Software Qual J (2018) 26:661–684
DOI 10.1007/s11219-016-9343-5

Architecture optimization: speed or accuracy? both!

Federico Ciccozzi[1] · Juraj Feljan[1] · Jan Carlson[1] · Ivica Crnković[1]

Published online: 16 November 2016

Abstract Embedded systems are becoming more and more complex, thus demanding innovative means to tame their challenging development. Among others, early architecture optimization represents a crucial activity in the development of embedded systems to maximise the usage of their limited resources and to respect their real-time requirements. Typically, architecture optimization seeks good architecture candidates based on model-based analysis. Leveraging abstractions and estimates, this analysis usually produces approximations useful for comparing architecture candidates. Nonetheless, approximations do not provide enough accuracy in estimating crucial extra-functional properties. In this article, we provide an architecture optimization framework that profits from both the speed of model-based predictions and the accuracy of execution-based measurements. Model-based optimization rapidly finds a good architecture candidate, which is refined through optimization based on monitored executions of automatically generated code. Moreover, the framework enables the developer to leverage her optimization experience. More specifically, the developer can use runtime monitoring of generated code execution to manually adjust task allocation at modeling level, and commit the changes without halting execution. In the article, our architecture optimization mechanism is first described from a general point of view and then exploited for optimizing the allocation of software tasks to the processing cores of a multicore embedded system; we target extra-functional properties that can be

✉ Federico Ciccozzi
federico.ciccozzi@mdh.se

Juraj Feljan
juraj.feljan@mdh.se

Jan Carlson
jan.carlson@mdh.se

Ivica Crnković
ivica.crnkovic@mdh.se

[1] School of Innovation, Design and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden

concretely represented and automatically compared for different architectural alternatives (such as memory consumption, energy consumption, or response-time).

Keywords Architecture optimization · Software quality · Model-driven engineering · Model transformations · Back-propagation · Execution · Monitoring · Multicore · Embedded systems

1 Introduction

Currently, embedded systems can be found in most electronic and electrical products. Software running on them is becoming more complex at daunting pace. To tame the intricacy of embedded software and its development, it is profitable to boost abstraction in the form of models which allow to (i) ease the reasoning about the system's architecture, (ii) automate certain stages of the development, and (iii) early analyse and optimise software quality attributes.

Pivotal is to provide means to seek good architecture candidates with respect to quality attributes; architecture optimization is a well-studied problem in the research community (Aleti et al. 2013). Commonly, a model representation of the software system under development is analysed to gather performance predictions. These values are exploited by exploring the search space of architecture candidates, iterating on a set of variable aspects of the architecture.

In order to do so, architecture optimization approaches have historically exploited some of the postulates of Model-Driven Engineering (MDE), where the core concept is represented by the model, considered as an abstraction of the system under development. Rules and constraints for building models are described through a corresponding language definition and, in this respect, a metamodel describes the set of available concepts and wellformedness rules a correct model must conform to Kent (2002). Following the MDE paradigm, a software system is developed by designing models and refining them starting from higher and moving to lower levels of abstraction until code is generated; refinements and analyses based on models are commonly performed through model manipulations.

Architecture optimization based on models is very profitable to efficiently scrutinise a large number of possible architecture candidates and to allow optimization early in the development. However, model-based analysis, which naturally relies on abstractions and estimations, provides approximations that in some cases need a verification at runtime in the form of execution-based measurements. These measurements are gathered by monitoring the execution of automatically generated code. Correctness-by-construction[1] of full-fledged code is essential to disclose the opportunity to rightfully combine model- and execution-based optimization.

In this article, we present a framework that promotes a novel combination of model-based and execution-based mechanisms for achieving fast and accurate architecture optimization. A good architecture candidate is meant to be rapidly reached by prediction-based optimization at modeling level. Afterwards, the identified architecture candidate is refined by carrying on with the same optimization mechanism, but capitalising on measurements

[1] By correctness-by-construction we mean the adherence of generated code to source models, once the transformation process is validated (Chapman 2006). Anyhow, mechanisms for ensuring correctness of models (and thereby code) are not in the scope of this work.

gathered by monitoring. Augmenting optimization with runtime measurements increases the accuracy of the performance metrics used for optimization, with respect to the system properties of interest.

In the article, we describe our architecture optimization mechanism which combines model-based analysis and system execution and demonstrate its usefulness by employing it for optimizing allocation of software tasks to the cores of a multicore embedded system. Since in several domains, the expert's knowledge still plays an important role in optimization activities, we also introduce the infrastructure needed for the developer to manually tune allocation of tasks to cores at modeling level and dynamically apply changes at runtime. This feature introduces observability of the running system at modeling level, and allows the developer to only focus on modeling activities and operate in the modeling environment. The developer can exploit execution monitoring results to fine-tune allocation at modeling level without having to investigate nor manually modify the running code and propagate model modifications to the running system without halting its execution.

The article is organised as follows. In Section 2, we discuss the motivation for enhancing model-based architecture optimization with monitored system execution, while in Section 3, we describe our proposed method. In Section 4, we apply our method to embedded systems; more specifically, we describe how we instantiated the method to create a framework for optimising the allocation of software tasks to the cores of a multicore embedded system. A set of experiments are presented in Section 5, where we demonstrate the feasibility of the approach and the usefulness of combined model- and execution-based optimization. In Section 6, we provide a snapshot of the related work documented in the literature, and we conclude the article with a summary and a set of possible future enhancements in Section 7.

2 Boosting architecture optimization

Among the many activities that characterise the development of software systems, architectural design is commonly considered as one of the most important. Decisions taken when outlining the architecture usually have substantial impact on system properties, both functional and more importantly extra-functional, but even on development-related aspects, such as costs and time-to-market. Classic examples of architectural decisions are the selection of software and hardware components, the allocation of software to hardware, and the configuration of the system topology.

Historically, design decisions have been taken by exploring, often manually, the search space to identify the most suitable choices. Considering the dramatic pace at which complexity of modern software systems is increasing, manual exploration of massive search spaces has become infeasible. This has made automated architecture optimization a prominent research topic (Aleti et al. 2013) that, over the recent years, has considered various system domains (e.g., enterprise systems and embedded systems), various system representations (e.g., mathematical models and architecture description languages), and extra-functional properties (e.g., reliability and timing), with diversified dimensionality (optimization of a single or multiple extra-functional properties), degrees of freedom (e.g., component selection, allocation, and scheduling) and strategies (e.g., local search and genetic algorithms).

In general, architecture optimization mechanisms employ model-based analysis to predict extra-functional properties which are used to compare architecture candidates; analysis is paired with an optimization strategy (i.e., search mechanism). Analysis based on models naturally leverages abstractions of the software system under development and relies on

estimations. Model-based analysis is needed to efficiently handle the large number of possible candidates and to allow optimization in early stages of development, but there is always a limit to how accurate the optimization can be, since model-based analysis inherently relies on abstractions and estimations, and thus gives approximate results. On the other hand, we have the more accurate execution-based optimization which uses performance measurements for comparing different architecture candidates. However, optimization-based purely on runtime measurements is typically too time consuming to be feasible, as each candidate has to be implemented and executed. Even when it is possible to specify candidate-specific information as parameters external to the code, and thus reuse the same code for all candidates, many extra-functional properties are still faster to simulate than to measure during execution.

Exploiting MDE and more specifically model transformations, we can employ design models not only for performance prediction but also for automatic full code generation. A model of the software system can be used to generate the complete implementation, instrumented with specific code for measuring extra-functional properties of interest. This makes it possible to combine model-based and execution-based optimization, and leverage the speed of the former and the accuracy of the latter. It is important to stress the fact that the ability to produce full code from models, when applicable, strengthens the combination of model-based and execution-based optimizations since consistency between models and code is ensured by-construction. In case full code generation cannot be achieved, an alternative way to ensure consistency between models and code shall be chosen in order to benefit from the combined optimization.

Please note that we only target extra-functional properties that can be concretely represented and automatically compared for different architectural alternatives (such as memory consumption, energy consumption, or response-time). The approach is not intended for more abstract properties or those that emerge from the architecture as a whole, such as evolvability or safety. Moreover, the presented approach assumes that test generation and execution can be fully automated, possibly after an initial manual setup. If this is not the case, and manual configuration is required for each test, it would still be possible to benefit from a combination of model-based and execution-based optimization but this scenario is outside the scope of this paper.

3 Fast and accurate architecture optimization through combined model- and execution-based mechanisms

In this section, we formalise our model- and execution-based optimization method as the result of a set of domain-specific research efforts whose previous results are documented in the literature (Feljan et al. 2015; Ciccozzi et al. 2013a; Ciccozzi et al. 2013b; Bucaioni et al. 2015) and which highlight the malleability of the method. The method is depicted in Fig. 1. The targeted user of the framework is the software architect. As part of the architecture design phase, the software architect defines the software and hardware models of the system using the designated modeling language(s). The former defines the software architecture of the system being built, in terms software components and connections among them, while the latter specifies the hardware platform. Additionally, the architect might need to define an initial configuration in terms of, e.g., allocation of software components to processes in a telecom application or timing elements of the software model intended as clocks and triggers on specific software circuits in a vehicular application. This initial configuration is

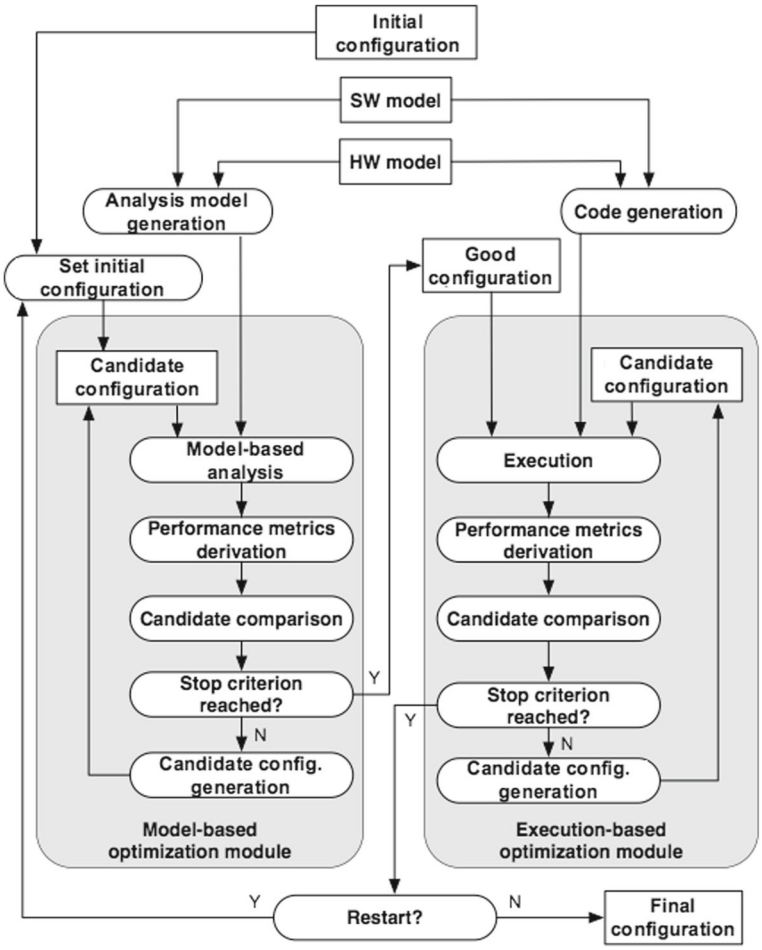


Fig. 1 Combined model- and execution-based optimization

used as starting point for the optimization mechanism. This step is not always needed since, depending on the instantiation scenario, the framework might be able to autonomously generate one or a set of initial configurations. Furthermore, the software architect is asked to specify extra-functional constraints (not visible in Fig. 1). These represent the rules that must not be violated by the candidate configurations considered during the optimization. For instance, in the case of timing analysis for seeking the optimal configuration of clocks and triggers, these rules would be represented by timing constraints to be obeyed by the configurations. Finally, the software architect has to specify the requirements (not visible in Fig. 1) for the extra-functional properties that are to be optimized, or in other words, define the stop criteria for the optimization mechanism. Besides reaching a certain value for a particular extra-functional property, other possible stop criteria can be a predefined number of iterations, a given time limit, a number of consecutive iterations that have not resulted in an improvement, or a combination of the above.

3.1 Automatic generation of analysis models and code

One of the pillars of MDE is the provision of automation in terms of model manipulations and refinement, which is performed through so called model transformations. A model transformation translates a source model to a target model (or text) while preserving their wellformedness (Czarnecki and Helsen 2003). In our approach, we exploit the following kinds of model transformation:

- **Model-to-text (M2T):** which translates source models to target artefacts 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 in-place transformations, which are by nature *endogenous*, the other transformations are *exogenous* meaning that they operate between artefacts expressed using different languages (Czarnecki and Helsen 2003). In our framework, M2T transformations are developed with Xtend (Xtend 2014), and T2M transformations are developed with the Operational QVT[2] language.

The method's workflow starts with the navigation of software and hardware models, and through M2T transformations, it proposes the generation of (i) an analysis model (not needed in some cases) and (ii) instrumented executable code. The former can be represented by the very same prescriptive software and hardware models if processable by the given simulator or a processable version of them which is automatically generated. The latter represents an implementation of the system, instrumented with code to extract the extra-functional properties of interest.[3] The analysis model and the code are used to obtain performance predictions and performance measurements, respectively.

3.2 Performing optimization

Once the needed artefacts have been produced, optimization (depicted by the two grey rounded rectangles in Fig. 1) can be run. The optimization mechanism tries to optimise the particular extra-functional property (e.g., minimisation of memory usage or maximisation of CPU throughput), while not violating extra-functional constraints. The optimization mechanism is run for each of the initial configurations on top of the analysis model.

The two optimization modules display the same structure, as it can be seen in Fig. 1. Each iteration starts with the optimization framework producing a new architecture candidate by applying a small change to the best candidate found thus far. The new candidate is analysed to derive relevant performance metrics, which are compared to the ones derived from the current best candidate; if the new candidate displays better performance metrics, it becomes the new best candidate. This process is carried on until the stop criteria (e.g., given time limit and max number of iterations) are met. The difference between model- and execution-based modules resides in how relevant performance metrics are derived: in

[2] http://www.eclipse.org/mmt/?project=qvto

[3] Note that the overhead related to the instrumentation in terms of execution time shall be negligible in comparison to the system's execution time.

the case of the model-based optimization module, we leverage performance predictions obtained by model-based analysis, while in the case of the execution-based optimization module we profit from performance measurements obtained by executing the generated system code. These complementary ingredients, one based on model-based analysis and the other based on system execution, represent the novelty of the optimization mechanism we propose.

Model-based optimization module The optimization process starts with the model-based module (grey rounded rectangle on the left-hand side of Fig. 1). At each iteration, the analysis model is complemented with information about a particular configuration and, as such, it represents a particular architecture candidate. Upon having analysed the analysis model, from the data obtained, we derive extra-functional property values specific to the instantiation scenario. Examples could be end-to-end delay in case of optimisation of timing configuration of a vehicular application or execution time and memory usage in case of a telecom application. These performance metrics are used to compare different configurations against each other. The current best candidate is used to generate a new candidate to be tested in the next iteration. Since model-based analysis is faster than system execution, we run model-based optimization to quickly converge to a good affinity specification.

Execution-based optimization module The good candidate identified by the model-based module is used as starting point for the continuation of the optimization process, carried out by the execution-based module (grey rounded rectangle on the right-hand side of Fig. 2). Similarly to the previous module, at each iteration, the generated code is complemented with a particular configuration, representing a particular architecture candidate. As aforementioned, executing the system in order to obtain performance measurements is slower than performing model-based analysis. Therefore, execution-based optimization is typically done for fewer iterations compared to model-based optimization. Performance measurements gathered at each iteration are used to compute the concrete performance metrics, which are in turn used to compare the different configurations against each other. Also, this module uses the best candidate to propose a new candidate for the next iteration.

The whole optimization process is restarted for each configuration provided by the software architect as well as for the generated ones. When the optimization mechanism stops, it compares the best configurations identified in each restart and outputs the best one among them as the final configuration. The two optimization modules can either use a common set of candidate comparison criteria and search heuristics or tailored module-specific criteria. The output of the optimization mechanism is represented by the best configuration. Through in-place T2M transformations, the configuration is used to update the initial deployment model for consistency reasons, but even for the developer to be able to scrutinise, and eventually adjust, the proposed configuration.

4 The specific case of task allocation optimization

In this section, we show an instantiation of our combined model- and execution-based optimization method for the domain of embedded multicore systems. By extending our previous works (Feljan et al. 2012; Feljan and Carlson 2014; Feljan et al. 2015), we developed a framework for optimizing the allocation of software modules (in the domain of embedded systems called *tasks*) to the processing cores of the hardware platform. The framework is

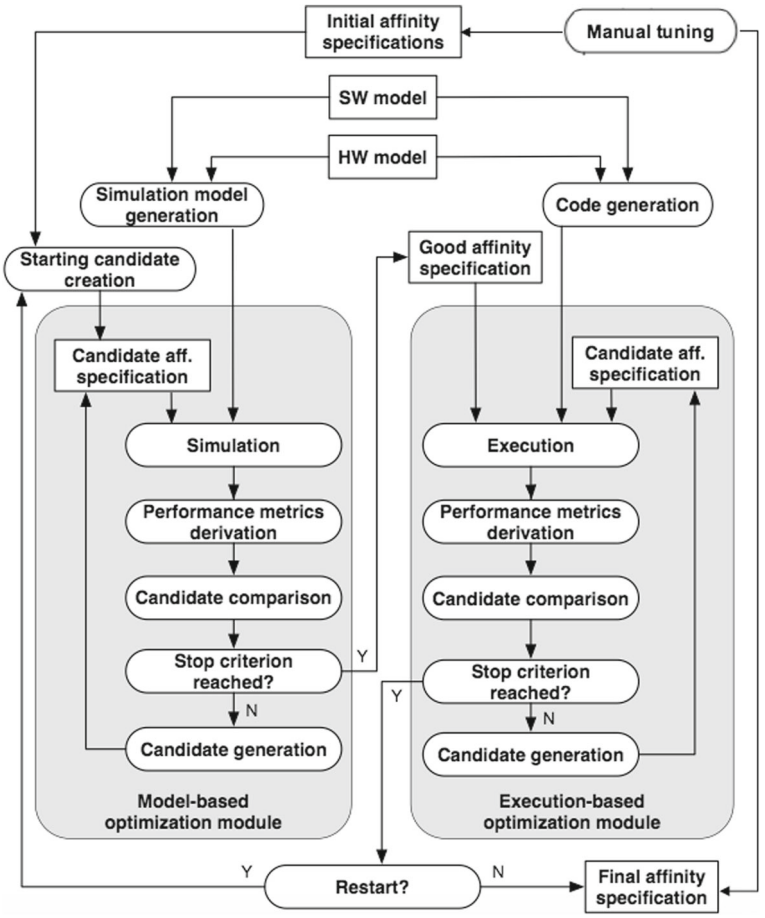


Fig. 2 Framework for task allocation optimization

built on the combination of (i) model-based, (ii) execution-based, and (iii) manual optimization. In the remainder of this section, we contextualise our solution (Section 4.1). Moreover, we describe the specific details of the instantiated model- and execution-based modules as well as their interplay (Section 4.2) and the manual tuning module (Section 4.3).

4.1 Context

Our research work targets architecture optimization for multicore embedded systems. The fact that modern embedded systems are expected to provide an increasing number of complex software functionalities has triggered a drastic increase in performance demands. The mainstream way to tackle this issue has been the introduction of multicore technology in the recent years. The earliest examples of multicore embedded systems are networking (or more specifically packet-processing) systems, such as routers. However, multicore is becoming reality for most domains of embedded systems (e.g., software-defined radio and vision-based driver assistance in modern cars) (Moyer 2013).

On the one hand, increasing the number of processing units does increase the performance power. On the other hand, developing software to run on multiple processing units must face a new challenge: how to allocate (deploy) software tasks to the processing cores available in the hardware platform. Besides obvious functional consequences, allocation can have a significant impact on pivotal extra-functional properties. An intuitive example is schedulability—if too many tasks are allocated to the same core, the core will become overloaded and the tasks will miss their deadlines.

4.1.1 Domain

The domain of our work is represented by embedded soft real-time systems, where accurate timing behavior is crucial for the correct functioning of the system, (a logically correct result that is produced at the wrong time point is equivalent to a logically incorrect result), but occasional deadline misses are tolerated (as opposed to hard real-time systems where the absence of deadline misses must be guaranteed beforehand). Timing plays a crucial role, therefore, the extra-functional properties in our focus are timing-related attributes such as: end-to-end response time (elapsed time between the point when the task starting the chain becomes ready for execution, and the point when the last task in the chain finishes the corresponding execution instance), deadline misses and core load (percentage of core's busy portion). Furthermore, since we consider soft real-time, we focus on average-case behaviours (as opposed to worst-case behaviours typical of hard real-time systems). Since these properties depend on the dynamic interplay between the tasks and there are no analytical methods to statically derive them from task and platform parameters, we use dynamic model-based analysis in form of model simulation.

4.1.2 Modeling and execution environment

The reference modeling language we employed in our approach is represented by UML[4] and its profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) (OMG 2014) for modeling software in terms of tasks, hardware in terms of cores and allocations of tasks to cores. We leverage concepts from the Generic Resource Modeling (GRM) and the Allocation Modeling (Alloc) packages in MARTE. More specifically, we exploit the following concepts (i.e., stereotypes): *"swSchedulableResource"* for modelling software schedulable tasks, *"hwProcessor"* for modeling cores and *"scheduler"* for modeling schedulers available in the platform, and *"allocate"* for stereotyping dependency links for modeling allocation of tasks to cores. The used MARTE stereotypes are accounted by the model transformations to correctly discern among the various modeling elements when navigating the source model for translation to simulation model or executable code. That is to say, MARTE concepts are not directly leveraged by the optimisation mechanism, but they are pivotal for correctly generating input artefacts for the optimisation mechanism.

The approach is implemented and runs on top of the Eclipse Papyrus Project (Gérard et al. 2007), an open source integrated environment for editing EMF (Budinsky et al. 2003) models and particularly supporting UML and related profiles, on the Eclipse platform. Regarding the target language, execution environment, and monitoring features, in the

[4] http://www.omg.org/spec/UML/2.3/.

current version of the approach we provide code generation from UML/MARTE models to multithreaded C to be run on Linux with a real-time kernel.[5]

4.2 Method instantiation for task allocation optimization

Our framework for task allocation optimization is depicted in Fig. 2. As part of the architecture design phase, using UML and the MARTE profile, the software architect defines the software and hardware models of the system. Additionally, the architect can also define a set of deployment models in terms of allocations of tasks to cores, so called initial affinity specifications, to be used as starting points for the optimization mechanism; this can be done by either exploiting the allocation stereotypes provided by MARTE, or simply describing the allocations in a textual format. This step is optional since the framework can autonomously generate the desired number of initial affinity specifications. An affinity specification defines for each task which core it is allocated to. Furthermore, the software architect can specify affinity constraints. These represent more complex affinity rules that must be satisfied for all allocation candidates considered during the optimization. For instance, the software architect can define that a particular task can be allocated only to a subset of the available cores, or that two tasks must be allocated to the same core, or to different cores. This is useful in scenarios where we, for example, want to keep tasks used for diagnostic purposes and tasks that implement the actual functionality separated to different cores. Eventually, the software architect specifies the stop criteria for the optimization mechanism too.

The framework navigates software and hardware models designed using UML and MARTE, and through M2T transformations it generates (i) a simulation model and (ii) instrumented executable code. The former is defined in terms of Java objects and represents an executable model of the software system to be fed as input to our task simulator. The latter represents an implementation of the system in C, instrumented with code to extract the extra-functional properties of interest.[6] Regarding code generation, tasks are transformed through a 1-to-1 mapping to POSIX (Opengroup 2013) threads. Depending on the task type, specific triggering code (according to the modeled period for periodic tasks, at triggering task completion for event-triggered tasks) is generated too.

The optimization mechanism tries to minimise end-to-end response times for a particular task chain, while keeping the overall number of deadline misses in the system below a desired limit; both task chains to be entailed and maximum deadline misses are parameters that can be customised by the software architect. Task allocation is the supported degree of freedom, i.e., the aspect of the system that the optimization mechanism is allowed to change. As task allocation is a bin-packing-like problem, which is NP-hard (Johnson 1973), rather than finding the optimal solution, the goal of our framework is to find a good solution quickly. We have therefore opted for a fairly simple optimization strategy, namely local search paired with a domain-specific heuristic (Feljan and Carlson 2014). The optimization mechanism is run for each of the initial affinity specifications, and in order to avoid local optima, it is beneficial to have a large number of initial affinity specifications (some of which can be provided explicitly by the software architect, while the rest can be randomly generated, as mentioned above).

[5] https://rt.wiki.kernel.org/

[6] The overhead related to the instrumentation in terms of execution time is negligible in comparison to the system's execution time.

Model-based optimization module The model-based module is run as first optimization step (grey rounded rectangle on the left-hand side of Fig. 2). At each optimization iteration, the simulation model is complemented with a particular affinity specification and as such represents a particular architecture candidate. After the execution of the simulation model, we derive average end-to-end response times and deadline misses for task chains, and information about core load. This performance information is derived from the data obtained by the simulation and is used to compare different affinity specifications against each other. The current best candidate is used to generate a new candidate to be tested in the next optimization round. As aforementioned, since model simulation is faster than system execution, we run model-based optimization to quickly converge to a good affinity specification.

Execution-based optimization module The model-based module has now identified a good candidate, which is used as starting point for the execution-based optimization module (grey rounded rectangle on the right-hand side of Fig. 2). At each iteration, the generated code is complemented with a particular affinity specification, and together, they represent a specific architecture candidate. Also, in this case, the module uses the best candidate to propose a new candidate for the next iteration.

The whole optimization process is restarted for each initial affinity specification provided by the software architect as well as for a desired number of randomly generated starting allocations. When the optimization mechanism stops, it compares the best affinity specifications identified in each restart, and outputs the best one among them as the final affinity specification. The output of the optimization mechanism is represented by the best affinity specification. Through in-place T2M transformations, the affinity specification is used to update the initial deployment model for consistency reasons and for the developer to inspect, and eventually tune, the proposed affinity specification as described in the next section.

4.3 Exploiting developer's experience for further task allocation tuning

Although automatic mechanisms are pivotal for effectively investigating large search spaces, expert knowledge can still play an important role in optimization activities. That is why we provide the mechanisms needed for the developer to manually tune allocation of tasks to cores at modeling level and dynamically apply changes at runtime (*Manual tuning* in Fig. 2). The developer can exploit execution monitoring results to fine-tune allocation at modeling level without having to investigate nor manually modify the running code, and propagate model modifications to the running system without halting its execution. Manual tuning is meant to be performed either at the beginning of the optimization process, for defining appropriate starting points to be employed by the automatic optimization mechanism, and/or when automatic optimization is completed, to assess and possibly further tune the proposed affinity specification.

The support for manual tuning is depicted in Fig. 3. From the design model (*A* in Fig. 4), we automatically generate an executable multithreaded C application (*C exec* in Fig. 3), which, without any manual adjustment, can be run directly from the modeling framework. When the execution of the generated application is triggered by the developer (*exec* in Fig. 3), and more specifically when threads are initialised, an *identity* file (part of *Allocation files* in Fig. 3) is generated for keeping track of the current thread unique identifiers. During execution, information about single tasks (e.g., core load) are monitored (*monitoring* in Fig. 3) through the interactive process viewer for Linux (htop) and shown to the developer in the modeling framework itself (*return* in Fig. 3). This is achieved by exploiting the *Terminal*

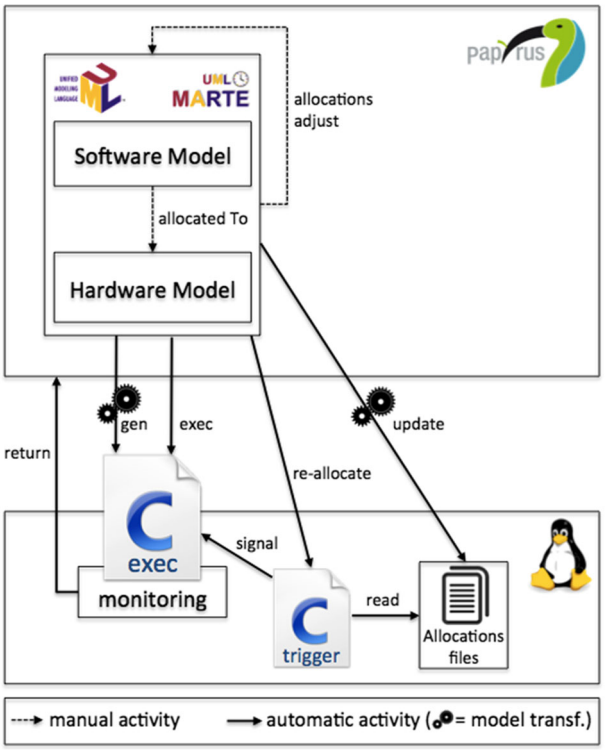


Fig. 3 Support for manual optimization

view in Eclipse and re-directing the Linux command line directly to an integrated window in the modeling environment (*C* in Fig. 4).

Specific automatic mechanisms for improving tasks allocation might have already been run, and the developer can now decide to manually tune those allocations (*allocations adjust* in Fig. 3). When the model is updated, the developer can commit her changes which are automatically propagated to the running system as follows. Firstly, a M2T transformation defined in Xtend navigates the design model and generates (or updates in case it already exists, *update* in Fig. 3) an *allocation* file (part of *Allocation files* in Fig. 3) that contains the updated allocations of tasks to cores. At this point, a specific trigger defined in C (*C trigger* in Fig. 3) is automatically called (*re-allocate* in Fig. 3). The trigger accesses identity and allocation files and uses that information for matching thread identifiers and thereby setting thread affinities (through `sched_setaffinity`, API available for POSIX threads) as specified by the developer in the design. This feature is meant to ease manual tuning by providing automatic means to back-propagate runtime values to the modeling level and thereby to propagate model changes to the running application without halting its execution. All the steps, except for the modeling activities, are fully automated and integrated in the Papyrus environment in terms of plugins. For each of the automatic activities which can be triggered by the developer, namely code generation, code execution, and re-allocation, we defined specific actions, exposed by the plugins. These actions can be accessed directly from the Eclipse project explorer by right-clicking on a Papyrus design model (*B* in Fig. 4).

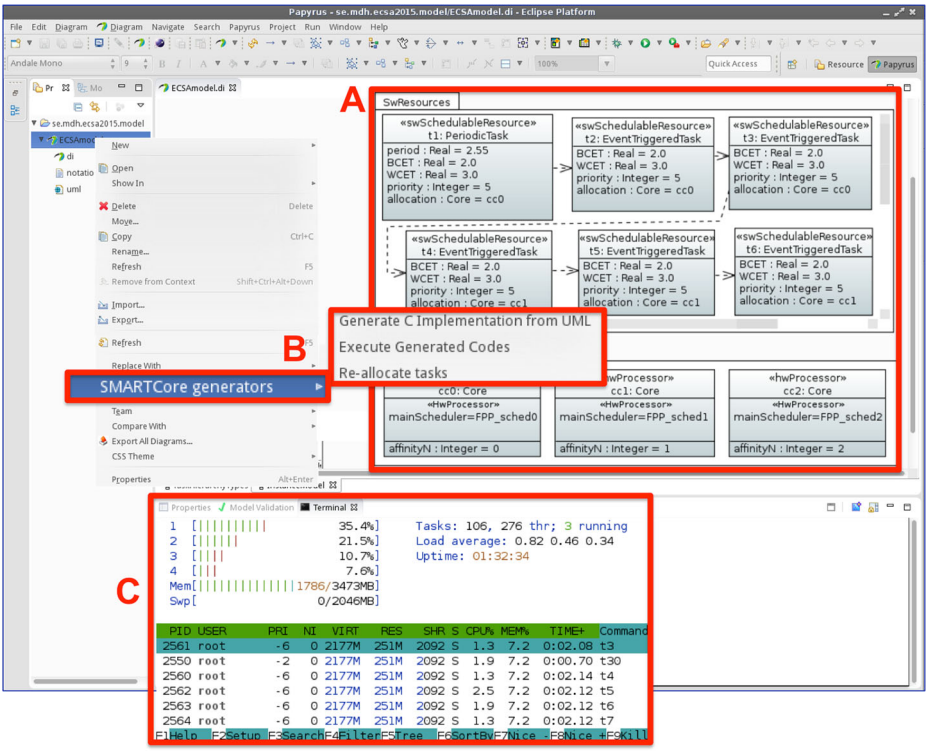


Fig. 4 Support for dynamic re-allocation in the modeling environment

5 Experiment

In this section, we present an experiment performed using our framework to demonstrate the feasibility of the combined model-based and execution-based architecture optimization approach. We start by describing the experiment setup, and then present and discuss the results. Since manual tuning is not a compulsory phase of the automated optimization mechanism, but rather a useful accessory to it, we deliberately did not include it in the experiment. Nevertheless, we used it to check the outcomes of the experiment by monitoring the execution of the final code and trying out alternative allocations.

5.1 Experiment setup

The software and hardware models of the experiment system are shown in Fig. 5. We aimed for a system that contains task chains of varying length. The optimization goal was to minimise the average end-to-end response time for the task chain consisting of tasks *t1* to *t10*. The execution platform had two cores, each running a preemptive priority-based scheduler.

The experiment consisted of four parts: pure model-based optimization, pure execution-based optimization, and two runs of combined optimization with different settings. In each experiment part, we repeated 100 optimization runs, in order to be able to generalize the results. All optimization runs started from the same initial affinity specification with an equal number of tasks allocated to each core: tasks *t1* to *t3* allocated to core 0, tasks *t4* to *t6*

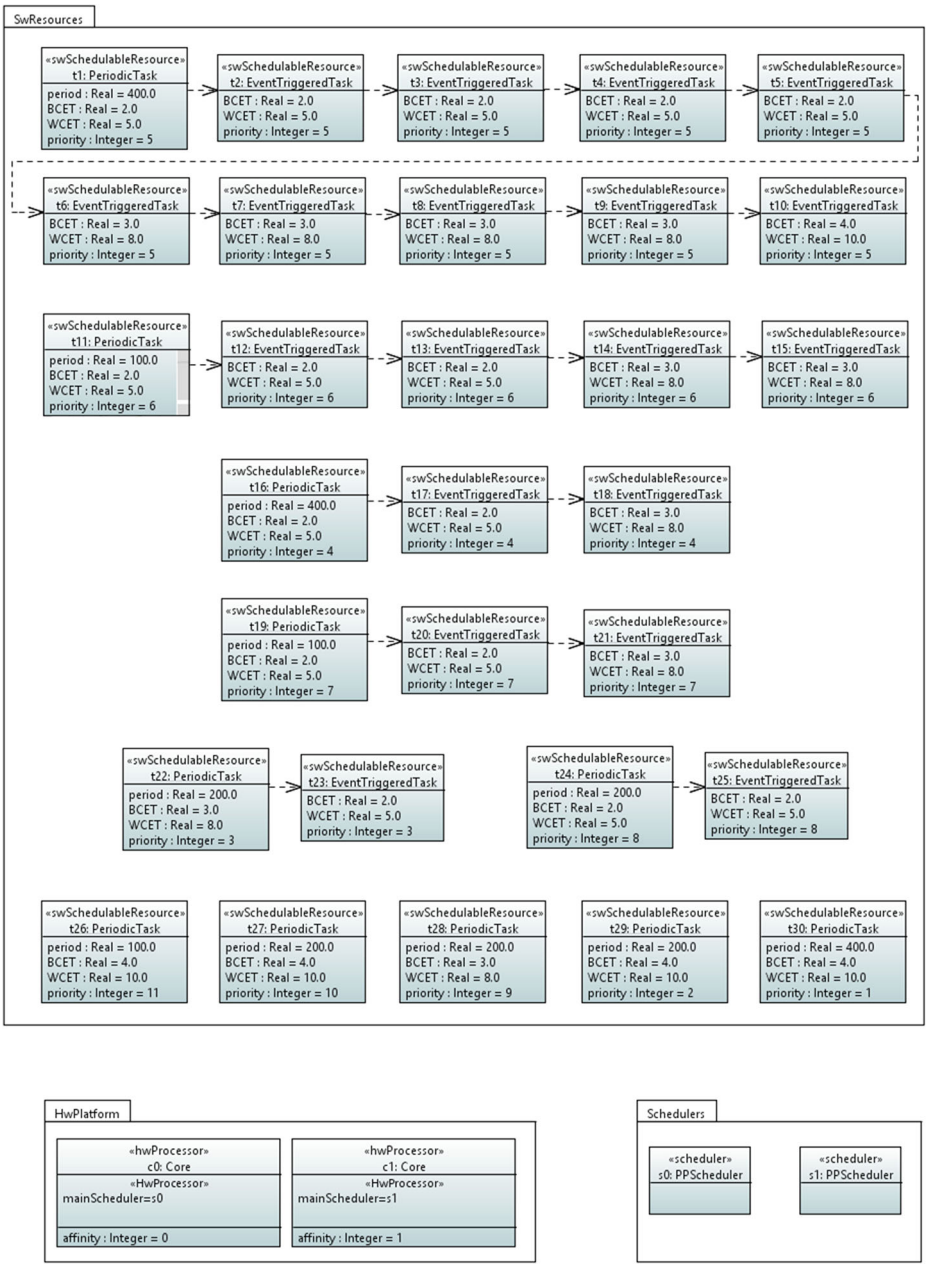


Fig. 5 Software and hardware models of the experiment system

allocated to core 1, tasks *t7* to *t9* to core 0, and so on. The following stop criteria were used for the different parts of the experiment:

1. Model-based optimization — 1250 optimization iterations,
2. Execution-based optimization — 270 optimization iterations,

3. Combined optimization 1 — 1000 model-based optimization iterations followed by 50 execution-based optimization iterations,
4. Combined optimization 2 — 350 model-based optimization iterations followed by 200 execution-based optimization iterations.

These numbers were chosen so that all four parts of the experiment take roughly the same time, allowing for more straightforward comparison of the end results, since each execution took about 4.5 times longer than each simulation.

Each candidate simulation in the model-based optimization was performed for 4000 simulation steps (clock ticks), while the execution-based optimization monitored the resulting system during 4 s. This was chosen in order to (i) run the system long enough (10 hyperperiods) to capture the average behaviour and (ii) result in a similar number of activations of the optimised chain during simulation and execution.

In all optimization runs, for proposing the candidate to be tested in the next iteration, we used a simple heuristic that randomly relocated one task to a different core. The heuristic was deliberately kept simple, compared to the heuristic we have proposed (Feljan and Carlson 2014), in order for the experiment to focus on the core novelty of the approach rather than on the heuristic.

Before running the optimization, from the models defined in UML and MARTE, the framework automatically generated:

1. a simulation model in Java, to be fed as input to the model-based optimization module, and
2. an instrumented implementation in C, to be used by the execution-based optimization module.

Simulation model and instrumented C are generated by M2T transformations, and no additional manual activity is needed in order for them to be simulated and executed, respectively.

In the implementation, each task was represented by a POSIX (Opengroup 2013) thread with read-execute-write semantics—it first reads input data, then performs calculations and finally writes output data. Since the input model did not contain a specification of task behaviour, task calculations were represented by a loop that repeats a simple addition operation. The number of loop iterations were selected for each task instance to result in random execution times within the ranges defined in the software model. The implementation was also instrumented with code for measuring the end-to-end response times for a particular chain.

The hardware and software environment consisted of an Intel Core 2 Duo E6700 processor (Intel 2014), running the 32-bit version of the Ubuntu 12.04 LTS operating system (kernel version 3.2.29). The operating system was patched with the PREEMPT RT patch (version 3.2.29-rt44) (RT-linux 2014), which turns the stock Linux kernel into a hard real-time kernel. By reducing the overall jitter and enabling the tasks to run at the highest levels of priority, and in combination with a high resolution timer, this setup reduces unwanted interference in the experiments and increases the accuracy of the measurements.

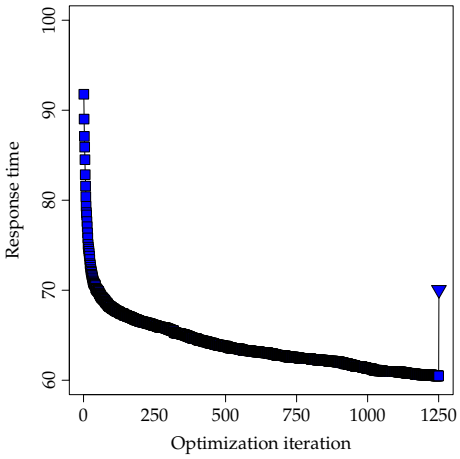
5.2 Experiment results

Figure 6 shows the results of the experiment. Each point in the diagrams shows the selected chain's end-to-end response time for the best affinity specification found after a particular number of optimization iterations, as an average of 100 optimization runs.

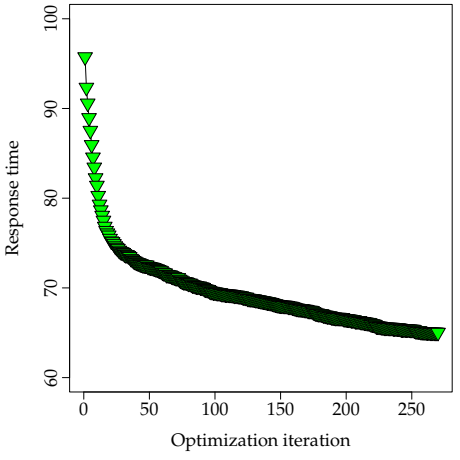
Model-based optimization iterations are depicted with squares, while execution-based optimization iterations are depicted with triangles.

Note that there is a significant jump in the response time value when changing from model-based to execution-based optimization (at iteration 1001 in Fig. 6c and iteration 351 in Fig. 6d). This is not a deterioration or a step back in the optimization, but rather depends on the limited accuracy of the model-based performance prediction, and could in the general case be a negative as well as a positive jump. This is also the reason why we added a single execution-based measurement after the final iteration in Fig. 6a, in order to make all final results measurements which allows for a fair comparison.

In Fig. 7, we can see all four experiment parts plotted against time rather than optimization iteration. The colors are used to make it easier to identify which line in the figure corresponds to which subfigure of Fig. 6.

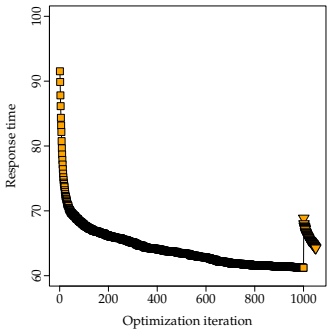


(a) Model-based optimization

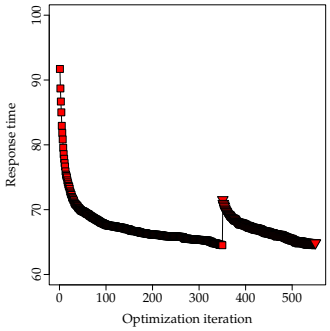


(b) Execution-based optimization

Fig. 6 Experiment results showing the selected chain's end-to-end response time for the best affinity specification; model-based optimization iterations are depicted with *squares*, while execution-based optimization iterations are depicted with *triangles*



(c) Combined model-based and execution-based optimization (1000 + 50 iterations)



(d) Combined model-based and execution-based optimization (350 + 200 iterations)

Fig. 6 (continued)

We can see that in this experiment, the model-based optimization converges towards a solution that is not, in fact, optimal in reality. For example, if we compare the first measured value after switching from the model-based optimization module in the blue and red experiments, the blue value at roughly 27 min is not much better than the red value at

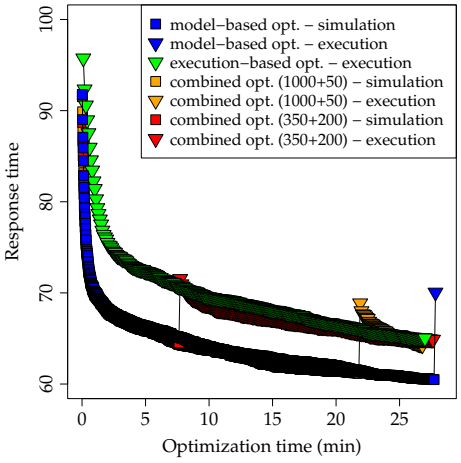


Fig. 7 Experiment results over time showing all four experiment parts plotted against time

roughly 8 min, despite the fact that the blue curve showed a steady improvement throughout the experiment. The difference in what simulation and execution consider to be a good allocation also means that a candidate which cannot be further improved by model-based optimization still has quite a lot of optimization potential when switching to execution-based optimization. This can be seen quite clearly in the two combined optimizations, where the allocation improves rapidly just after the switch. The generated tasks in this experiment have a uniform execution time distribution, and with task code generated from detailed functional specifications, the difference between model-based analysis and simulation would be even bigger, thus further increasing the motivation for the execution-based optimization module. The model-based optimization converges faster in the beginning, although the difference in this particular experiment is rather small since the difference in time between measurement and model simulation is only a factor 4.5. This small difference also meant that 270 iterations were sufficient to reach a good candidate also with pure execution-based optimization. However, if the time available for the optimization was shorter, e.g., 10 min, the combined approach (in this case the red variant) finds a better allocation than pure execution-based optimization.

An important aspect of the combined optimization method is thus choosing the point when to perform the change from the model-based to the execution-based optimization module. Providing a general rule that works for all approaches is not possible, since this depends on how well model-based analysis and execution agree on what is a good allocation, and on the difference in duration between obtaining performance predictions and performance measurements. For instance, if performance prediction is imprecise, the model-based optimization module could guide the search in a slightly wrong direction before handing over to the execution-based module. Also, the smaller the difference in duration between model-based analysis and execution, the smaller the motivation for running the model-based optimization module for a large number of iterations. On the other hand, the closer that model-based analysis comes to execution in terms of accuracy, the better it is to run many iterations of the model-based module before switching to the execution-based one. The same is valid when model-based analysis is much faster than execution. For instance, when performance predictions can be obtained analytically (without simulation), the difference in duration between model-based analysis and execution is bigger than in the experiment, meaning that the model-based module could process a much larger part of the search space than the execution-based module in the same amount of time.

It is also worth pointing out that this experiment presents the average result of 100 optimization runs with a fixed number of iterations and no restarts. With respect to improving individual optimization runs, a promising alternative would be to switch from model-based to execution-based optimization earlier than after the fixed number of iterations if we get stuck in a local optimum (a number of consecutive iterations without an improvement), in order to have time for more restarts.

6 Related work

The literature is rich of different software architecture optimization methods that focus on various aspects. A systematic literature review (Aleti et al. 2013) provides an analysis of the solutions provided by the different research groups. In addition, it provides a taxonomy aiming at classifying existing approaches in order to establish a common reference for the many problem variations as well as solutions within software architecture optimization.

In the embedded domain, the natural dependency between software architecture and quality has been the focal point of numerous research works (Sharma et al. 2005; Bondarev et al. 2005). Especially, the evaluation of the different quality concerns of a system in relation to its architectural aspects, such as a specific deployment configuration, is of paramount importance and it should be performed as early as at modeling level. There are a number of model-driven approaches that provide ways to model specific quality attributes and push them to the software level (Islam et al. 2006; Grunske et al. 2007). Zhu et al. (2012) present model-driven mechanisms for the optimization of task allocation, signal to message mapping, and assignment of priorities to tasks and messages in order to meet end-to-end deadline constraints and minimise latencies.

In the scope of deployment optimization, techniques have been defined to automatically explore the space of deployment options to identify the near optimal candidate. A relevant body of these solutions aim at only satisfying predefined constraints (Martens and Koziolek 2009), while the rest seeks a near optimal deployment candidate without violating a set of given constraints (Medvidovic and Malek 2007; Fredriksson et al. 2005). Model-driven approaches usually provide good approximations, which are although often not enough for embedded systems where runtime measurements are needed in order to assess certain critical performance-related quality attributes. A systematic review (Koziolek 2010) reports on the approaches dealing with optimization based on measurements at system implementation level that can be found in the literature. The COMPAS framework by Mos and Murphy (2002) is a performance monitoring approach for J2EE systems. For performance prediction of the modeled scenarios, the approach suggests using existing simulation techniques, which are not part of the approach. Based on the COMPAS framework, two further approaches have been proposed: AQUA, by Diaconescu and Murphy (2005), and PAD, by Parsons and Murphy (2008).

The common characteristic of these approaches is that they provide optimization at one specific abstraction level. That is to say, they either provide optimization based on the modelled architecture (at model level) or to identify performance issues in the running system and adapt the corresponding code to make it able to fulfil specified constraints (at code level). In fact, to the best of our knowledge, there are no documented optimization methods that combine performance prediction and performance measurement, regardless of the system domain. Sailer et al. (2013) come close in their approach for allocating tasks to multicore electronic control units (ECUs) in the automotive domain. However, rather than combining simulation- and execution-based optimization, they run only simulation-based optimization which uses a system description in AUTOSAR and runtime measurements of the related runnables in hardware traces as input. A genetic algorithm is then used to create and evaluate solutions to the task allocation problem. While leveraging runtime values for generating simulation models, the approach does not provide any execution-based optimization mechanism. The uniqueness of our approach consists in fact in providing a software architecture optimization mechanism that incrementally leverages model-based predictions and runtime measurements gathered at system implementation level. Moreover, automatic optimization can be aided by manual intervention of the experienced developer.

On the one hand, when measurements at system implementation level are considered, besides runtime monitoring other code level verification techniques (e.g., static analysis) not based on execution can be employed, even though their application for large and complex systems may not always be practically and economically favorable (Wall et al. 2003). In fact, conditions that may cause invalidation of the analysis results at runtime can occur. A typical example of this is the differences between the ideal execution environment (considered

for performing analysis) and the actual one which can lead to the violation of the assumptions taken into account when performing static analysis (Chodrow et al. 1991). On the other hand, the information gathered through monitoring system execution is useful for (i) observing the actual system's behavior and to detect violations at runtime, and for (ii) making adaptation decisions. For instance, in Saadatmand et al. (2012), the authors use monitoring information for balancing timing and security properties in embedded real-time systems. Huselius and Andersson (2005) describe a method for the generation of design models of embedded real-time systems from the monitoring information gathered at runtime. In our framework, we profit from monitoring results from which observed values for selected system properties are computed and used to improve the architecture.

7 Conclusion and future work

Modern embedded systems become more and more advanced and complex, thus exhibiting increasing performance demands. Over the recent years, we have seen that multicore technology, previously successfully used in general-purpose computer systems, made an entry into the embedded domain. While on the one side, it does provide higher performance capabilities; on the other side, it introduces a problem that was not present with singlecore processors—how to best allocate software tasks to the cores of the hardware platform to achieve satisfactory performance.

Since model-based analysis leverages abstractions and approximations to provide performance predictions, architecture optimization based only on this kind of analysis has limited accuracy. Nevertheless, one of the core facilities typical of model-driven approaches, namely automatic full-fledged code generation, can be leveraged to generate the complete system implementation instrumented with code for extracting performance measurements for the extra-functional properties of interest. Doing so, model-based optimization can be combined with execution-based optimization that uses performance measurements for candidate comparison.

In this research work, we presented a novel method for combined model-based and execution-based architecture optimization. The method relies on model-based optimization to quickly converge to a good architecture candidate, which is then used as the starting point for the slower but more accurate execution-based optimization. We instantiated the method for the specific problem of optimizing task allocation based on response time and we carried out an experiment that, although limited in size and scope, demonstrated the feasibility and value of the approach. The experiment showed that, for the used system, combined optimization found on average a slightly better solution than both pure execution-based and pure model-based optimization. It was performed optimizing only chain end-to-end response time using a simple random heuristic. Support for enabling the combination of several optimization factors, such as chain end-to-end response times and deadline misses, could be a notable improvement. Furthermore, an interesting experiment would be to use separate heuristics for the two optimization modules, each specially tailored to the respective optimization module. Moreover, to assess the reusability of our approach, we would like to apply the idea of combined model- and execution-based optimization in a framework that is significantly different from ours (for instance, one that does not employ model simulation, but rather obtains performance predictions analytically).

In many domains, the expert's knowledge is an important factor in optimization activities. To allow the expert developer to leverage her knowledge, we provide an infrastructure for her to manually tune allocation of tasks to cores at modeling level and dynamically

apply changes at runtime. This capability gives the developer the chance to focus on modeling activities and operate in the modeling environment only. Execution monitoring results are used by the developer to (i) fine-tune allocation at modeling level without having to investigate the code and (ii) propagate model modifications to the running system without halting its execution. These manual activities can be useful at two different stages: (i) at the beginning of the optimization process, for defining smart starting points for the automatic optimization mechanism and (ii) when automatic optimization is completed, to assess and possibly further tune the architecture.

Concerning the support for manual tuning of task allocation, in this work, we do not provide any specific monitoring feature, but rather rely on the information gathered by *htop*, such as core load (even per thread). Even by just monitoring this kind of information, a change in the task allocation propagated from the design model can be noticed in terms of core load change. On the one hand, this is enough to appreciate the possibility for the developer to only focus on modeling activities and operate solely in the modeling environment, without interacting directly with code and platform, but still affecting them. On the other hand, a broader set of extra-functional properties need to be taken into account for meaningful model optimizations. In this sense, we aim at including monitoring of properties such as average end-to-end response times, deadline misses, and more detailed information about core load as in Ciccozzi and Feljan (2014). Moreover, since re-allocation of tasks at runtime can create unexpected states in the running system, we aim at investigating these situations too. While in this solution monitoring is reported to the developer in its original format (in command line style), the next step would be to enable back-propagation of the values to the model elements themselves through specific in-place model-to-model transformations similarly to what is shown in Ciccozzi et al. (2013b). This information will be shown as extra-functional decorations of model elements to enhance understandability of values in direct relation to the model.

Acknowledgments This work was supported by the Swedish Foundation for Strategic Research via the Ralf 3 project (IIS11-0060) and by the Knowledge Foundation through the projects SMARTCore (20140051) and ORION (20140218).

References

Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., & Meedeniya, I. (2013). Software architecture optimization methods: a systematic literature review. IEEE Transactions on Software Engineering.

Bondarev, E., de With, P., Chaudron, M., & Muskens, J. (2005). Modelling of input-parameter dependency for performance predictions of component-based embedded systems. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pages 36–43. IEEE.*

Bucaioni, A., Cicchetti, A., Ciccozzi, F., Eramo, R., Mubeen, S., & Sjödin, M. (2015). Anticipating implementation-level timing analysis for driving design-level decisions in east-adl. In *International Workshop on Modelling in Automotive Software Engineering*.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., & Grose, T.J. (2003). Eclipse Modeling Framework. Addison Wesley.

Chapman, R. (2006). Correctness by construction: a manifesto for high integrity software. In *Procs of SCS*.

Chodrow, S.E., Jahanian, F., & Donner, M. (1991). Run-time monitoring of real-time systems. In *Real-time systems symposuim (RTSS)*.

Ciccozzi, F., & Feljan, J. (2014). Model-driven deployment optimization for multicore embedded real-time systems: the optimall approach. In *Procs of WATERS*. http://www.es.mdh.se/publications/3621-.
Ciccozzi, F., Cicchetti, A., & Sjödin, M. (2013a). Round-trip support for extra-functional property management in model-driven engineering of embedded systems Information. *Information & Software Technology*, *55*(6), 1085–1100.
Ciccozzi, F., Saadatmand, M., Cicchetti, A., & Sjödin, M. (2013b). An automated round-trip support towards deployment assessment in component-based embedded systems. In *Procs of CBSE*.
Czarnecki, K., & Helsen, S. (2003). Classification of Model Transformation Approaches. In *Procs of OOPSLA*.
Diaconescu, A., & Murphy, J. (2005). Automating the performance management of component-based enterprise systems through the use of redundancy. In *20th IEEE/ACM international Conference on Automated Software Engineering (ASE). ACM*.
Feljan, J., & Carlson, J. (2014). Task allocation optimization for multicore embedded systems. In *40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*.
Feljan, J., Carlson, J., & Seceleanu, T. (2012). Towards a model-based approach for allocating tasks to multicore processors. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 117–124).
Feljan, J., Ciccozzi, F., Carlson, J., & Crnkovic, I. (2015). Enhancing model-based architecture optimization with monitored system runs. In *41st Euromicro Conference on Software Engineering and Advanced Applications*. http://www.es.mdh.se/publications/3926-. copyright IEEE.
Fredriksson, J., Sandström, K., & Åkerholm, M. (2005). Optimizing resource usage in component-based real-time systems. In *International Symposium on Component-Based Software Engineering (CBSE)* (pp. 49–65): Springer.
Gérard, S., Dumoulin, C., Tessier, P., & Selic, B. (2007). Papyrus A UML2 Tool for Domain-Specific Language Modeling. In *Model-Based Engineering of Embedded Real-Time Systems* (pp. 361–368).
Grunske, L., Lindsay, P., Bondarev, E., Papadopoulos, Y., & Parker, D. (2007). An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems. In *Architecting dependable systems IV* (pp. 188–209): Springer.
Huselius, J., & Andersson, J. (2005). Model Synthesis for Real-Time Systems. In *Ninth European Conference on Software Maintenance and Reengineering (CSMR)*.
Intel (2014). Intel Core 2 Duo E6700 processor. http://ark.intel.com/products/27251/Intel-Core2-Duo-Processor-E6700-4M-Cache-2_66-GHz-1066-MHz-FSB, [Accessed: 2014-12-19].
Islam, S., Lindstrom, R., & Suri, N. (2006). Dependability driven integration of mixed criticality SW components. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC). IEEE*.
Johnson, D.S. (1973). Near-optimal bin packing algorithms. PhD thesis, Massachusetts Institute of Technology, Department of Mathematics.
Kent, S. (2002). Model Driven Engineering. In *Third International Conference on Integrated Formal Methods (iFM)*.
Koziolek, H. (2010). Performance evaluation of component-based software systems: A survey, Performance Evaluation, Special Issue on Software and Performance.
Martens, A., & Koziolek, H. (2009). Automatic, model-based software performance improvement for component-based software designs. *Electronic Notes in Theoretical Computer Science*, *253*(1), 77–93.
Medvidovic, N., & Malek, S. (2007). Software deployment architecture and quality-of-service in pervasive environments. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE* (pp. 47–51).
Mos, A., & Murphy, J. (2002). A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *Third International Workshop on Software and Performance (WOSP)*.
Moyer, B. (2013). Real World Multicore Embedded Systems. Newnes. ISBN 0124160182, 9780124160187.
OMG (2014). The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. http://www.omgmarte.org/, [Accessed: 2014-11-28].
Opengroup (2013). POSIX. http://pubs.opengroup.org/onlinepubs/9699919799, [Accessed: 2013-08-20].
Parsons, T., & Murphy, J. (2008). Detecting Performance Antipatterns in Component Based Enterprise Systems Journal of Object Technology.
RT-linux (2014). PREEMPT RT patch, 2014. https://rt.wiki.kernel.org/index.php/Main_Page, [Accessed: 2014-12-19].
Saadatmand, M., Cicchetti, A., & Sjödin, M. (2012). Design of adaptive security mechanisms for real-time embedded systems. In *4th International conference on Engineering Secure Software and Systems (ESSoS)*.
Sailer, A., Schmidhuber, S., Deubzer, M., Alfranseder, M., Mucha, M., & Mottok, J. (2013). Optimizing the task allocation step for multi-core processors within AUTOSAR. In *International Conference on Applied Electronics (AE)* (pp. 1–6).

Sharma, V.S., Jalote, P., & Trivedi, K.S. (2005). Evaluating performance attributes of layered software architecture. In *International Symposium on Component-Based Software Engineering (CBSE)* (pp. 66–81): Springer.

Wall, A., Kraft, J., Neander, J., Norström, C., & Lembke, M. (2003). Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems. In *9th IEEE International Conference on Embedded and RealTime Computing Systems and Application (RTCSA). Springer Berlin Heidelberg*.

Xtend (2014). Xtend programming language, 2014. http://www.eclipse.org/xtend/documentation.html, [Accessed: 2014-11-28].

Zhu, Q., Zeng, H., Zheng, W., Di Natale, M., & Sangiovanni-Vincentelli, A. (2012). Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)*, *11*(4), 85.

Federico Ciccozzi is Assistant Professor at Mälardalen University, Sweden, School of Innovation, Design and Engineering where he received his PhD degree in 2014. His research focuses on the definition of metamodels and model transformations for several automation aspects in the model-driven development of component-based embedded real-time systems, such as code generation, preservation of system properties, back-propagation, to mention a few. Moreover, he carries out research in the area of multi-paradigm modelling, model versioning, (co)evolution and synchronisation, as well as application of model-driven and componentbased techniques to (multi-)robot systems. He has co-authored over 40 publications in journals and international conferences and workshops in these areas. In his research activity he has collaborated with several companies and research institutions such as Ericsson, ABB, Alten, Thales, CEA list, etc. He is a member of the IEEE. More information is available at http://www.es.mdh.se/staff/266-Federico_Ciccozzi.

Juraj Feljan received his Ph.D. in Computer Science in 2015 from Mälardalen University, Sweden and his M.Sc. degree in Software Engineering in 2008 from the University of Zagreb, Croatia. His main research interests are software architecture and model-driven engineering, primarily for the domain of multicore embedded real-time systems.

Jan Carlson is an associate professor at Mälardalen University, Sweden. He received his M.Sc. degree in Computer Science from Linköping University in 2000, and his doctoral degree from Mälardalen University in 2007. His research interests include component- and model-based development of embedded systems, safety contracts and model-level timing analysis.

Ivica Crnković received the MSc degree in 1981 in computer science, the MSc degree in theoretical physics in 1984, and the PhD degree in 1991 in computer science, all from the University of Zagreb, Croatia. After 15 years of work in industry, he moved to academia 1999. He is professor of software engineering at Chalmers University and Mälardalen University, Sweden, and professornat the Faculty of Electrical Engineering, University of Osijek, Croatia. He is a coauthor of two books, and the coauthor of more than 100 refereed publications on software engineering topics. His research interests include componentbased software engineering, software architecture, software configuration management, software development environments and tools, and software engineering in general. He is a member of the IEEE.