This way, *a dependence grouping $\mathcal{D}$ is a partitioning of the task set such that chains of dependent tasks belong to the same group*

$\mathcal{D}_i$. A dependence grouping $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots\}$ partitions the task set into groups $\mathcal{D}_i$, where all the tasks within a group are dependent. A *dependence grouping algorithm* [41] starts from the tasks without any predecessors and iterates over them creating a group for each of these, which includes the task and all those that form a chain with it.

The grouping reduces the scheduling possibilities (the degree of flexibility) and introduces some pessimism into the task timing constraints. The pessimism depends on the case study considered (the graph topology), but also on the latencies applied. On the other hand, the scheduling complexity is drastically reduced with the grouping, since the number of elements to be scheduled is smaller than the case without grouping.

4.2 Grouping Application.

Santinelli et al. [41] applied the grouping abstraction to the FAS initial motivational example, Figure 1. Figure 4(a) shows the graph representation for the precedence constraints of the FAS task set. Figures 4(b) and 4(c) present the independence group and the dependence group classification for those tasks. In those figures it is possible to infer the pessimism introduced by the scheduling as well as the complexity reduction due to the groups classification and their scheduling. The FAS benchmark analyzed is a modified version with mono-rate tasks; the task period has been set to $1000ms$ and the execution times, expressed in milliseconds, have been set to fit into a $1000ms$ period. To evaluate



(a) Functional graph representation

(b) Independence grouping with the forward algorithm

(c) Dependence grouping

Fig. 4. Functional graph and grouping to the FAS benchmark.

the impact of grouping on I/O latencies and the pessimism the grouping introduces, two I/O latencies for the FAS benchmark have been defined. The first, $L(gps0, gnc0)$, with $gps0$ as the input task and $gnc0$ as the output task, and the second, $L(str0, pws0)$, with $str0$ as the input task and $pws0$ as the output task. By defining the optimal latency as the smallest latency, it is possible to have an optimization problem where all the possible task schedules are explored looking for the smallest I/O latency.

Without grouping, the optimal latencies are $275ms$ and $605ms$ for $L(gps0, gnc0)$ and $L(str0, pws0)$, respectively. In case of independence grouping (with the forward grouping algorithm) the optimal $L(gps0, gnc0)$ latency become $465ms$, while for the $L(str0, pws0)$ is $845ms$. In case of dependence grouping the optimal $L(gps0, gnc0)$ latency stays $275ms$ (as the no-grouping case), while for the $L(str0, pws0)$ it is $835ms$. Combining the two latencies, we could look for the maximum between them, $max\{L(gps0, gnc0), L(str0, pws0)\}$.

The latencies results into $615ms$ with no grouping applied, $855ms$ with the independence grouping (forward independence grouping) and $835ms$ with the dependence grouping. Results are slightly larger than the single latency case, due to the combination of the two requirements.

Grouping Integration. We are currently working on integrating the grouping approach with the PRELUDE-SCHEDMCORE framework. By definition, the grouping represents an intermediate task abstraction. Thus, it could be a module aside the schedulability analysis, as shown in Figure 2, which would take the task description in the form of precedence and timing constraints and produce the task set partitioning as output for what we could call the grouping schedulability analysis. The task set partition is then given to the SCHEDMCORE modules to verify schedulability of the groups and execute them while evaluating their timing performance. As future work, we plan to compare schedulability analysis with or without grouping in a more formal manner. Furthermore, we plan to implement the UPPAAL analysis with grouping and compare it with other schedulability analysis tools.

5 The Execution Time Measurement Problem

The schedulability analysis relies on Worst-Case Execution Time (WCET) knowledge. Recently, researchers have investigated measurements with extreme value theory as an alternative to static timing analysis. Among those works there is [45], which measures task execution times with the help of PRELUDE-SCHEDMCORE.

5.1 Tracing approaches

Measuring and observing a real-time system requires to set up an appropriate environment to execute various real-time task-sets and to collect information about their behavior at each iteration. In particular, there is need for a real-time environment that ensures real-time execution with appropriate scheduling policies, memory locking, core affinity, etc. such as SCHEDMCORE. Since the objective is to address any kind of real execution target, it is not possible to rely on simulation. The support has also to be abstract enough to approach any computational architecture and consider the widest range of execution condition for task sets.

It is also required to have a tracing tool which allows to collect timestamped traces containing execution information, such as execution time, response time, number of cache misses, etc. Hardware-specific assisted observation may be unavailable, thus the best option is to perform instrumented real execution, often called tracing, ensuring a minimal perturbation due to real-time measurements. LTTng (Linux Trace Toolkit new generation) [46] applies static instrumentation to achieve low overhead, at the cost of only a small increase in the code size.

In order to conduct low-level performance analysis or tuning, retrieving counts of hardware-related activities, hardware performance counters are a common choice, since they are available in most of modern processors, also in the embedded computing area. Hardware performance counters are a set of special-purpose registers built into modern microprocessors that track low-level operations or events within the processor accurately and with minimal overhead.

Compared to software profilers, such as gprof and Valgrind, hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, cache memories, main memory etc. Besides, in general no source code modifications are needed, although the types and the meaning of counters may vary depending on the architecture, due to the different hardware organizations.

It is also important to have a precise characterization of the benchmark set we wish to adopt, in order to quantify in advance the runtime behavior of the program. To this aim we could explore MICA[3] [47], a PIN tool[4] which allows to collect program characteristics independently from the microarchitecture on which the measurements are done.

Finally, a way to integrate realistic task implementations into the real-time execution and tracing environment is required. SchedMCore can support any C function to be loaded as the body of a task. Melani et al. [45] used CMake[5] for managing the build process of software, in particular creating a dynamic library starting from the C function we want to use as the task implementation. The result is a shared object which is the task implementation function, and a taskfile with the description of the tasks that will run in the system, their mapping on a specific core mask and the user functions to be associated to each one of the tasks. The shared object and the taskfile have all the information that SchedMCore requires to verify system schedulability and execute it.

The LTTng session has to be created, enabling all the kernel events and the *start* and *stop* events specified by the user (implemented by SchedMCore). A Python script to extract data from LTTng traces is available within SchedMCore.

5.2 Task Measurements

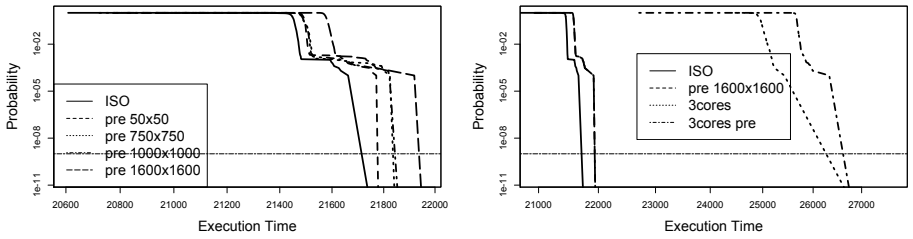
Once a real-time measurement environment with SchedMCore has been set up, it is possible to measure task execution time profiles in different execution conditions. Task executions exhibit variability due to the interferences from the system. The measured profiles are probability distribution representations of such variabilities. Those measurements can have a twofold application. At one hand they can be applied to estimate worst-case execution time and perform scheduling analysis. At the other hand, they can be used to validate worst-case execution time estimations and thus scheduling validation. Figure 2 depicts the measurements applied to generate task models.

As a first example where to apply the Prelude-SchedMCore framework to the measurement problem, we consider a *cnt* task from the Mälardalen Benchmark suite [48]. This is a single path task. With SchedMCore we are able to isolate most of the interferences and measure their effects on the task execution times. Figure 5(a) shows the execution time profile as a 1-Cumulative Distribution Function (1-CDF) of task cnt in different execution configuration. *ISO* is the isolation case where the monitored task suffers a minimal set of interferences from the system. *pre 50x50* is where the cnt task is preempted by a task with a small $50x50$ data structure; *pre 750x750* to mean that cnt is preempted by $750x750$ data structure, and so on. Figure 5(a) shows the effects that preemptions have on the caches and consequently on task execution time.

[3] http://boegel.kejo.be/ELIS/MICA/

[4] A PIN is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis, http://www.pintool.org

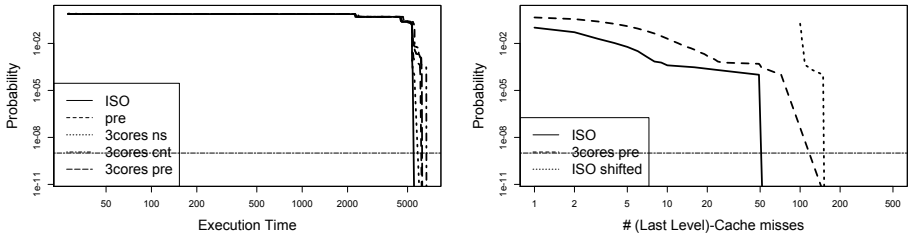
[5] www.cmake.org



(a) cnt: execution time comparison (b) cnt: execution time comparison

Fig. 5. cnt benchmark example with execution times in different conditions.

Figure 5(b) outlines the effects that multi-core (3 cores total) executions with or without preemption, have on the measured task execution time profiles. There is still the isolation case as reference, then a condition where cnt is preempted by a task (*pre 1600x1600*), and also the case with 2 more cores executing other interfering tasks.



(a) ns: execution time comparison (b) ns: number of last level cache misses

Fig. 6. ns benchmark example with execution times and cache misses compared.

The second example is the Mälardalen benchmark *ns*, which is a multi-path task. Figure 6(a) depicts the effects that multi-core interferences have on the ns execution time profiles. Through the 1-CDF representation we are able to identify those effects on all three paths of the ns task. The worst case is the case where ns suffers preemptions and there are 2 other cores (3 in total) that execute tasks and are able to interfere with the reference task ns through the shared cache. Finally, Figure 6(b) shows the flexibility of the measurement approach applied to shared caches. With SchedMCore and LTTng it is possible to accurately measure the number of cache misses per run and build probability distributions out of measurements. Intuitively, the 1-CDF representation outlines how *3cores pre* is the worst possible condition in terms of number of cache misses. The curve *ISO shifted* has been built from the isolation case in order to upper-bound the worst-condition. It could be used for safe and pessimistic cache analysis.

6 Conclusion

With this paper we have first described the integrated development framework based on PRELUDE and SCHEDMCORE. We are able to cope with the modeling, analysis, and verification of safety-critical embedded systems which combine functional requirements and non-functional requirements. We have also shown how the framework applies to problems of today's embedded systems.

The PRELUDE-SCHEDMCORE is also flexible to approach further scheduling and verification applications. The developed solutions for new problems can be plugged into the integrated framework to give it new potentials. On the other hand, to apply an integrated framework to those problems would give means to better integrate modeling, analysis, and verification, to enhance the development of safety-critical embedded systems.

References

1. *Software Considerations in Airborne systems and Equipment Certification*, RTCA, 1992.
2. The Mathworks, *Simulink: User's Guide*, The Mathworks, 2009.
3. P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From simulink to scade/lustre to tta: a layered approach for distributed embedded applications," in *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003, pp. 153–162.
4. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.
5. F.-X. Dormoy, "Scade 6 a model based solution for safety critical software development," in *Embedded Real-Time Systems Conference (2008)*, 2008.
6. A. Curic, "Implementing Lustre programs on distributed platforms with real-time constraints," Ph.D. dissertation, Université Joseph Fourier, Grenoble, 2005.
7. C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," in *Proceedings of the 6th International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, Oct. 2006, pp. 21–33.
8. S. Tripakis, C. Pinello, A. Benveniste, Albert Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale, "Implementing synchronous models on loosely time-triggered architectures," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1300–1314, Oct. 2008.
9. A. Girault, X. Nicollin, and M. Pouzet, "Automatic rate desynchronization of embedded reactive programs," *ACM Trans. Embedded Comput. Syst.*, vol. 5, no. 3, pp. 687–717, 2006.
10. P. Aubry, P. Le Guernic, and S. Machard, "Synchronous distribution of Signal programs," in *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, 1996, pp. 656–665.
11. T. P. Baker and S. K. Baruah, "Schedulability analysis of multiprocessor sporadic task systems," in *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.
12. R. I. Davis and A. Burns, "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems," University of York, Department of Computer Science, techreport YCS-2009-443, 2009.
13. R. Urunuela, A.-M. Déplanche, and Y. Trinquet, "STORM, simulation tool for real-time multiprocessor scheduling," Institut de Recherche en Communications et Cybernétique de Nantes, Tech. Rep., september 2009.
14. J. Pasaje, M. Harbour, and J. Drake, "Mast real-time view: a graphic uml tool for modeling object-oriented real-time systems," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, Dec 2001, pp. 245–256.

15. A. David, J. Illum, K. G. Larsen, and A. Skou, *Model-Based Design for Embedded Systems*. CRC Press, 2010, ch. Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1, pp. 93–119.
16. D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An edf scheduling class for the linux kernel," in *Proceedings of 2009 Real Time Linux Workshop*, 2011, revised version.
17. M. A. Rivas and M. G. Harbour, "A POSIX-Ada Interface for Application-Defined Scheduling," in *International Conference on Reliable Software Technologies, Ada-Europe 2002*, 2002, pp. 136–150.
18. P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on posix real-time operating systems," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 613–629, September 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1018037.1018391
19. A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the Signal language and its semantics," *Sci. of Compu. Prog.*, vol. 16, no. 2, 1991.
20. M. Pouzet, *Lucid Synchrone, version 3. Tutorial and reference manual*, Université Paris-Sud, LRI, 2006.
21. M. Cordovilla, F. Boniol, J. Forget, E. Noulard, C. Pagetti *et al.*, "Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset," in *19th International Conference on Real-Time and Network Systems*, 2011.
22. J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A multi-periodic synchronous data-flow language," in *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, China, Dec. 2008.
23. J. Forget, "A synchronous language for critical embedded systems with multiple real-time constraints," Ph.D. dissertation, Université de Toulouse - ISAE/ONERA, Toulouse, France, Nov. 2009.
24. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.
25. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environmet," *Journal of the ACM JACM*, vol. 20, no. 1, pp. 46–61, 1973.
26. H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, 1990.
27. C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307–338, 2011.
28. M. Mollison and J. Anderson, "Bringing theory into practice: A userspace library for multi-core real-time scheduling," in *Proceeding of 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
29. M. Cordovilla, F. Boniol, E. Noulard, and C. Pagetti, "Multiprocessor schedulability analyser," in *Proceedings of the 26th ACM Symposium on Applied Computing (SAC'11)*, 2011.
30. L. Cucu and J. Goossens, "Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems," in *Proceedings of the conference on Design, automation and test in Europe (DATE'07)*. San Jose, CA, USA: EDA Consortium, 2007, pp. 1635–1640.
31. F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, *Scheduling in real-time systems*. John Wiley & Sons, October 2002.
32. E. Grolleau and A. Choquet-Geniet, "Off-line computation of real-time schedules by means of petri nets," in *Workshop On Discrete Event Systems, WODES2000*, ser. Discrete Event Systems: Analysis and Control. Ghent, Belgium: Kluwer Academic Publishers, 2000, pp. 309–316.
33. J. Xu and D. Parnas, "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 360–369, March 1990.

34. J. Xu and D. L. Parnas, "On satisfying timing constraints in hard-real-time systems," *IEEE Transaction Software Engineering*, vol. 19, pp. 70–84, January 1993.
35. T. Shepard and J. A. M. Gagné, "A pre-run-time scheduling algorithm for hard real-time systems," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 669–677, July 1991.
36. G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Optimal scheduling using priced timed automata," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, pp. 34–40, March 2005.
37. G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, ser. LNCS, no. 3185. Springer–Verlag, September 2004, pp. 200–236.
38. J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.
39. W. Puffitsch, E. Noulard, and C. Pagetti, "Mappting a multi-rate synchronous language to a many-core processor," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
40. N. Fisher and S. Baruah, "The partitioned multiprocessor scheduling of non-preemptive sporadic task systems," in *14th International COnference on Real-Time and Network Systems*, 2006.
41. L. Santinelli, W. Puffitsch, A. Dumerat, F. Boniol, C. Pagetti, and J. Victor, "A grouping approach to task scheduling with functional and non-functional requirements," in *Embedded Real-time Software and Systems (ERTS)*, 2014.
42. C. Bartolini, G. Lipari, and M. Di Natale, "From functional blocks to the synthesis of the architectural model in embedded real-time applications," in *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, March 2005, pp. 458–467.
43. S. Kodase, S. Wang, and K. G. Shin, "Transforming structural model to runtime model of embedded software with real-time constraints," in *Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum - Volume 2*, ser. DATE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 20 170–. [Online]. Available: http://dl.acm.org/citation.cfm?id=1022685.1022945
44. A. Mehiaoui, E. Wozniak, S. Tucci-Piergiovanni, C. Mraidha, M. Di Natale, H. Zeng, J.-P. Babau, L. Lemarchand, and S. Gerard, "A two-step optimization technique for functions placement, partitioning, and priority assignment in distributed systems," in *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, 2013, pp. 121–132.
45. A. Melani, E. Noulard, and L. Santinelli, "Learning from probabilities: Dependences within real-time systems," in *8th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2013.
46. P.-M. Fournier, M. Desnoyer, and M. R. Dagenais, "Combined tracing of the kernel and applications with lttng," in *Linux Symposium*, 2009.
47. K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *Micro, IEEE*, vol. 27, no. 3, pp. 63–72, 2007.
48. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," B. Lisper, Ed. Brussels, Belgium: OCG, Jul. 2010.