Model-driven Deployment Optimization for Multicore Embedded Real-time Systems: the OptimAll Approach

Federico Ciccozzi, Juraj Feljan
School of Innovation Design and Engineering (IDT)
Mälardalen University
Västerås, Sweden
E-mail: federico.ciccozzi@mdh.se, juraj.feljan@mdh.se

Abstract—The power of modern embedded systems is continuously increasing together with their complexity, thereby making their development more challenging. In the specific case of the adoption of multicore solutions, while processing power is heavily increased, the issue of allocating software tasks to specific cores on the target platform arises. In this paper we introduce OptimAll, an automated model-driven approach that aims at providing support in the delicate phase of task allocation at design time. Besides introducing the entire approach, in this work we focus on the automatic generation of a suitable input to the task allocation optimization mechanism from a UML–MARTE system design model, as well as on the actual optimization mechanism and its outcomes in relation to the design model elements.

I. INTRODUCTION

Nowadays most computer systems that connote our everyday life are embedded and characterised by real-time properties. One of the main traits that affects the development of this kind of systems is their ever-increasing performance demand, as they include more and more complex functionality. In most cases, the higher performance needs are tackled by increasing the processing power through, e.g., the adoption of multicore and manycore solutions; in this work we focus on the former. A multicore processor is a single chip that contains two or more processing units that are tightly coupled together in order to preserve energy-efficiency.

Adopting a solution with multiple processing units introduces the challenge of how to deploy software components to the available cores to best utilize the hardware platform. In OptimAll we address deployment of software components as a two-step phase: (i) allocation of software components to software schedulable entities (i.e., tasks) and (ii) allocation of these entities to specific cores. In this work we focus on the former. A multicore processor is a single chip that contains two or more processing units that are tightly coupled together in order to preserve energy-efficiency.

In our solution, we exploit Model-Driven Engineering (MDE) [1] for providing automation and aiding the developer in the delicate phase of deployment optimization at design time. One of the main goals of MDE is to shift the focus of the development from hand-written code to models that represent an abstraction of the problem at hand, and from which early analysis, simulation and testing are made possible through the exploitation of model transformations [2].

More specifically, we exploit MDE for modelling the system under development and in particular the details related to the task allocation problem. Model transformations are defined in order to automatically generate a suitable representation of the task allocation, which can be simulated in order to obtain performance predictions. This in turn enables task allocation optimization — we assess a number of task allocation candidates in search for one that satisfies the constraints on EFPs. We also support visualization of the resulting EFPs for an allocation candidate.

Contribution. In this work we introduce OptimAll, a model-driven deployment optimization approach for multicore embedded soft real-time systems. In the long run OptimAll is meant to offer automatic full code generation capabilities as well as back-propagation features for optimization based on monitored system runs (as described in Section IV). In this paper we lay the foundations of the approach and we focus on the modelling and prediction-based optimization phases. Starting from a deployment model, we show how OptimAll automatically generates a suitable input for the task allocation optimization mechanism [3] through a set of model transformations, and then iterates the simulation runs in search for a good allocation candidate.

The remainder of the paper is organized as follows. Section II provides an overview on related approaches documented in the literature, while Section III describes the basic concepts on which our approach is built upon. In Section IV we describe the OptimAll approach in all its intended features. In Section V a running example is exploited to show OptimAll at work. The paper is concluded with Section VI where we provide a short recall of the paper’s contribution and an outlook on the coming planned activities.

II. RELATED WORK

MDE for embedded systems has a general goal of lifting the level of abstraction from code to models thus simplifying software development. Models are used both to reason about the EFPs of the system under development, and as a specification from which the implementation can be automatically generated. Regarding the former, models enable obtaining performance predictions already at an early stage of development, prior to the implementation, via model-based analysis and model simulation, in line with what software
performance engineering advocates [4]. These model-based performance predictions make it possible to quickly assess a large number of system configurations (e.g., deployment of software components to hardware nodes), thus enabling system optimization at an early development stage. In the remainder of this section, we present related work by discussing several approaches for model-driven performance prediction and optimization. However, in contrast to OptimAll, none of the approaches are specifically tailored for soft real-time multicore embedded systems, and cover the complete cycle of automatically going from models, through an optimization engine to code and back.

ProCom [5] is a component-based and model-based approach for developing automotive embedded systems. It has a notion of a rich component, which is a set of models, documentation and code. Through different modelling formalisms, ProCom can be used to analyse worst-case execution times, end-to-end response times, and resource usage. It also provides support for automatic synthesis of code from the models, however it does not enable optimization or back-propagation of information from code to models.

ArcheOpterix [6] is a framework for optimizing embedded systems modelled using the Architecture Analysis and Description Language (AADL). It supports several categories of EFPS, such as reliability and performance. The optimization mechanism employs various general purpose heuristics including genetic algorithms, Bayesian learning and hill climbing. The approach can account for uncertainties of design time parameter estimations, through its extension called Robust ArcheOpterix [7] — it proposes architectures that reduce the impact of the uncertainties.

Additional approaches (not limited to embedded systems) can be found in the survey of component-based approaches for performance evaluation by Koziolke [8], and in the survey of architecture optimization approaches by Aleti et al. [9].

III. BACKGROUND

In this section we set the domain of the work and describe techniques and technologies employed for defining the solution.

A. Task allocation problem

The domain of our work is represented by embedded soft real-time systems, where accurate timing behaviour is crucial for the correct functioning of the system, but occasional deadline misses are tolerated (as opposed to hard real-time systems where the absence of deadline misses must be guaranteed beforehand). Therefore, the EFPS we are interested in are related to timing, and include end-to-end response time, deadline misses and core load. These properties depend heavily on the allocation of tasks to cores. An intuitive example is allocating too many tasks to the same core, which will become overloaded and the tasks will therefore miss their deadlines.

It is desirable to identify a good allocation early in the development process, already prior to the implementation. The earlier design faults that lead to performance issues are caught, the cheaper and simpler it is to correct them [10]. Therefore we base our work on MDE concepts, and strongly rely on model-based analysis at design time. Since we focus on the average-case behaviour (as opposed to worst-case behaviour in hard real-time systems), and since the aforementioned EFPS depend heavily on the dynamic interplay between tasks, they cannot be derived analytically from task parameters. Rather, we obtain the property values by performing simulation of an allocation model.

B. Task model

In our approach we support tasks of two kinds: periodic and event-triggered (triggered by other tasks finishing their execution). Each task is assigned a number of parameters: priority, affinity (specifying which core the task is allocated to), best-case execution time (BCET), worst-case execution time (WCET). Moreover, in the specific case of periodic tasks, two additional parameters, namely deadline and period, are defined. An event-triggered task is considered to have missed its deadline if it is triggered again while its previous instance has still not finished executing. Currently, only a uniform distribution of task execution times is implemented. However, additional distributions are planned to be added. A task chain represents the flow of execution, and it is defined by a periodic task starting the chain and a set of event-triggered tasks triggered in ordered sequence. End-to-end response times are EFPS that are defined at chain level, and represent the duration between the point in time when the periodic task at the start of the chain begins its execution, until the point in time when the last task in the chain finishes its execution. During their execution tasks do not move between cores, as they are statically allocated. Each core has a scheduler in charge of running the tasks assigned to it and we currently support preemptive and non-preemptive fixed priority-based schedulers.

Next we formalise the aforementioned notions of task, periodic task, event-triggered task and chain.

Definition 1: A task $T$ is a non-instantiable tuple $T = \langle B, W, pr, a \rangle$, where $B$ represents $T$’s BCET, $W$ represents the WCET, $pr$ represents $T$’s scheduling priority and $a$ represents the affinity parameter identifying the core to which $T$ is allocated.

Definition 2: A periodic task $PT$ is an instantiable specialization of $T$ defined as the tuple $PT = \langle T, pe, d \rangle$, where $T$ represents the tuple $\langle B, W, pr, a \rangle$, $pe$ represents $PT$’s period and $d$ represents $T$’s deadline.

Definition 3: An event triggered task $ET$ is an instantiable specialization of $T$ defined as the tuple $ET = \langle T \rangle$, where $T$ represents the tuple $\langle B, W, pr, a \rangle$.

Definition 4: A chain $C$ is a non-empty ordered set of tasks $\{PT, T_1, T_2, \ldots, T_n\}$ with $|C| \geq 1$ and where the first element is always represented by a periodic task $PT$.

C. Modelling language

The reference modelling language exploited in the OptimAll approach is represented by UML [11] for functional descriptions, and by its profile for Modeling and Analysis of
Real Time and Embedded systems (MARTE) [12] for extra-functional as well as deployment modelling. Moreover, the approach is implemented and runs on top of MDT Papyrus [13], an open source integrated environment for editing EMF [14] models and particularly supporting UML and related profiles, on the Eclipse platform.

Concerning the functional modelling of the system we follow the component-based pattern [11] where each component is equipped with provided and required interfaces realised via ports and with state-machines and other standard UML diagrams to express functional behaviour. Moreover, the Action Language for Foundational UML (ALF) [15] is meant to be exploited for defining complex behaviours. Functional models are decorated with extra-functional information either through MARTE stereotypes or through specific annotations defined appositely for the purpose. For describing deployment information we exploit specific concepts provided by MARTE through which the modeller defines allocation of software components first to schedulable tasks and then to processing units.

D. Model transformations

Following the MDE paradigm, a system is developed by designing models and refining them starting from higher and moving to lower levels of abstraction until code is generated; refinements are performed through transformations between models. A model transformation translates a source model to a target model while preserving their well-formedness [2]. More specifically, in OptimAll we exploit the following kinds of model transformation:

- **Model-to-model (M2M):** which translates between source and target models that can be instances of the same or different languages;
- **Model-to-text (M2T):** which is a particular case of M2M where the target artefact is represented by text;
- **Text-to-model (T2M):** that operates in the opposite direction as the M2T, generating a model from a textual representation.

Moreover, any of these types of model transformations can be defined as in-place, meaning that source (or one of the sources) and target are represented by the same model; in this case, the transformation provides as output an updated version of (one of) the model(s) in input. Except for the in-place transformations which are by nature endogenous, the other transformations entailed in OptimAll are exogenous meaning that they operate between artefacts expressed using different languages [2]. M2M transformations are implemented with the Operational QVT\(^1\) language, M2T transformations with Xpand\(^2\), and T2M transformations with Java.

IV. THE OptimAll APPROACH

The goal of the OptimAll approach is to provide support to the developer in optimizing the deployment, already at design phase, by iteratively exploiting simulation based on (i) predicted performance-related EFPs, as well as on (ii) actual runtime performance-related EFPs gathered by monitoring

\(^1\)http://www.eclipse.org/mmt/?project=qvto  
\(^2\)http://www.eclipse.org/modeling/m2t/?project=xpand

the execution of automatically generated code. In Fig. 1 the OptimAll approach is depicted; note that the contribution of this paper provides a solution for the steps grouped in the dashed box. The approach is meant to operate at design level starting from a design system model defined in UML describing functional aspects. Deployment is modelled using MARTE and encompasses allocation of (i) software entities to tasks and (ii) tasks to specific processing units. From the design model, an M2M transformation (Fig. 1.a1) is in charge of generating an allocation model from which an M2T transformation (Fig. 1.a2) generates a simulation model to be fed as input to the task allocation optimization mechanism. This mechanism performs an iterative search process, where in each step the simulation model is executed, EFPs are derived.
from the simulation results, and a new allocation candidate is generated for testing in the next iteration (Fig. 1.b). After the cycle finishes, it outputs the best allocation it was able to find (Fig. 1.c). Together with the allocation, the mechanism outputs the corresponding values of the EFPs and a visual trace of how the EFPs changed during the simulation (see Section V).

Simulation and optimization results are meant to be back-propagated (Fig. 1.d) through specific in-place M2M transformations to the design model in a similar way to what was proposed in [16]. Selected EFPs would be shown as extra-functional decorations of the functional model elements they pertain to, while the optimal allocation is meant to be shown in the design model through hints to the user by means of suggested optimal allocation links.

The first run of the task allocation optimization is based on predicted performance-related EFPs (expert estimations) and aims at providing a fair approximation of the actual optimal allocation. In order to reach a solution even closer to the theoretically optimal, OptimAll will have to exploit actual runtime values rather than predictions. In order to do that, the approach will first automatically generate target code from the design model (Fig. 1.e) taking into account the best allocation based on predictions (Fig. 1.d). Such an ability is pivotal in order not to jeopardize the consistency between modelling artefacts, as well as the validity of simulations and optimizations run on them, and the final implementation of the system. In this respect, the generated code is not meant to be edited by hand. Possible optimizations are indeed not performed directly through code editing, but rather by re-iterating the code generation process once the task allocation has been refined according to the optimization mechanism.

Actual runtime values of EFPs to be exploited by the task allocation optimization mechanism are gathered by monitoring the execution of the generated code (Fig. 1.f, 1.g), similarly to what is proposed in [17]. Once gathered, EFPs are back-propagated to the design model (Fig. 1.h), and finally the task allocation optimization mechanism can be re-run leveraging the back-propagated values. The whole approach is iterated until the user is satisfied with the identified allocation.

V. Solution

In this section we describe the various steps of the approach and show how they operate on a running example.

A. Modelling the system

Since in this work we address task allocation to cores, we focus on the modelling artefacts describing the deployment. In OptimAll the deployment of software components to the processing nodes is achieved through two intermediate layers: (1) a software component is allocated through a one-to-one connection to a specific schedulable task (stereotyped as «<swSchedulableResource>»), (2) a task is allocated through a one-to-one connection to a core (stereotyped as «<hwComputingResource>»). An excerpt of the deployment model we will exploit for showing the proposed solution at work is depicted in Fig. 2.

As mentioned in Section III, we support two kinds of tasks: (i) periodic (defined as PeriodicTask in the model), and (ii) event-triggered (defined as EventTriggeredTask). Cores are defined as Core in the model and they have a main scheduler, defined in the stereotype’s property mainScheduler that refers to a scheduler instance (stereotyped as «<scheduler>»). In the example two instances sched0 and sched1 of a fixed priority preemptive scheduler (defined as FPP_scheduled) are shown.

The allocation links are stereotyped as «allocate». In Fig. 2 we can see, e.g., that component c1 is allocated to task t3, which is in turn allocated to core cc0 with scheduler sched0. Additionally, since the task allocation optimization mechanism is based on task chains, we model this information through directed dependency links called chain; in the coming enhancements of the approach we plan to introduce a specific MARTE’s stereotype providing the needed chain-related attributes for this purpose. In Fig. 2 we can see two chains, one constituted of periodic task t3, event-triggered task t4 and event-triggered task t2 and the other one constituted of the sole periodic task t1. In the next increment of OptimAll we plan to automatically derive this information from the
connections among software components in order to relieve
the developer from a manual definition of chains which might
be laborious and error-prone for complex systems.

As aforementioned, the model portion depicted in Fig. 2
does not represent the complete running example which in fact
consists of the following elements:

- **Event-triggered tasks**: \( t_2 = \{1, 2, 5, cc0\} \), \( t_4 = \{2, 4, 4, cc4\} \), \( t_6 = \{3, 6, 3, cc1\} \), \( t_8 = \{4, 6, 2, cc0\} \), \( t_{10} = \{5, 8, 1, cc1\} \);

- **Periodic tasks**: \( t_1 = \{1, 3, 5, cc1, 10, 10\} \), \( t_3 = \{2, 4, 4, cc0, 20, 20\} \), \( t_5 = \{3, 5, 3, cc0, 20, 20\} \), \( t_7 = \{4, 7, 2, cc0, 25, 25\} \), \( t_9 = \{5, 8, 1, cc0, 50, 50\} \);

- **Chains**: \( c_1 = \{t_1\} \), \( c_2 = \{t_3, t_4, t_2\} \), \( c_3 = \{t_5, t_6\} \), \( c_4 = \{t_7, t_8\} \), \( c_5 = \{t_9, t_{10}\} \).

### B. Generating the simulation model

From the system model, the approach automatically ex-
tacts the information needed for running the task allocation
optimization, that is to say tasks, chains, schedulers, cores and
the allocation of tasks to cores. The generation of the simula-
tion model that will be fed as input to the task allocation opti-
mization, is a two-step process. First an M2M transformation
generates an allocation model, conforming to the allocation
metamodel depicted in Fig. 3. From the allocation model,
an M2T transformation generates the actual simulation model,
i.e. the input to the task allocation optimization mechanism.

The reason for such a multi-step approach resides in our
goal to maximise independence of the approach from the
entailed modelling language. In fact, employing the approach
for similar purposes starting from a non-UML system model
would be possible just by redefining the M2M transformation
generating the allocation model, while the rest of the approach
would remain unchanged. Nevertheless, in order to have a full
UML-compliant approach, we aim at bypassing the allocation
model and generate the Java simulation model directly from the
UML–MARTE system model, possibly leaving the allocation
model as an optional generated artefacts.

The allocation metamodel depicted in Fig. 3 represents
the allocation of tasks to cores and it is defined through Ecore
in the Eclipse Modelling Framework\(^3\). The main element is
Configuration which contains tasks, chains, schedulers and
cores. Tasks are represented through the abstract metaclass
Task that defines the common properties of a task. Task
is specialised by (i) EventTriggeredTask, which
represents event-triggered tasks, and (ii) PeriodicTask,
representing periodic tasks. Any Task can trigger a number
of EventTriggeredTask and is allocated to, at most,
one Core. Each Core may have a scheduler (Scheduler) of
type NonpreemptivePriorityScheduler or PreemptivePriorityScheduler. Chains are
represented by the metaclass Chain which points to an
ordered set of Task elements. Once the allocation model\(^4\) is
generated, an M2T transformation takes it as input to generate
the simulation model. The simulation model can be seen as
an executable textual representation of the allocation model.
As the optimization mechanism is implemented using Java,

\(^3\)https://www.eclipse.org/modeling/emf/

\(^4\)Due to space limitation and to its straightforwardness once defined the
allocation metamodel, we do not show the allocation model.

---

![Fig. 3. Allocation metamodel](image319x529 to 571x738)

The simulation model is a Java class that reflects the allocation
model — it defines the existing tasks, their groupings into
chains and their allocation to the available cores.

### C. Task allocation optimization

Task allocation optimization is performed with respect to
to-end-to-end response times. The goal of the optimization is
to keep the number of deadline misses in the system below
a desired boundary, while minimizing the average response
time for a selected task chain. Generally, the optimization
mechanism is envisioned to be extended for other EFPs.

The simulation model serves as input to the task allocation
optimization. In each step the optimization mechanism exe-
cutes the simulation model, and then derives relevant EFPs
from the obtained simulation results. The EFPs are used
in order to quantify the current allocation candidate against
the best allocation candidate found thus far. If the current
allocation is better than the best one, it becomes the new
best allocation. As last step in each iteration, the optimization
mechanism proposes a new allocation candidate to be tested in
the following iteration. This is done using our custom heuristic,
which takes the best allocation candidate as basis, and then
identifies a task to be relocated to a different core. The more a
task delays other tasks and the more it is itself delayed by other
tasks, the bigger the chance it will be picked for relocation.
When choosing a new core for the picked task, core load is
taken into account — the lower the load of a particular core,
the bigger the chance that the picked task will be allocated
there. More details and an evaluation of the heuristic can be
found in [3]. Having relocated a task means that the simulation
model is updated, and the updated version will be assessed in
the next iteration of the optimization mechanism. After having
performed the desired number of iterations, the optimization
mechanism outputs the best allocation candidate it was able to
find.

Next we illustrate one optimization run, using the running
example. The optimization is set to run for 100 iterations. In
In each iteration, the simulation is performed for 1000 time units, which corresponds to 10 hyperperiods in the running example. The chain we are optimizing (i.e., whose average response time is to be minimized) is \( c_2 \). The limit of allowed deadline misses is set to 0. The starting allocation results with no deadline misses for chains \( c_1 \) and \( c_2 \), 1 deadline miss for \( c_3 \), 30 deadline misses for \( c_4 \) and 19 deadline misses for \( c_5 \), so the heuristic will first try to find an allocation with no deadline misses. Task \( t_7 \) is selected for relocation from core \( cc_0 \) to core \( cc_1 \). This new allocation candidate is assessed in the second iteration — \( c_3 \) now has no deadline misses, \( c_4 \) has 11, while \( c_5 \) has 5. Since there are now less deadline misses in the system, this becomes the new best allocation candidate, and the one used as basis for proposing the next candidate. The optimization continues in the same way.

The best allocation is found in step 71: it has no deadline misses and an average response time for chain \( c_2 \) of 7.46. A more detailed illustration and assessment of the optimization mechanism can be found in [3]. Other than the best allocation specification, as mentioned before, the optimization mechanism outputs a visual trace of the relevant performance metrics for any desired allocation. The ones currently supported are: task execution (an excerpt for the best allocation for core \( cc_0 \) is shown in Fig. 4), core load, task deadline misses and chain deadline misses.

VI. OUTLOOK

In this paper we introduced the OptimAll approach for model-driven deployment optimization of multicore embedded soft real-time systems. Besides an overall description of the foundations of the approach, we described in more details the core modelling and prediction-based optimization phases. From a deployment model, we showed how OptimAll automatically generates an input model for the task allocation optimization mechanism and then iteratively runs the simulation in search for a good allocation candidate. In the ongoing work we started to address the steps of the approach that were not covered by this contribution:

- **Automatic code generation**: from system models we aim at generating instrumented code tailored for multicore, which can be executed and monitored to gather the information needed to optimise deployment;
- **Monitoring**: when executing code, we want to be able to observe and gather selected EFPs through specific extensions to the platform (e.g., monitoring routines at OS-level);
- **Back-propagation of EFPs**: the values resulting from model-based simulation described in this work as well as the values gathered at runtime through monitoring shall be propagated back to the system models, in the form of both extra-functional decorations as well as computed textual/graphical allocation hints, for user’s investigation.

Moreover, regarding the phases described in this paper, future enhancements will cover: (i) the automated generation of task chains from message passing and function calls among software components instead of manually modelling them, (ii) the entailment of multi-branch chains meant as the possibility for one task to be triggered by or to trigger several tasks.

ACKNOWLEDGMENT

This research was supported by Swedish Foundation for Strategic Research through grant IIS11-0060 (Ralf project) and by ARTEMIS grant 333053 (CONCERTO project).

REFERENCES