Enhancing model-based architecture optimization with monitored system runs

Juraj Feljan, Federico Ciccozzi, Jan Carlson and Ivica Crnković
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
E-mail: {juraj.feljan, federico.ciccozzi, jan.carlson, ivica.crnkovic}@mdh.se

***Abstract*—Typically, architecture optimization searches for good architecture candidates based on analyzing a model of the system. Model-based analysis inherently relies on abstractions and estimates, and as such produces approximations which are used to compare architecture candidates. However, approximations are often not sufficient due to the difficulty of accurately estimating certain extra-functional properties. In this paper, we present an architecture optimization approach where the speed of model-based optimization is combined with the accuracy of monitored system runs. Model-based optimization is used to quickly find a good architecture candidate, while optimization based on monitored system runs further refines this candidate. Using measurements assures a higher accuracy of the metrics used for optimization compared to using performance predictions. We demonstrate the feasibility of the approach by implementing it in our framework for optimizing the allocation of software tasks to the processing cores of a multicore embedded system.**

I. Introduction

As software systems are becoming increasingly complex, methods that aid in searching for good architecture candidates with respect to quality attributes are necessary. Architecture optimization is a problem that has generated a lot of interest in the research community [1]. Typically, architecture optimization pairs performance prediction, obtained by performing analysis on a system representation in the form of a model, with a mechanism that explores the search space by iteratively modifying certain aspects of the architecture. This enables assessing and comparing a large number of candidate architectures in reasonable time. Model-based architecture optimization is needed to efficiently handle the large number of possible architectures, and to allow optimization in early stages of development. However, there is always a limit to how accurate the optimization can be when based on models that approximate the real system.

In this paper, we present a novel architecture optimization approach, and corresponding framework, that exploits a combination of model-based predictions for optimization speed, and measured runtime values — gathered from executing the generated system code — for optimization accuracy. The method allows quickly finding a good architecture candidate by prediction-based optimization at modeling level, and then further refining it by continuing the same optimization mechanism, but now based on monitored system runs. Enriching optimization with performance measurements, rather than only using performance predictions, increases the accuracy of the performance metrics used for optimization, with respect to the actual system properties. We first present our general method for architecture optimization that combines model-based analysis with system execution, and then demonstrate it by implementing it in our framework for optimizing the allocation of software tasks to the cores of a multicore embedded system.

The paper is organized as follows. In Section II we discuss the motivation for enhancing model-based architecture optimization with monitored system execution and outline our proposed method. Next, in Section III we present an instantiation of the method for embedded systems — we describe our framework for optimizing allocations of software tasks to the cores of a multicore embedded system. Section IV demonstrates the feasibility of the approach — we present an experiment performed using the framework. Section V surveys related work, before we conclude the paper and present future work in Section VI.

II. Enhancing architecture optimization

Architecture design is one of the most important activities when developing a software system, since decisions made during architecture design have a considerable impact on the quality attributes (extra-functional properties) of the system (among other aspects such as the cost of development). Typically, architecture-level decisions include selecting software and hardware components, allocating software components to the available hardware nodes, deciding on system topology, etc. Due to the ever increasing system complexity, software architects face today a search space where manual exploration is not sufficient. This has made automated architecture optimization a prominent research topic over the recent years [1]. Research on architecture optimization has been done for various system domains (e.g., enterprise systems, embedded systems), various system representations (e.g., mathematical models, architecture description languages) and extra-functional properties (e.g., reliability, timing), with varying dimensionality (optimizing for a single or for multiple extra-functional properties), degrees of freedom (e.g., component selection, allocation, scheduling) and optimization strategies (e.g., local search, genetic algorithms). However, in general the methods are structured in a similar way: they use model-based analysis to predict extra-functional properties which are used to compare architecture candidates, and pair the analysis with an optimization strategy (search mechanism).

Model-based analysis inherently relies on abstractions and estimations, and thus gives approximate results. This reduced accuracy can be problematic, in particular for applications

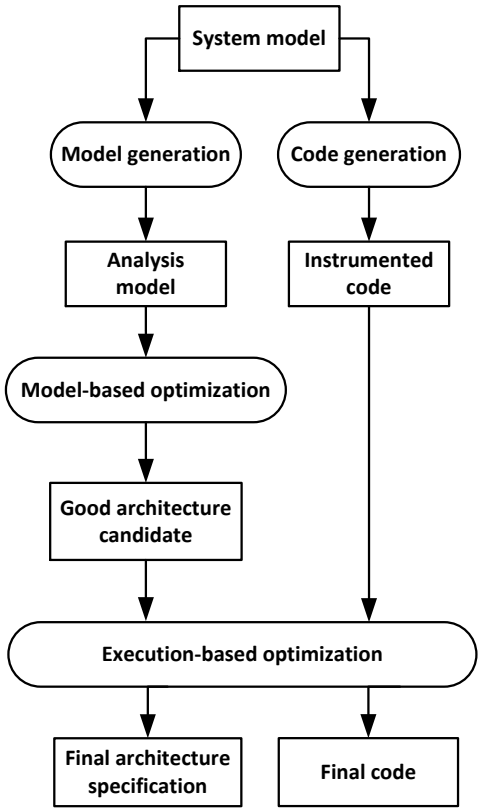


Fig. 1: An overview of the proposed combined model-based and execution-based architecture optimization method

with crucial performance-related extra-functional properties that require runtime measurements in order to be sufficiently assessed. On the other hand, due to the large search space, optimization based purely on runtime measurements is typically too time consuming to be feasible. For these reasons, we want to enhance optimization based on model analysis with monitored system runs. Towards this goal, the model of the system is used to generate the implementation, instrumented with code for measuring the extra-functional properties of interest. In this way, system runs can be monitored for gathering performance measurements, which are in turn used for assessing architecture candidates. By combining model-based and execution-based optimization, performance predictions are complemented with performance measurements, for both optimization speed and accuracy.

Figure 1 illustrates the combined model-based and execution-based architecture optimization method we propose. A system representation in the form of a system model is used as input. From the model, by means of automatic model transformations (model-to-model and/or model-to-text), an analysis model is generated, from which the extra-functional properties of interest can be predicted. Similarly, instrumented implementation code is also generated automatically, from which the extra-functional properties of interest can be measured during execution.

The method first runs an optimization cycle based on performance predictions by model analysis (and/or simulation). It uses the analysis model mentioned above, and modifies it in each iteration of the optimization in order to analyze and evaluate a particular architecture candidate. Since model analysis is performed on an abstraction of the system, it can typically produce performance predictions quite fast. Therefore, model-based optimization is used to quickly asses a large number of candidate architectures and thus rapidly find a good candidate. Then, execution-based optimization takes over. In each iteration, it modifies the generated code in order to execute and evaluate a particular architecture candidate. It is comparatively slower, but it leverages the good candidate identified by model-based optimization as its starting point, and can thus be run for fewer iterations. In return, the extra-functional properties that are obtained by runtime measurements are more accurate than the model-based predictions. In other words, we combine the speed of model-based analysis with the accuracy of performance measurements. The final output of the method is twofold: (i) the resulting architecture specification, and (ii) an implementation of the system.

III. Combined model-based and execution-based architecture optimization for embedded systems

In this section we present an instantiation of the method described in the previous section, for the domain of embedded multicore systems. By extending our previous work presented in [2] and [3], we developed a framework for optimizing allocations of software modules (in the domain of embedded systems called tasks) to the processing cores of the hardware platform. The framework is built on a combination of model-based and execution-based optimization. We first describe the context of the work, before discussing specific details about the framework, each in their respective subsection.

A. Context

In our work, we do architecture optimization for multicore embedded systems. Multicore technology has emerged over the last years as an answer to the increasing performance demands of embedded systems. The increase in performance demands comes from the fact that modern embedded systems include more functionality than before (for instance, functionality that was traditionally realized in hardware is instead being implemented in software, like software-defined radio [4]), but also from the fact that the included functionality is becoming more and more advanced (e.g., vision-based driver assistance in modern cars). While increasing the number of processing units does indeed increase the performance capacity, it also introduces the problem of how to allocate (deploy) the software tasks to the processing cores of the hardware platform. For some extra-functional properties, the allocation can have a significant influence. An intuitive example is schedulability — if we allocate too many tasks to the same core, the core will become overloaded and the tasks will miss their deadlines.

In particular, we focus on soft real-time multicore embedded systems, where timing is crucial for the correct behavior (a logically correct result that is produced at the wrong time point is equivalent to a logically incorrect result), however, a certain number of deadline misses can be tolerated (as opposed to hard real-time systems where the absence of deadline misses has to

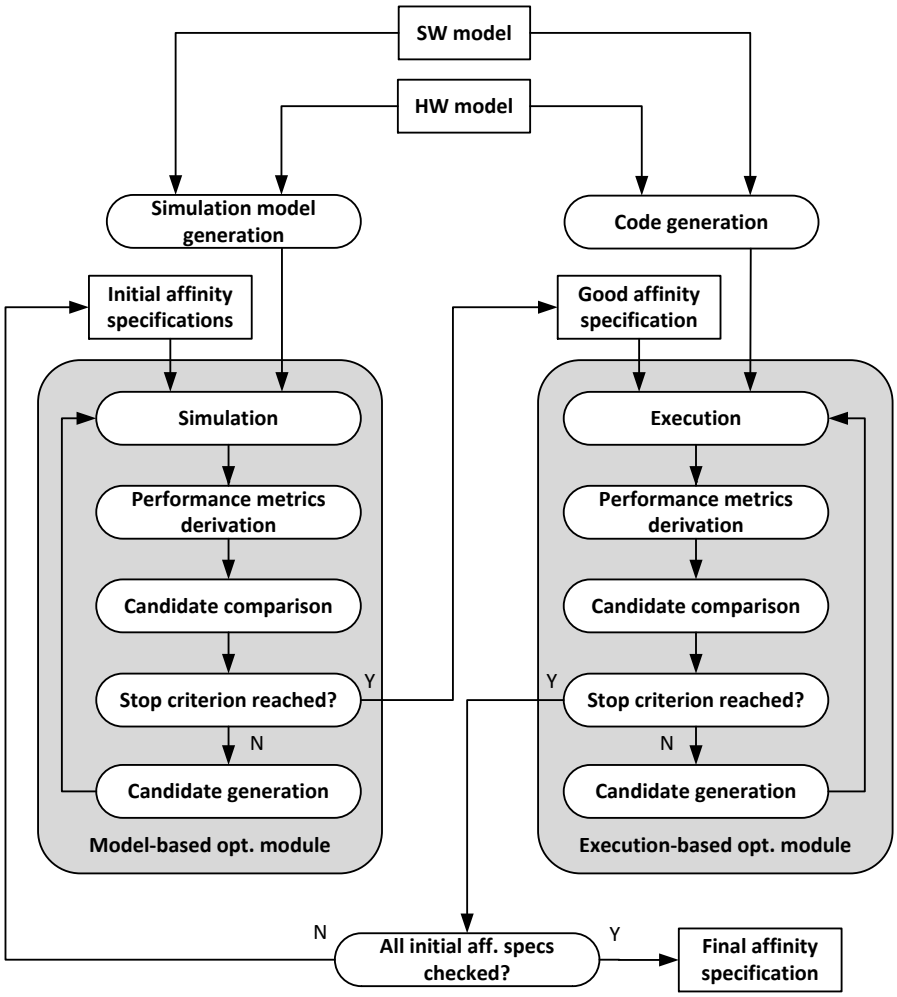


Fig. 2: Framework for task allocation optimization

be guaranteed). With timing having a crucial role, the extra-functional properties in our focus are timing-related attributes such as end-to-end response times, deadline misses and core load. Furthermore, since we consider soft real-time, we focus on average-case behavior (as opposed to worst-case behavior in hard real-time systems). Since these properties depend on the dynamic interplay between the tasks and since we are interested in average-case behavior, model-based performance prediction cannot easily derive the properties analytically from task and platform parameters. Instead, they are obtained by performing model simulation.

B. Framework for model-based and execution-based task allocation optimization

Our framework for task allocation optimization is depicted in Figure 2. As part of the architecture design phase, using UML and the MARTE profile [5], the system architect defines the software and hardware models of the system. The former defines the software architecture of the system being built, in terms of tasks and the connections between them, while the latter specifies the hardware platform, including the number of available cores and the type of scheduler for each core. In addition to these models, the architect can also define a set of deployment models in terms of mappings of tasks to cores, so called initial affinity specifications, to be used as starting points for the optimization mechanism. This is however optional, as the framework can automatically generate the desired number of initial affinity specifications.

In the model-driven engineering methodology, one of the core aspects is the provision of automation in terms of model manipulation and refinement, which is performed through so called model transformations. A model transformation translates a source model to a target model while preserving their well-formedness [6]. In our approach, we exploit model-to-text transformations which translate source model artefacts into structured text. The framework navigates the software and hardware models designed using UML and MARTE, and by means of automatic model-to-text transformations defined using the Xtend language [7] generates (i) a simulation model and (ii) instrumented system code. The former is a Java class file that represents an executable model of the system to be fed as input into our task simulator. The latter represents an

implementation of the system in C, instrumented with code to extract the extra-functional properties of interest. In other words, the former is used to obtain performance predictions, while the latter gives performance measurements.

Having generated all the necessary artefacts, in the next phase optimization (depicted by the two gray rounded rectangles in Figure 2) is performed. The optimization mechanism tries to minimize the end-to-end response times for particular task chains, while keeping the overall number of deadline misses in the system below a desired limit. Task allocation is the supported degree of freedom, i.e., the aspect of the system that the optimization mechanism is allowed to vary. As task allocation is a bin-packing like problem, which is NP-hard [8], rather than finding the optimal solution, the goal of the framework is to find a good solution quickly. We have therefore opted for a simple optimization strategy, namely local search paired with a domain-specific heuristic [3]. The optimization is repeated for each initial affinity specification, and in order to avoid local optima, there should be a large number of initial affinity specifications (some of which can be provided explicitly by the system architect, while the rest can be randomly generated, as mentioned above).

The overall structure of the two optimization modules is the same, as can be seen in Figure 2. In each iteration, the framework generates a new architecture candidate by making a small modification to the best candidate found thus far, derives relevant performance metrics for the new candidate, and determines whether it was an improvement over the best candidate. This continues until the stop criteria have been met. The stop criteria can be a fixed number of optimization iterations, a certain number of iterations that have not found an improvement (a better allocation candidate), a given time limit, or a combination of the latter two (for example, run the optimization cycle for at most 10 minutes or until there is no improvement in 10 consecutive iterations). Where the two modules differ is how the relevant performance metrics are derived: in the case of the model-based optimization module we have performance predictions obtained by model simulation, while in the case of the execution-based optimization module we have performance measurements obtained by executing the generated system code. These complementary ingredients, one based on model simulation and the other based on system execution, represent the novelty of the optimization mechanism.

Optimization starts with the model-based module (the left gray rounded rectangle in Figure 2). In each iteration, the simulation model is complemented with information about a particular affinity specification and as such represents a particular architecture candidate. Upon having executed the simulation model, from the data obtained by the simulation, we derive average end-to-end response times and deadline misses for task chains, and information about core load. These performance metrics are used to compare different affinity specifications against each other. As mentioned above, the best candidate is kept, and used to generate a new candidate to be tested in the next iteration. Since model simulation is faster than executing the system, we use model-based optimization to quickly converge to a good affinity specification.

Having done this, the execution-based module of the optimization (the right gray rounded rectangle in Figure 2) takes over, using the affinity specification identified by the model-based optimization module as its starting point. In each iteration, the generated code is complemented with a particular affinity specification, representing a particular architecture candidate. Since executing the system in order to obtain performance measurements is slower than performing model simulation, execution-based optimization is typically done for fewer iterations compared to model-based optimization. Having obtained the performance measurements in each iteration, from these we compute the concrete performance metrics, which are then in turn used to compare the different affinity specifications against each other. Again, the best candidate is kept and used to propose a new candidate for the next iteration. Both optimization modules can use the same candidate comparison criteria and search heuristic, but these could also be tailored for each optimization module. When the cycle stops, it outputs the best candidate it was able to find.

IV. Experiment

In this section we present an experiment performed using the framework described in the previous section. The point of the experiment is to demonstrate the feasibility of the combined model-based and execution-based architecture optimization approach. In particular, we aim to show that:

i a combination of model-based and execution-based optimization is more efficient than pure execution-based optimization, and that
ii execution-based optimization still has room to improve the optimization goal even after the preceding model-based optimization got stuck.

We start by describing the experiment setup, and then present and discuss the results.

The optimization goal in the experiment was to minimize the average end-to-end response time for a particular task chain. A task chain is a chain of control flow, and is defined by a periodic task and a number of event-triggered tasks triggered in sequence. An event-triggered task is activated when the task preceding it finishes one instance of execution. This means that each chain is activated with the same period as the periodic task at the start of the chain. The end-to-end response time for a chain is defined as the time elapsed between the point when the periodic task at the start of the chain gets activated and the point when the last task in the chain finishes the corresponding execution instance.

The software and hardware models of the experiment system are shown in Figure 3. We aimed for a representative system that contains task chains of varying length. The chain whose end-to-end response time was optimized is the one consisting of the following tasks: $t11$, $t12$, $t13$, $t14$ and $t15$. The execution platform had two cores, each running a preemptive priority-based scheduler, which means that when multiple tasks are ready for execution on a core, the one with the highest priority executes until it is finished or until a task with higher priority becomes ready.

The experiment consisted of two parts. In the first part, we performed the combined model-based and execution-based optimization, while in the second part we performed only execution-based optimization. We executed both parts for

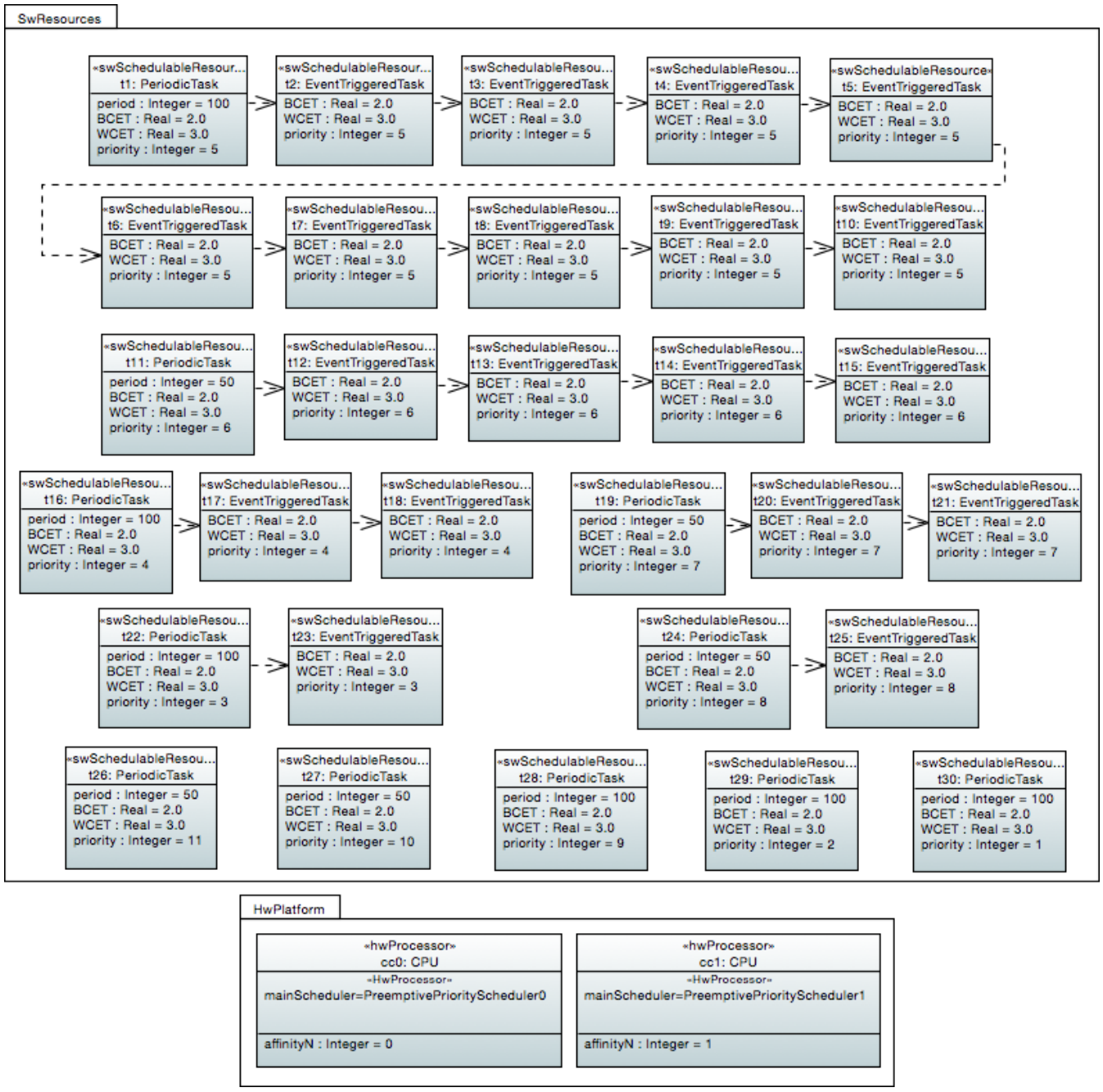
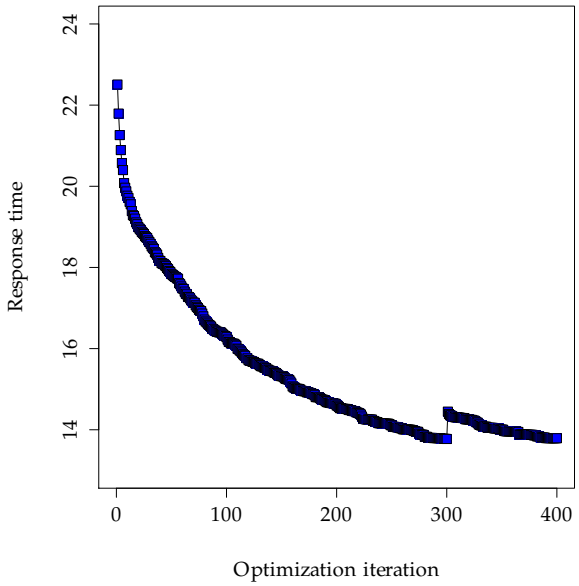


Fig. 3: Software and hardware models of the experiment system

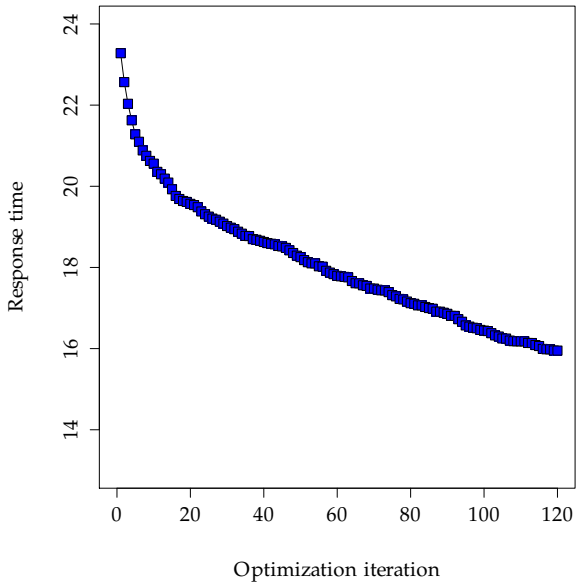
roughly the same amount of time, in order to be able to compare the end result. In both parts, we repeated 100 optimization runs. All optimization runs started from a single initial affinity specification with an equal number of tasks allocated to each core: tasks $t1$ to $t3$ allocated to core 0, tasks $t4$ to $t6$ allocated to core 1, tasks $t7$ to $t9$ to core 0 and so on. In the first part of the experiment, each optimization run consisted of 300 iterations of the model-based optimization module followed by 100 iterations of the execution-based module. The latter continued with the best affinity specification identified by the former. In the second part of the experiment, each optimization run consisted of 120 iterations of the execution-based module.

As mentioned above, the numbers of iterations were chosen so that both parts of the experiment take roughly the same time. Each execution was roughly 30 times slower than each simulation, meaning that 300 simulations and 100 executions took similarly long time as 120 executions. Each simulation was performed for 2000 simulation steps (clock ticks), while each execution was run for 20 seconds. This was chosen in order to:

i run the system long enough (20 hyper-periods) to capture the average behavior, and to
ii result in a similar number of activations of the optimized chain in the simulation and the execution.



(a) Combined model-based and execution-based optimization



(b) Execution-based optimization

Fig. 4: Experiment results

In all optimization runs, for both the model-based and execution-based optimization module, for proposing the affinity specification candidate to be tested in the next iteration, we used a simple heuristic that randomly relocated one task to a different core.

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

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

In the implementation, each task is a POSIX thread with read-execute-write semantics — it first reads input data, then performs calculations and finally writes output data. Task calculations are represented by running a loop that repeats a simple addition, and they take a considerably longer time than reading and writing data. The implementation is instrumented with code for measuring the end-to-end response times for chains. The executions were performed on a system with an Intel Core 2 Duo E6700 processor [9], running the 32-bit version of the Ubuntu 12.04 LTS operating system (kernel version 3.2.29). The operating system is patched with the PREEMPT RT patch (version 3.2.29-rt44) [10], 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.

Figure 4 shows the results of the experiment: Figure 4a for the first part (combined model-based and execution-based optimization) and Figure 4b for the second part of the experiment (pure execution-based optimization). In the first part of the experiment, the model-based optimization module (which consisted of 300 optimization iterations) took 195.3 seconds on average over the 100 optimization runs, while the execution-based module (100 optimization iterations) took 2099.9 seconds. In the second part, the execution-based module (120 optimization iterations) took 2520 seconds on average. Each point in the diagrams shows the chain 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. The similar duration of both parts of the experiment allows us to compare their end results: the combined optimization on average found an affinity specification with 13.79 as the end-to-end response time for chain $t11 - t15$, while the pure execution-based optimization found an allocation with 15.95 as the response time. This speaks clearly in favor of the combined optimization, and using model-based optimization to quickly converge to a good allocation.

Looking at Figure 4a, we can also see why the combined optimization is better than pure model-based optimization (in addition to the motivation discussed in Section II, regarding the validity of the extra-functional properties and the insufficiency of pure model-based optimization for particular applications). In the last 10 iteration steps of the model-based optimization module (just before step 300 in the diagram), we can see that there is very little improvement. After step 300 the curve again shows a slight decline, meaning that the execution-based optimization module still has room for improving the optimization goal, even after the model-based module made no significant progress.

An important aspect that needs to be clarified here is the jump in the response time value at step 300. This is not a deterioration or a step back in the optimization. The jump is expected when moving from the model-based optimization

module to the execution-based one. It depends on the accuracy of the model-based performance prediction, and could in the general case be a negative or a positive jump. This also means that the final response time value after both optimization modules and the response-time value after the model-based optimization module cannot be compared based solely on their absolute values, since one is a predicted value coming from the simulation mechanism, while the other is a measured value, and thus has a higher confidence.

For this particular optimization problem, where the model-based performance prediction is done by means of simulation, the iterations in the execution-based module took roughly 30 time longer time than those in the model-based module. For optimizations where performance predictions can be done analytically, without simulation, the difference would be much bigger, meaning that the model-based module could process an even larger portion of the search space.

V. Related work

Software architecture optimization methods are variegated and focus on disparate aspects. Aleti et al. [1] performed a systematic literature review to analyze the solutions provided by the different research communities. Moreover, they provide a taxonomy to classify the existing research in order to establish a common reference for the many problem variants that fall under software architecture optimization.

In several research works the idiosyncratic dependency between software architecture and quality is highlighted, especially when it comes to the embedded domain [11], [12]. Evaluating the quality of a system in relation to particular architectural aspects, such as a specific deployment candidate, is a critical and sensitive task that should be dealt with already at modeling level. Various model-driven approaches provide ways to model specific quality attributes and push them to the software level [13], [14]. Zhu et al. [15] present model-driven facilities 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 minimize latencies.

Specifically to deployment optimization, techniques have been defined to automatically explore the space of deployment options to identify the near optimal candidate. Part of these solutions aim at only satisfying predefined constraints [16], while others seek a near optimal deployment candidate without violating a set of given constraints [17], [18].

Model-driven approaches provide good approximations, but this is often not enough for embedded applications where runtime measurements are needed in order to assess certain performance related quality attributes. Several approaches dealing with optimization based on measurements at system implementation level can be found in the literature, as described in [19]. The COMPAS framework by Mos et al. [20] 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 et al. [21], and PAD, by Parsons et al. [22].

The goal of these approaches is either to provide optimization based on the modeled architecture, or to identify performance issues in the running system and adapt the corresponding code to make it able to fulfill specified constraints. An approach which attempts at combining models and runtime values is presented by Sailer et al. [23]. The authors present a model-based optimization approach for the task allocation problem in the embedded domain starting from a system description in AUTOSAR and runtime measurements of the related runnables in hardware traces. A genetic algorithm is then used to create and evaluate solutions to the task allocation problem. While leveraging runtime values for generating simulation models and optimizing allocation, the approach does not provide any prediction-based 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.

Regarding measurements at system implementation level, besides runtime monitoring, other verification techniques (e.g., static analysis) can be employed, even though their application for large and complex systems may not always be practically and economically favorable [24]. When these techniques are applicable, conditions that may cause invalidation of the analysis results at runtime may still occur. An example of this could be 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 [25]. For this reason, the information gathered through monitoring system execution is useful for (i) observing the actual system's behaviour and to detect violations at runtime, and for (ii) making adaptation decisions. As an example, in [26] the authors use monitoring information for balancing timing and security properties in embedded real-time systems. Huselius et al. [27] describe a method for the generation of design models of embedded real-time systems from the monitoring information gathered at runtime. In this paper we leverage monitoring results from which observed values for selected system properties are computed and used to improve deployment.

VI. Conclusion and future work

Since model-based analysis gives performance predictions using abstractions and approximations, architecture optimization based only on model analysis is not sufficiently precise for all applications, and needs to be paired with performance measurements obtained by running the system. In this paper, we presented our approach for combined model-based and execution-based architecture optimization. The approach 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 implemented the approach in our framework for optimizing task allocations in multicore embedded systems and carried out an experiment that demonstrated the feasibility of the approach. In particular, we showed that the combined optimization can be superior to both pure execution-based and pure model-based optimization. Regarding the former, we demonstrated that leveraging the speed of model-based optimization and using its result as a starting point for execution-based optimization can save a lot of time compared

to only performing execution-based optimization. Regarding the latter, even when model-based optimization showed no significant improvement, there was still room for refining the architecture candidate further by performing execution-based optimization. Furthermore, ending the optimization mechanism with execution-based optimization makes the extra-functional properties used for comparing different architecture candidates more likely to be valid in the final system. Even though the experiment was limited in scope, it clearly demonstrates the feasibility and value of the approach.

As future work, we plan to perform more thorough experiments, including more types of systems, optimization of additional extra-functional properties (e.g., deadline misses) and using other optimization heuristics, such as the delay matrix heuristic we developed for end-to-end response time optimization [3]. This would require extending the code generation mechanism, so that the generated instrumentation code can extract additional performance metrics. Furthermore, an interesting experiment would be to implement the idea of combined model-based and execution-based architecture optimization in an optimization framework in a different application domain.

A possible extension of the method is to use the generated system code to measure certain extra-functional properties and back-propagate them to the models through specific in-place model-to-model transformations in a similar way to what was proposed in [28]. This information would be shown as extra-functional decorations of specific software model elements, specifically tasks and task chains. Doing so, measured values would be used instead of estimates for increasing the accuracy of the model-based performance predictions.

Acknowledgments

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

References

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 2013.

[2] J. Feljan, J. Carlson, and T. Seceleanu. Towards a model-based approach for allocating tasks to multicore processors. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 117–124, 2012.

[3] Juraj Feljan and Jan Carlson. Task allocation optimization for multicore embedded systems. In *40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2014.

[4] T. Ulversoy. Software defined radio: Challenges and opportunities. *Communications Surveys & Tutorials, IEEE*, 2010.

[5] The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. http://www.omgmarte.org/, [Accessed: 2014-11-28].

[6] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Procs of OOPSLA*, 2003.

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

[8] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, Dept. of Mathematics, 1973.

[9] 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].

[10] PREEMPT RT patch. https://rt.wiki.kernel.org/index.php/Main_Page, [Accessed: 2014-12-19].

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

[12] E. Bondarev, P. de With, M. Chaudron, and J. Muskens. 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, 2005.

[13] S. Islam, R. Lindstrom, and N. Suri. Dependability driven integration of mixed criticality SW components. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2006.

[14] L. Grunske, P. Lindsay, E. Bondarev, Y. Papadopoulos, and D. Parker. An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems. In *Architecting dependable systems IV*, pages 188–209. Springer, 2007.

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

[16] A. Martens and H. Koziolek. Automatic, model-based software performance improvement for component-based software designs. *Electronic Notes in Theoretical Computer Science*, 253(1):77–93, 2009.

[17] N. Medvidovic and S. Malek. 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*, pages 47–51. ACM, 2007.

[18] J. Fredriksson, K. Sandström, and M. Åkerholm. Optimizing resource usage in component-based real-time systems. In *International Symposium on Component-Based Software Engineering (CBSE)*, pages 49–65. Springer, 2005.

[19] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation, Special Issue on Software and Performance*, 2010.

[20] A. Mos and J. Murphy. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *Third International Workshop on Software and Performance (WOSP)*. ACM, 2002.

[21] A. Diaconescu and J. Murphy. 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, 2005.

[22] T. Parsons and J. Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology*, 2008.

[23] A. Sailer, S. Schmidhuber, M. Deubzer, M. Alfranseder, M. Mucha, and J. Mottok. Optimizing the task allocation step for multi-core processors within AUTOSAR. In *International Conference on Applied Electronics (AE)*, pages 1–6, Sept 2013.

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

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

[26] M. Saadatmand, A. Cicchetti, and M. Sjödin. Design of adaptive security mechanisms for real-time embedded systems. In *4th International conference on Engineering Secure Software and Systems (ESSoS)*, 2012.

[27] J. Huselius and J. Andersson. Model Synthesis for Real-Time Systems. In *Ninth European Conference on Software Maintenance and Reengineering (CSMR)*, 2005.

[28] F. Ciccozzi, A. Cicchetti, and M. Sjödin. Round-trip support for extra-functional property management in model-driven engineering of embedded systems. *Information & Software Technology*, 55(6):1085–1100, 2013.