BOTTOM-UP PERFORMANCE ANALYSIS OF HW/SW PLATFORMS

Kai Richter, Dirk Ziegenbein, Marek Jersak, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
D-38106 Braunschweig / Germany
{richter|ziegenbein|jersak|ernst}@ida.ing.tu-bs.de

Abstract Today's complex embedded systems integrate multiple hardware and software components, many of them provided as IP from different vendors. Performance analysis is crucial for such heterogeneous systems. There already exists a variety of formal timing analysis techniques for small sub-problems, e. g. task performance, scheduling strategies, etc.. In this paper, we analyze these individual approaches in the context of performance analysis of heterogeneous platforms at different levels of abstraction, and present a three-level bottom-up analysis procedure.

Keywords: Timing Verification, Scheduling Analysis, Platform-Based Design

Introduction

The advances in silicon technology enable the integration of more and more functionality in a single system. (Future) platform-based systems-on-chip (SoC) integrate multiple programmable micro-controllers and DSPs with specialized memories and dedicated or weakly programmable hardware accelerators, which are connected using complex on-chip networks. Examples are multimedia-processors as found in set-top boxes and entertainment systems. A similar development can be observed for physically distributed systems.

The design of such systems is driven by conflicting objectives and constraints such as cost, timing, power, and reliability. The increasing heterogeneity of such systems with respect to architecture components, design tools, specification languages, etc., adds to the overall design complexity. As a result, the design can only be managed with enormous increase in design productivity in order to meet time-to-market requirements. In response, industry shifted to platform-based design. Reusing existing hardware components, often provided as IP, drastically reduces design time. Operating systems and device drivers supplement the hardware components, and well defined APIs allow fast, mainly software-based platform customization.

The emerging limits of todays platform integration methodologies are verification and optimization of system timing and memory. Verification is currently limited to simulation approaches, such as in VCC [3], Seamless CVE [16], CoWare [4], or CoCentric [22]. In addition to the long-term simulation runs, simulation approaches provide only marginal optimization support, since the influence of the individual design decisions on the simulation results is unclear. Other known limitations of simulation such as incomplete coverage and corner case identification are aggravated since many of the design errors only result from system integration requiring detailed knowledge which is often not available to the integrator.

Formal analysis techniques represent a promising alternative for timing and memory constraint verification. In contrast to simulation, formal analysis ensures complete corner case coverage by nature. With a carefully selected level of system and component abstraction, the results are conservative, i. e. they are guaranteed under all circumstances. Formal analysis approaches capture the component and system properties using parameterized mathematical models. Such models account for instruction execution, critical program paths, scheduling influence, and component interaction in order to derive conservative bounds on system function timing or required memory size.

In this paper, we discuss and evaluate several basic ideas underlying formal analysis approaches in the context of HW/SW platform integration. We investigate a representative subset of existing individual analysis approaches in order to find a reasonable level of detail at which platform performance analysis can be currently performed. We propose a three-step bottom-up approach. While much effort has been put into timing analysis at the task and the single resource level, approaches to global timing analysis of complex heterogeneous HW/SW platforms are comparatively rare. Therefore, the component interaction is discussed more thoroughly.

The paper is organized as follows. The next section presents out three level analysis hierarchy. Section 2 reviews analysis techniques for single tasks. The influence of operating systems on single architecture components is considered in Section 3. The analysis of multi-component platforms is captured in Section 4. We conclude the paper with an evaluation of the current analysis possibilities.

1. The Three-Level Approach

The analysis of target system timing can be divided into three parts:

1 analysis of a single task executed on a target system component in absence of resource sharing techniques (task level)

2 analysis of resource sharing effects on a single resource component (resource level)

3 analysis of component interaction in multi-component systems (system level)

At the task level, timing analysis is performed separately for each task. Upper and lower bounds on task execution times are obtained. Since these times may depend on the target architecture (i. e. the processor the task will be executed on), the implementation has to be considered in this step. The sources of intervals can be input data dependent task behavior (due to input data dependent control structures) or limited analyzability of the target architecture (due to features like pipelining, caches etc.). There are many recent contributions combining implicit or explicit program path analysis and cycle-true processor modeling, such as [13, 6, 25, 9]. In addition, task communication can be determined to analyze communication channel load and timing [25]. Such parameters are then captured using a reasonable, abstract intermediate representation (abstract task model), e. g. SPI [5].

At the resource level, there is a huge amount of work, e. g. [14, 12, 24] and many more, mainly in the domain of real-time operating systems to calculate task response times. Again, these response times abstract from the actual implementation. All of these approaches assume certain event models, e. g. periodic or burst. From the IP perspective, these can be used to specify operation conditions for which the response times are guaranteed.

For single-component systems, the resource level already is the system level. There also exist many techniques for homogeneous multi-processors [20]. However, this does not hold for heterogeneous multi-component systems.

The system-level analysis for heterogeneous HW/SW platforms has been neglected for a long time. In a recent publication, Pop et. al. [17] extended the existing scheduling analysis to specialized classes of multi-component platforms. In [18], we proposed to couple the existing approaches rather than finding solutions with only limited applicability. We identified the lack of compatibilities between event models as being the key obstacle for this analysis coupling process, and proposed to use event model interfaces within an iterative event model interfacing technique.

Each analysis technique focuses on only one of the three mentioned steps: task, resource, or system level analysis. As a result, a sound and complete analysis of the whole system requires several individual approaches to be combined bottom-up, i. e. starting at the task-level and ending up with complete system level performance information. Furthermore, all approaches make certain assumptions on the system's tasks and resources. Otherwise, the techniques can not be reasonably applied to a given problem. Here, two important issues have to be considered. First, it is a prerequisite that the lower levels within this analysis hierarchy provide the information that is required by the more high-level approaches. In other words, the information is propagated through the analysis bottom-up. Secondly, the information provided should not be overly detailed. Otherwise, these details can neither be utilized by a subsequent analysis step, nor can analysts benefit from the efficiency of less detailed approaches. In general, the performance parameters that are propagated should be consistent at all levels of the analysis hierarchy. The following sections introduce analysis techniques at each of the mentioned steps (task, resource, system).

2. Task-Level Analysis

This section briefly summarizes timing analysis approaches for individual tasks in absence of operating systems. Typically, such analysis is based on basic blocks, i. e. blocks of code with a single control flow. The latency time model in [13] is established as a standard model for static approaches, which is also called the *sum-of-basic-blocks* model. Here, the overall task execution time is the sum of all basic block execution times multiplied by the corresponding execution count for each of the basic blocks. Evidently, the execution time of a basic block depends on the target architecture whereas the execution count is architecture independent. Both values, time and count, are intervals representing the worst case and best case bounds. As a result, the overall execution time is an interval, too.

It is assumed that all executions of one basic block have the same time interval. However, data dependent instruction execution and pipelined architectures as well as unpredictable cache behavior and register allocation lead to a widely varying basic block execution time. This effect is referred to as overlapping basic block execution. Many other approaches to task execution time analysis are also based on the analysis granularity of basic blocks or single basic block transitions [8] or require complex modifications to execution time determination [15]. Very few approaches like [9] also consider more fine-grain influences of complex processor architectures, e. g. pipelines and super-scalar machines. The major drawback of such detailed approaches is state-space explosion.

The SYMTA (SYMbolic Timing Analysis) tool suite [25] extends the sum-of-basic-blocks approach by raising the analysis granularity from basic blocks to task segments which are sequences of basic blocks having a single input data independent control flow across basic block boundaries. Due to the raised granularity, the number of points where worst case assumptions (e. g., empty pipeline) have to be made is reduced leading to a higher analysis accuracy.

Additionally, SYMTA allows to specify task execution contexts which provide information on the input data in order to predict input data dependent control structures. This way, execution paths are selected and task segments can be merged even further. As a result, not only a single task execution time interval but also a set of comparatively narrow execution time intervals, one for each context, can be given.

3. Resource-Level Analysis

In this section, the influence of resource sharing is investigated, based on the core execution times from the previous section.

In the area of real-time operating systems, there are several substantially different resource sharing (scheduling) strategies. Preemptive scheduling based on static priorities (e. g. rate-monotonic), dynamic priorities (e. g. earliest deadline first), and time-slicing (e. g. time division multiple access and round robin) are among the most important strategies. For each of the mentioned strategies, a set of analysis approaches exist. The approaches take the scheduling strategy and the core execution times (from the previous section) as input to a self-contained mathematical description in order to calculate conservative bounds on the response times of tasks.

In the early 70's, Liu and Layland proposed a preemptive priority-driven scheduling to guarantee deadlines for periodic hard real-time tasks [14]. They considered a static (Rate Monotonic) and a dynamic (Earliest Deadline First) priority assignment and provided a formal analysis framework for both. In [10], Kopetz and Gruensteindl proposed TTP (time triggered protocol) for communication scheduling in distributed systems and presented an analysis. TTP implements the TDMA (time division multiple access) scheduling strategy. Both contributions assume a periodic activation of tasks. Recent extensions of the mentioned work allow periodic activation with jitter, e. g. [20], and arbitrary deadlines [12]. Sprunt et. al. [21] analyze the influence of sporadic task preemption. Tindell presents an approach for task bursts [24].

Gresser [7] and Thiele et. al. [23] use more general activation models. They introduce a vector of sequential time intervals rather than a few parameters (period, jitter, etc.) only, in order to calculate performance quanta for each task. Gresser uses this model for analysis of dynamic priority (EDF) scheduling. Thiele et. al. analyze a mixture of hard and soft real-time tasks in network processors with a specialized scheduling algorithm.

The above mentioned approaches are only concerned with the coarse-grain influences of scheduling, i. e. task preemption or –in the case of non-preemptive scheduling– delay. However, the operating system that implements the scheduling strategy also needs to be considered. Most importantly, the context switching overhead has to be considered, as well as the OS drivers to control busses or other peripherals. If possible, these are usually included in the task execution time, e. g. communication primitives. The context switch overhead can be added to the overall execution time. House keeping functions can be treated as extra tasks without any specific functionality.

In contrast to the scheduler and the housekeeping functions, the OS drivers require special attention. Many drivers manage and modify static data structures, e. g. communication buffers and other queues, which are shared among several tasks. In other words, the driver function is called by more than one task. Clearly, such driver functions must not be preempted by calls to the same driver, since this would result in an unknown state of the shared data. This is usually solved using OS semaphores and/or monitors. As a result, a task may experience an additional delay to resolve such conflicts. Extensions to the basic scheduling analysis approaches can be found in [2, 20, 24]. Although such conflicts can not be seen before tasks are integrated for resource sharing, the information about the actual blocking times needs to be obtained at the individual task level. The approach in [25] can be used.

Other fine-grain influences, e. g. in the presence of caches and pipelines, are more complex to include in the performance models. Ferdinant et al. [6] account for the cache state during task performance analysis. This way, they compute worst-case delays resulting from cache misses which result from preemption. This problem is also addressed in [11]. Clearly, such influences can neither be completely captured at the task level, nor at the resource sharing level, since the actual task performance results from a combination of both. In contrast to semaphore blocking, most of the current scheduling analysis approaches do not include parameters representing fine-grain influences like caches, etc. These are usually conservatively estimated based on experience.

4. System-Level Analysis

While there exist a huge amount of work in the area of task and resource level analysis, there is only little work on system level analysis, i. e. the analysis of multiple connected and interacting resource components.

One reason for this lack of global models are incompatibilities of the input event models, e. g. periodic or burst. An input event model describes the frequency and type of input events that lead to task or communication execution. In effect, the input events determine the system workload. Resource sharing analysis techniques, therefore, typically assume certain input event models, as mentioned in Section 3.

The importance of transitions between different event models has been widely neglected in literature. In general, global heterogeneous system analysis is currently limited to special classes of problems. Pop et. al [17] extended the idea of self-contained mathematical equations as presented in Section 3 to capture distributed interacting tasks. Their approach is limited to static priority task scheduling combined with a TDMA bus protocol. However, it is doubtful that a general approach to a self-contained solution for arbitrarily complex platform architectures can be found, mainly because of the highly complex dependencies in such systems. Other approaches like [1] require a completely homogeneous system in terms of input description, scheduling algorithms and architecture structure. Then, the analysis problem is solved by finding critical paths in the system level description. Such approaches are neither applicable to strongly heterogeneous systems, nor do all of them account for the influence of scheduling.

In a recent publication [18], we have presented a more general approach. The basic idea is to re-use the existing work mentioned in Section 3 (and possibly the work in [17], too) for the individual components. At the system level, we couple the individual analysis to obtain a global platform performance model. This is not trivial due to the mentioned event model incompatibilities. However, we developed an event interface model to overcome the limitations of the individual scheduling analysis approaches. But before we present our event interface model, we will give a brief overview about event models in the context of scheduling analysis.

4.1. Event Models

In the literature on real-time analysis, there are four event models of major importance. A simple and efficient assumption is a stream of *periodic* input events. Here, the arrival of events can be captured by a single parameter, the period T. Often, periodic events are allowed to deviate with respect to their period. This adds another parameter to the periodic model, the *jitter* (J). Other models capture *bursts* of events. A burst is characterized by a number of events (burst length b) within a given time interval (the outer period T). This outer period may also jitter (J). If known, a minimum time distance (the inter-arrival time t) between two successive events within a burst can be specified. *Sporadic* events are captured by the minimum inter-arrival time t, only.

The models of Gresser and Thiele are closest to a general event model trying to capture rather complex event stream patterns. Both models target general event stream modeling at the cost of model complexity that excludes direct application of most of the analysis techniques mentioned above.

4.2. Event Model Coupling

In this section, we present the actual coupling of event models. We first discuss compatibility issues between the four event models introduced in the preceeding section, and derive event model interfaces (*EMIF*s) to transform the parameters of one event model into those of another model. In those cases, where no simple interface can be derived, we introduce the interposition of event adaptation functions (*EAF*s) like buffers and timers in order to couple initially incompatible event models. This step incorporates buffer sizing and event delay determination due to the additional system functions.

We introduce the basic idea of event model coupling using a simple example. Consider a (sub)system consisting of two tasks $\mathcal{T}_1$ and $\mathcal{T}_2$ which exchange data via event stream *ES*. Let us assume the two tasks are mapped to two different resources. Other tasks are implemented on both resources, too. Each resource implements a different resource sharing strategy (*RSST*). Let us furthermore assume, that the output of $\mathcal{T}_1$ is produced with jitter, while the analysis of $\mathcal{T}_2$ requires a sporadic event model. Clearly, the two local analysis approaches for $\mathcal{T}_1$ and $\mathcal{T}_2$ can not be coupled directly since the two event models (jitter and sporadic) are basically incompatible. However, we are able to *derive the required* parameter values of the sporadic event model *from the known* values of the jitter event model: The minimum temporal separation of two successive events with jitter is the period minus the maximum jitter: $t_2 = T_1 - J_1$. In the following, we generalize this idea.

4.2.1 Event Model Interfaces.

The intended use of event model interfaces (*EMIF*s) is shown in Figure 1. $\mathcal{T}_1$'s output event model X results from the selected technique to analyze the $\mathcal{T}_1$'s execution behavior with respect to the resource sharing strategy $RSST_1$. Similarly, the required input event model Y

$EMIF_{X\to Y}$	Y=periodic	Y=jitter	Y=burst	Y=sporadic
X=periodic	$T_Y = T_X$ (identity)	$T_Y = T_X, J_Y = 0$	$T_Y = T_X, b_Y = 1, t_Y = T_X$	$t_Y = T_X$ (lossy)
X=jitter	—	$T_Y = T_X, J_Y = J_X$ (identity)	—	$t_Y = T_X - J_X$ (lossy)
X=burst	—	—	$T_Y = T_X, b_Y = b_X, t_Y = t_X$ (identity)	$t_Y = t_X$ (lossy)
X=sporadic	—	—	—	$t_Y = t_X$ (identity)

Table 1. *EMIF*s for simple model transformations

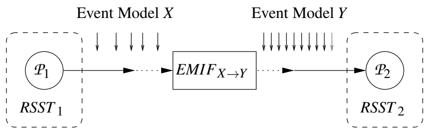

Figure 1. An event model interface ($EMIF$)

of $\mathcal{P}_2$ is determined by the selected analysis technique for $RSST_2$. Now, it is the $EMIF_{X\to Y}$'s task to translate the event stream's properties from event model X into an instance of event model Y. Such interfaces are generally uni-directional ($X \to Y$). They can only transform instances of X into instances of Y, not vice versa.

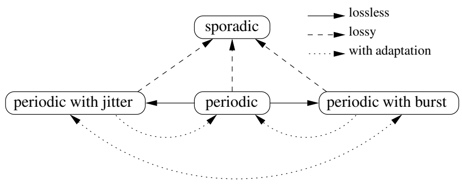

Figure 2. Possible event model interfaces

Figure 2 depicts the possible transformations ($X \to Y$) for which such an $EMIF$ can be found. The solid lines indicate that the interface losslessly (i. e. with the same modeling accuracy) transforms the information of model X into model Y, while the dashed lines indicate a loss of information during the transformation process. The dotted lines indicate that the transformation requires additional adaptation of the input event stream.

In the beginning of this section, we already solved this interfacing for the transformation (jitter→sporadic). The $EMIF$s of the other feasible transformations are given in Table 1. In case when $X = Y$, no event model interface is actually needed, since the parameter values are identical. As can be seen in the table, for only 5 out of 12 (not considering $X = Y$) possible event model transformations an $EMIF$ can be given. For the other seven transformations like (jitter→periodic) no $EMIF$ can be found. Which model combinations are interfaceable can be determined by formally analyzing the corresponding event models: we need to show that the behavior of every instance of event model X is contained in the set of all possible instances of model Y. In other words, we have to bound the given event stream X using the parameters of the event model Y. We have to find upper and lower bounds representing the maximum and minimum load arrival, respectively. Proofs can be found in [18].

4.2.2 Event Adaptation Functions. As already mentioned, periodic event streams with jitter can not be captured by purely periodic event models. However, in digital signal processing systems, internal events are often assumed periodic since the system's overall input is purely periodic. But due to resource sharing and/or data-dependent task execution times, internal events most likely experience a jitter. To keep up with periodic models, buffers and timers are widely used to re-synchronize such events according to the initial period. This shows that it is generally possible to couple initially incompatible event models for analysis at the cost of additional system functions. We refer these functions to as event adaptation functions (*EAF*s). An example for the use of *EAF*s is given in Figure 3.

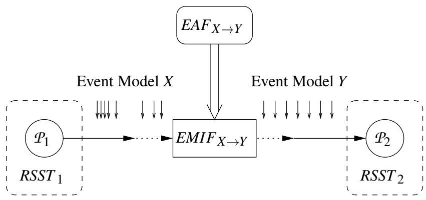

Figure 3. An *EMIF* with event adaptation function (*EAF*)

The actual functionality of these *EAF*s has to be derived from the two event models. For the above mentioned very simple case (X=jitter→Y=periodic), the *EAF* can be found straightforward. A buffer of size 1 and an output issue period of T_Y is needed. Now, we can derive an *EMIF* for this situation by simply setting T_Y to the value of T_X. In general, the parameters of the timed buffers need to be formally derived from the model transformations. Again, we refer to [18] for formal details.

4.3. Output Event Interfaces

The actual coupling of the individual analysis techniques from Section 3 is an iterative procedure of selecting analysis approaches and adapting the input event models depending on the analysis assumptions until all event models converge. This can be complex, since in many situations, the analyst has more than one option. Balancing between analysis efficiency (as in completely periodic events) and behavioral freedom (as in strongly bursty event streams) is a non-trivial task. Additionally, output event models have to be derived in order to find the required *EMIF*s and *EAF*s.

The basic idea of deriving output event models is simply based on abstract event propagation through architecture components. An input event activates a dedicated function inside the component. A corresponding output event will occur after the function is completed, i. e. after the corresponding response time. When having a constant response time, each event experiences the same propagation delay. Thus, the output model is identical to the input model. However, in complex software systems only upper and lower bounds for the response time will be given, e. g. due to data dependent task execution times or a varying number of preemptions by other tasks. For a periodic input model, the output is not purely periodic anymore, but will experience a jitter that equals the difference between the upper (t^{+}_{resp}) and the lower (t^{-}_{resp})response time bounds. An overview of how output event models can be derived from the analysis characteristics (input event model and response time) is provided in [19].

5. Conclusions

In this paper, we presented a three-level bottom-up approach to formal timing analysis of complex heterogeneous HW/SW platforms. We started with analyzing single task execution on the target processors. Subsequently, we analyzed the influence of resource sharing strategies for single architecture components. Both areas have been investigated in the past. However, this does not apply to the analysis of the complex interactions in heterogeneous multi-component platforms. We introduced an event interface model in order to couple the approaches to resource sharing analysis for single resources (step 2) to obtain a system-level performance model.

In summary, a sound and complete analysis of the whole system currently requires several individual approaches to be combined. Other approaches to gather platform performance at once are either not practical due to algorithm inefficiencies, or too restrictive to analyze real-world heterogeneous platforms.

The interface model enables the integration of analysis techniques in the same way components and simulators have been coupled in the past. Designers of complex platforms can for the first time benefit from the large amount of work in the area of formal timing analysis. However, existing component simulators can be integrated, since the formal nature of our methodology provides excellent corner case identification support. By carefully selecting the level of detail/abstraction, designers can balance between analysis efficiency and data accuracy.

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Journal of Real-Time Systems*, 8(5):284–292, 1993.

[3] Cadence. *Cierto VCC Environment*. http://www.cadence.com/products/vcc.html.

[4] CoWare. *CoWare N2C*. http://www.coware.com/cowareN2C.html.

[5] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich. Hardware/software codesign of embedded systems - The SPI Workbench. In *Proceedings IEEE Workshop on VLSI*, Orlando, USA, June 1999.

[6] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 16–30, 1998.

[7] K. Gresser. An event model for deadline verification of hard real-time systems. In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pages 118–123, Oulu, Finland, 1993.

[8] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, pages 53–70, January 1999.

[9] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *Proceedings of Design, Automation and Test in Europe (DATE '00)*, pages 552–559, Paris, March 2000.

[10] H. Kopetz and G. Gruensteidl. TTP - a time-triggered protocol for fault-tolerant computing. In *Proceedings 23rd International Symposium on Fault-Tolerant Computing*, pages 524–532, 1993.

[11] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *IEEE Transactions on Computers*, pages 700–713, 1998.

[12] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings Real-Time Systems Symposiom*, pages 201–209, 1990.

[13] Yau-Tsun Steven Li and Sharad Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[14] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[15] T. Lundquist and P. Stenström. Integrating path and timing analysis using instruction level simulation techniques. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, Canada, June 1998.

[16] Mentor Graphics. *Seamless Co-Verification Environment*. http://www.mentorg.com/seamless/.

[17] P. Pop, P. Eles, and Z. Peng. Bus access optimization for distributed embedded systems based on schedulability analysis. In *Proc. Design, Automation and Test in Europe (DATE 2000)*, Paris, France, 2000.

[18] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proc. of Design, Automation and Test in Europe Conference (DATE'02)*, Paris, France, March 2002.

[19] K. Richter, D. Ziegenbein, R. Ernst, L. Thiele, and J. Teich. Model composition for scheduling analysis in platform design. In *submitted to Proceeding 39th Design Automation Conference*, New Orleans, USA, June 2002.

[20] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[21] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, 1989.

[22] Synopsys. *CoCentric System Studio*. http://www.synopsys.com/products/cocentric_studio/.

[23] L. Thiele, s. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors - models and algorithms. In *Proc. 1st Workshop on Embedded Software (EMSOFT)*, Lake Tahoe (CA), USA, October 2001.

[24] K. W. Tindell. An extendible approach for analysing fixed priority hard real-time systems. *Journal of Real-Time Systems*, 6(2):133–152, Mar 1994.

[25] F. Wolf and R. Ernst. Execution Cost Interval Refinement in Static Software Analysis. *The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design*, 47(3-4):339–356, April 2001.