Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Mathematics and Computers in Simulation 66 (2004) 231–242
MATHEMATICS AND COMPUTERS IN SIMULATION
www.elsevier.com/locate/matcom

Interval-based analysis in embedded system design

M. Jersak*, K. Richter, R. Ernst

Institut für Datentechnik und Kommunikationsnetze, Technische Universität Braunschweig, Hans-Sommer-Str. 66, D-38106 Braunschweig, Germany

Available online 6 January 2004

Abstract

Complex multi-processor systems-on-chip and distributed embedded systems exhibit a confusing variety of run time interdependencies. For reliable timing validation, not only application, but also architecture, scheduling and communication properties have to be considered. This is very different from functional validation, where architecture, scheduling and communication can be idealized.

To avoid unknown corner-case coverage in simulation-based validation on one had, and the state-space explosion or over-simplification of unified formal performance models on the other, we take a compositional approach and combine different efficient models and methods for timing analysis of single processes, real-time operating system (RTOS) overhead, single processors and communication components, and finally multiple connected components. As a result, timing analysis of complex, heterogeneous embedded systems becomes feasible.

Keywords: Real-time embedded systems; Performance verification; Interval analysis

1. Introduction

Embedded system platforms consist of a combination of different processor types, specialized memories, weakly programmable or fixed function components, and a communication infrastructure composed of busses, switches or point-to-point connections. Programmability and configurable software architectures are key to adapt platforms to a variety of applications. The verification of such platforms faces two main problems: on one hand verification of the system function, and on the other hand verification of the platform performance, i.e. the adherence to system timing constraints, memory requirements, or power consumption. For functional verification, different simulation-based and formal techniques already exist. This paper instead focuses on performance verification, where dealing with the additional complexity introduced through resource-sharing is key.

Compared to traditional, hardware-centric designs, embedded software adds a new level of complexity to embedded platforms. Processors, buses and other system resources are shared between different pro-

* Corresponding author. Tel.: +49-531-391-3729; fax: +49-531-391-4587.
E-mail address: marek@ida.ing.tu-bs.de (M. Jersak).

0378-4754/$30.00 © 2003 Published by Elsevier B.V. on behalf of IMACS.
doi:10.1016/j.matcom.2003.11.008

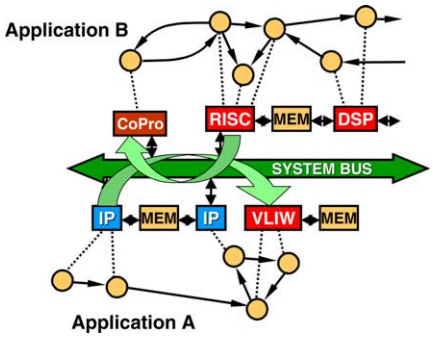

Fig. 1. Embedded multi-processor system-on-chip (MP-SoC) example.

cesses and logical channels. The underlying operating systems, which implement a variety of static and dynamic, time-driven and event driven, preemptive and non-preemptive scheduling strategies, introduce timing dependencies between functionally independent processes on a single resource. The complexity is once more substantially higher for heterogeneous multi-processor platforms. Here, several scheduling strategies can be combined in one system, e.g. a static schedule on a DSP and a priority-driven schedule on a micro-controller. Communication between processors adds to behavioral complexity by introducing additional resource-sharing. Communication buffering changes the memory requirements and can lead to internal event bursts. Finally, the integration of re-used and/or IP (intellectual property) components significantly complicates the performance verification process, since the implementation details of such "black-box" components are usually only partially known. An example of a complex Multi-Processor System-on-Chip (MP-SoC) with six processing cores and several memories including IP components is illustrated in Fig. 1. The example also shows how a shared system bus introduces complex non-functional run-time dependencies between two functionally independent applications.

The appearance of MP-SoC had a profound impact on timing validation. Never before, technical systems with similar heterogeneity and complexity had to be built with a comparable productivity. MP-SoC, composed in a system-level "cut-and-paste" approach, expose such a confusing variety of communication and run-time interdependencies, that it cannot be fully overseen by anyone in a design team. Both upper and lower run-time and communication bounds and their combinations contribute to this complex behavior. A best-case situation in one place, e.g. a minimum process response time, can lead to a worst-case situation in a different place, e.g. maximum load on a bus, resulting in worst-case delay of a lower-priority communication. Decades of work in real-time operating systems (RTOS) have revealed other types of non-intuitive run-time anomalies, e.g. [6,18]. The situation is aggravated with the integration of black-box IP, where corner cases are completely hidden from the integrator.

For performance validation, application, architecture, scheduling and communication properties with all their complex interdependencies need to be considered. This is very different from functional validation, where architecture, scheduling and communication properties are idealized. Simulation-based techniques for timing validation are increasingly unreliable with growing application and architecture complexity. Therefore, formal timing analysis techniques which consider conservative min–max behavioral intervals are becoming more and more attractive as an alternative or supplement to simulation.

Many existing unified representations for formal validation, e.g. temporal logic, timed process algebras or timed automata, do not consider the complex run-time interdependencies that arise from

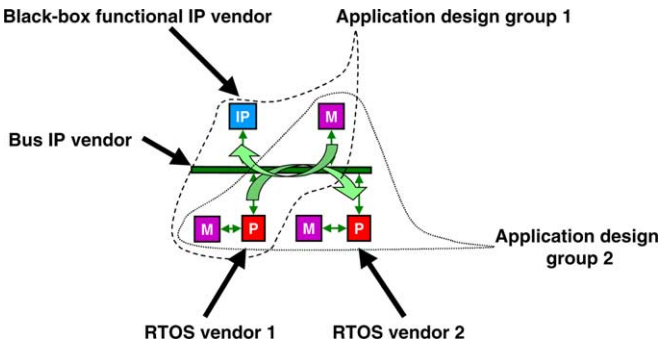

Fig. 2. Different system functions coming from different sources that have to be integrated. Performance has to be validated for the complete system.

resource-sharing and behavioral intervals, e.g. [1,3]. This renders such representations unsuitable if a realistic implementation has to be considered. Adequate extensions restrict unified approaches to small problems due to the exploding state-space [2].

To overcome these problems, we take a compositional approach. Instead of using a single, complex model, we compose different established models and methods. These problem- and community-specific models are successful in engineering because they are well-understood, optimized, and thus efficient in their respective domain. Although a large field of research, in practice we have not yet seen an indication that a single, super-model' is going to work. However, the models and methods that we compose share two key commonalties, namely abstraction and the use of conservative min–max intervals. Composability ensures a seamless design-flow, and allows to integrate black-box IP components (Fig. 2). A critical requirement is that results from conservative analyses of sub-components can be composed, in order to conservatively analyze the real-time performance of a larger component.

Parts of this ongoing work have been previously presented [8,12,14,17,19]. Here, for the first time, we present all key steps of our methodology (Section 2), namely single process performance modeling and analysis (Section 3), real-time operating system performance modeling and analysis (Section 4), single component schedulability analysis (Section 5) and analysis coupling for heterogeneous MP-SoC (Section 6). In each section, we emphasize the different models and techniques best suited for the corresponding analysis stage.

2. Methodology

In formal timing analysis, the goal is to obtain conservative execution time intervals for every system function, in order to validate timing constraints (end-to-end latencies, sample rates, maximum jitter, etc.). Two approaches can be safely combined:

- Timing of (sub)functions that is input-data independent, or for which worst- and best-case input data is known, can be measured, or simulated using a cycle-true simulator. This is both efficient and accurate.
- These lower-level timing intervals can then be used to perform min–max calculations for higher-level functions where the timing is input-data dependent and so complex that worst- and best-case input data is not known.

The basic idea of our methodology is to decompose the problem of system-level timing analysis in order to be able to use existing tools or approaches. The results of these tools are then combined in order to obtain system-level timing parameters such as end-to-end latencies. The decomposition divides the influences on system-level timing into several orthogonal classes. The first class is based on individual process analysis which assumes exclusive resource access for each process. The second class models the influence of the RTOS itself on timing. The third class is based on individual resource analysis which considers resource-sharing influences. The fourth class composes single-component analysis results into analysis of a heterogeneous multi-processor system, for which no consistent analysis exists. Our approach satisfies a key requirement, namely to compose results from conservative analyses of sub-components for conservative analysis of a larger component. This will be explained in the following sections. For each step, we will also summarize suitable models and methods.

3. Single process analysis

A process consists of a sequential program that is activated when its activation condition is met, and is then executed under RTOS control, until it completes. The goal of single process timing analysis is to determine the minimum and maximum execution time of one activation of a single process assuming an exclusive resource, i.e. after activation the process runs to completion without interrupts. The result obviously depends on the processor, on which the process is executed. Obtaining tight analysis bounds is challenging for complex processor architectures with pipelines and caches.

Recent analysis approaches, e.g. [11], first determine execution time intervals for each basic block (BB) by adding the cost of each instruction in the BB. Conservative overhead at BB boundaries is then added, e.g. to account for pipeline flushes after a jump. Using an integer linear programming (ILP) solver, a shortest and a longest path through the process is then found based on basic block execution counts and cost, leading to an execution time interval for the whole process. The designer typically has to bound loops and exclude infeasible paths to tighten the process-level execution time intervals.

For architectures with pipelines and caches, execution time intervals for basic blocks can be rather pessimistic because empty pipelines or cache flushes have to be assumed. On the other hand, if input data independent sequences of basic blocks (called process segments, e.g. a loop with a fixed number of iterations) and the resulting address access sequence are considered, then conservatism at BB boundaries can be reduced as shown in [17]. Tight execution time bounds for each segment can be obtained by executing the segment using a cycle-accurate simulator or a suitable measurement setup. Furthermore, established cache tracing techniques can be applied. The segment-based approach significantly reduces the problem size of previous approaches based on transition graphs for single basic blocks. For input data dependent control structures between process segments, data flow analysis can be applied to predict cache line contents. Cache analysis results can then be accounted for when calculating execution time intervals for process segments.

For the remaining data dependent control structures, execution time intervals are determined using the known technique from [11], but between process segments instead of basic blocks. Even larger segments, and thus narrower performance intervals, can be obtained, if system contexts are considered. An example is shown in Fig. 3, where context-dependent execution times and communication of an image filter are given. In this example, the architecture was a StrongArm-processor. Overall, our methodology can lead

Timing [ms] Received Data [kB] Sent Data [kB]		Address not considered	Address miss	Address match
Size not considered	T	[5 , 681]	[6 , 40]	[38 , 681]
	Rec	[6.2 , 25.0]	[6.2 , 25.0]	[6.2 , 25.0]
	Snt	[0 , 24.4]	[0 , 0]	[5.9 , 24.4]
Large Picture	T	[19 , 572]	[20 , 39]	[265 , 572]
	Rec	[25.0 , 25.0]	[25.0 , 25.0]	[25.0 , 25.0]
	Snt	[0 , 24.4]	[0 , 0]	[24.4 , 24.4]
Small Picture	T	[5 , 67]	[6 , 13]	[38 , 64]
	Rec	[6.2 , 6.2]	[6.2 , 6.2]	[6.2 , 6.2]
	Snt	[0 , 5.9]	[0 , 0]	[5.9 , 5.9]

Fig. 3. Context-dependent execution times and communication of an image filter.

to much tighter execution time intervals compared to previous approaches, as has been shown with the prototype tool SYMTA/P [17].

The illustrated single-process analysis approach is not easily applicable to IP components which are available as object code only. For the characterization of such “black-box” software IP components, the IP provider should (among other information) provide tight upper and lower execution-time bounds. She has the source code and can use the same analysis approaches, ideally considering relevant context information. This way, integrators have sufficient performance data for reliable integration while IP protection is preserved for the suppliers.

Models	Tools and techniques
Control- and data flow graph	Process segment detection Cycle-accurate processor simulator or measurement setup for process segment measurements
Process segment execution cost intervals	Integer linear program solver Cache line content prediction

4. RTOS analysis

Apart from influencing the timing of individual tasks through scheduling, the RTOS itself may consume a considerable amount of processor time. The RTOS has to activate tasks, make scheduling decisions, and to terminate tasks. Additionally, low-level device drivers, e.g. system timers and I/O, require processor time. Typical RTOS primitives are described, e.g. in [4]. The most important RTOS influences are: task or context switching including start/preemption/resumption/termination of tasks; and general OS overhead, including periodic timer interrupts and some house-keeping functions. For formal timing analysis to work, these numbers need to be considered in a conservative way.

In order to calculate RTOS overhead, execution time intervals for each RTOS primitive and their dependency on the number of tasks scheduled by the RTOS are required. Additionally, patterns in the execution of RTOS primitives have to be known, e.g. which primitives are executed when a higher priority task interrupts a lower priority task. Ideally, the RTOS vendor would appropriately characterize the timing

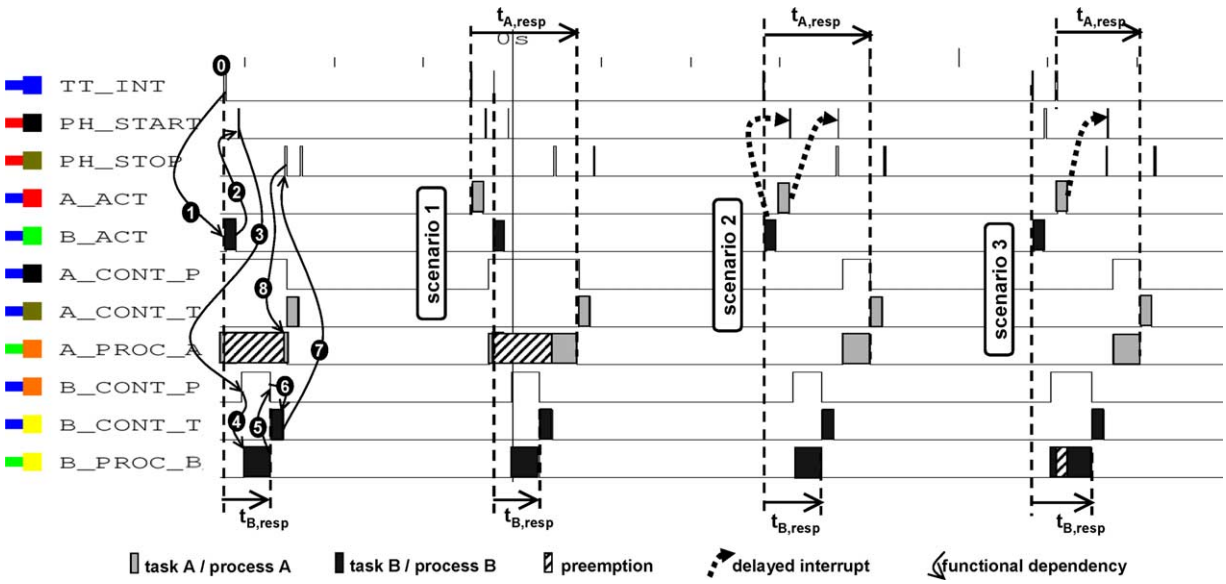

Fig. 4. RTOS influence on process response times.

of each RTOS primitive and provide rules to calculate RTOS overhead, or alternatively provide suitable tests such that the user can determine timing intervals for her architecture herself. As in Section 3, this allows IP protection and reliable integration at the same time. For the RTOS that we used, we had to perform these measurements ourselves, because conservative performance numbers were not provided by the RTOS vendor. Therefore, we cannot guarantee that we considered all corner cases. Ultimately, for safe performance analysis, this characterization will have to come from the RTOS vendor herself. Some vendors have taken first steps in this direction [13].

Fig. 4 shows an example execution trace of RTOS primitives which we obtained by monitoring the execution of an actual RTOS using a logic state analyzer (LSA). We measured:

- **TT_INT** TimeTable Interrupt: Executed whenever the time table needs to be evaluated to start a new task.
- **PH** PreemptionHandler: Executed whenever a task has to preempt the actually running one.
- **X_ACT** ActivateTask X: Executed whenever a task is activated (i.e. ready for execution).
- **X_CON_P** ProcessContainer: Each task X has a process container that subsequently calls all processes within this task.
- **X_CON_T** TerminateTask: Executed after task X has finished.
- **X_PROC_Y** Process: This is the actual user process Y within task X.

A sample pattern of RTOS primitives that is traversed for one execution of one task is shown on the left side in the LSA printout in Fig. 4. A system timer generates a time-table interrupt (step 0). This selects and activates the corresponding tasks by executing the ActivateTask function (step 1). Although not visible in the source code, based on the LSA output it seems that this in turn generates a software interrupt which then starts the preemption handler. Since all tasks seem to be using the same preemption handler, we distinguish between start and stop using two different I/O signals in order to record recursive interrupts.

The preemption handler calls the dispatcher, which seems to start the task, i.e. the process container (step 3), which finally executes the process (step 4). After completion, control returns to the process container (step 5), and since there is only one process in the task, the TerminateTask function is executed (step 6). Finally, the preemption handler finishes (step 7) and a preempted task is resumed (step 8).

In Fig. 4, we also show three different scenarios for task interference. Depending on the order in which tasks A and B are activated by TT_INT (A before B in scenario 1, A and B at the same time in scenario 2, A after B in scenario 3), the tasks experience different delays. Task A is the lower-priority task, task B the higher-priority task. One interesting observation is that preemption handlers seem to have the same priority as their corresponding task, since task A's preemption handler is delayed until the completion of task B in scenarios B and C. Another interesting observation is that the ActivateTask functions have a higher priority than all tasks. Therefore, the activation of lower-priority task A can delay or preempt the execution of higher-priority task B (scenarios 2 and 3, respectively). These observations show that it is important to model the RTOS at such a detailed level to reliably calculate its influence of task response times. In general it should be possible to provide formulas for these calculations.

Models	Tools and techniques
RTOS model with RTOS primitives, RTOS vendor performance benchmark suite	Cycle-accurate processor simulator or measurement setup for RTOS primitives measurements
RTOS primitives cost and execution patterns	Cost and patterns will be considered in single component analysis

5. Single component analysis

A large amount of work exists that analyzes real-time performance of singe components, where a set of processes with certain properties is scheduled by an RTOS, or a set of logical communication channels is scheduled on a shared communication resource. These approaches capture system timing in a closed form using timing equations and appropriate solution algorithms which reflect the used resource-sharing (scheduling) strategy. Regardless of the actual scheduling strategy, all formal scheduling analysis techniques share some key properties. First, they are based on process core execution times and RTOS overhead numbers; upper and lower bounds are provided by the analyses introduced in the previous two sections. Secondly, the analysis techniques provide response times for the processes, i.e. the time between the activation of a process until the time of completion. And thirdly, they do so by formally capturing and evaluating information about the external signals which activate the execution of processes. On of the first approaches is the work by Liu and Layland who proposed a preemptive priority-driven scheduling to guarantee deadlines for periodic hard real-time processes [10]. They considered a static (Rate Monotonic) and a dynamic (Earliest Deadline First) priority assignment and provided a formal analysis framework for both. Fig. 5 shows the influence of scheduling on task response times for three tasks scheduled with a preemptive static priority scheduler and a rate monotonic priority assignment. Note that for all but the highest priority task, the response time is an interval. Consequently, even though task are activated exactly periodically, the response jitters within an interval of length $\max(t_{\text{response}}) - \min(t_{\text{response}})$.

An overview of existing techniques can be found, e.g. in [4,5] Every scheduling analysis technique makes assumptions about the component environment which generates the component's workload. These

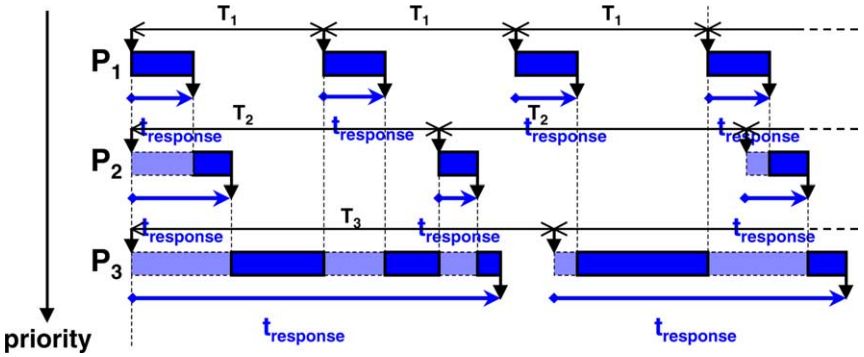

Fig. 5. Response time intervals for processes scheduled with static priorities.

assumptions are the event frequency, and how much data is provided per event. One can distinguish different types of events, such as the arrival of different packet types (control or data) in a packet network. The event frequency, the event types and the distribution in time are the main environment parameters. Typically, the numerous input event models used in practice are classified into four classes (e.g. [12]):

- *Periodic*: events arrive in equidistant periods of time (T)
- *Periodic with jitter*: here, the events arrive generally periodic with the period T. However, each event may be delayed within a so called jitter window of size J.
- *Periodic with burst*: here, n (burst size) events with a minimum distance of t (minimum inter-arrival time) arrive within a period T
- *Sporadic*: in contrast to all other models, sporadic events—either bursty or not—are not generally periodic. Therefore, only a minimum inter-arrival time t between two successive events or successive bursts is given.

While earlier work assumed periodic activation of processes, more recent extensions allow periodic activation with jitter, e.g. [15], and arbitrary deadlines and burst [9] for static-priority scheduling. The work by Sprunt et. al. is one example where the influence of sporadic process activation is analyzed.

Event models can be nicely represented as integrals over time using event model functions [7] or *arrival curves* [16]. Time integrals yield intuitive graphical representations and fit the load analysis approach to scheduling analysis, but finding a closed form can be difficult in the general case. One approach presented in [16] is to introduce a new analysis technique, where the arrival curves are approximated with piecewise linear functions for the maximum and minimum number of arriving events. In our approach, if necessary, we conservatively bound more complex arrival curves with simpler arrival curves that correspond to the event models used in classical real-time analysis. This way, we are able to make use of existing powerful analysis techniques that have been developed in the field of real-time analysis.

Definition: For any Δt, the upper arrival curve is a tight upper bound for the number of events that can arrive during any interval of length Δt, while the lower arrival curve is a tight lower bound for the number of events that must arrive during any interval of length Δt.

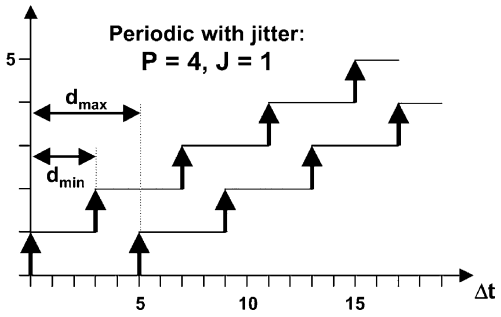

Fig. 6. Upper and lower arrival curves for periodic with jitter event model.

In Fig. 6, upper and lower arrival curves are shown for the *periodic with jitter* event model with $P = 4$, $J = 1$. Also shown is the minimum and maximum distance between two events.

Models	Tools and techniques
Single process performance models	
RTOS performance models	
Activating event models	Schedulability analysis

6. Communicating component analysis

A process on one resource generates output events which are input to another resource. These events can also be captured using event models, as explained in the previous section. In [14], we provided rules to derive output event models from the already known performance parameters (input event models and response times). Furthermore, we showed that these output event models stay within the four input event model classes introduced above. We can thus couple the performance analyses for individual sub-systems by propagating event models through the architecture components along paths of process dependencies. An output event model of one component becomes an input event model for the connected component. This leads to an iterative procedure.

While this is comparatively simple for feed-forward systems, it is more complex for systems with cyclic event model dependencies. In the presence of process dependency loops, we have no dedicated starting point where all event models are given. Rather, we have to make assumptions about the actual parameters of certain input event models to start the iterative procedure. Furthermore, we have no dedicated termination point comparable to the system outputs in purely feed-forward systems. Therefore, we have to iterate until all event models either converge or diverge. Convergence means that we have found a valid resource interaction abstraction. Divergence means that such an abstraction cannot be found.

Cyclic process dependencies are not the only source for cyclic event model dependencies. Due to resource-sharing influences, also functionally independent processes may influence each others timing, and thus each others output event models. In certain cases, the superposition of process dependencies and

scheduling dependencies can result in additional cyclic dependencies of the corresponding event models. However, the impact on the overall analysis problem is the same as before.

The integration of IP components represents an additional challenge with respect to this iterative design/analysis flow. The integrator has to request performance data for a component each time the input event models change. Most elegantly, the IP provider would ship an appropriate analysis module that performs the component analysis at the integrators site together with the IP component. This module would be IP protected, too.

In [12], we presented an approach to interface between different event models, which we call event model interfaces (EMIFs). An EMIF transforms the representation of an event stream from one event model into the abstract parameters of another event model. For instance, a periodic event stream with the period T_X can be captured by the burst event model, if we set the burst length to $b_Y = 1$, the outer burst period to $T_Y = T_X$, and the minimum inter-arrival time to $t_Y = T_X$.

Note that EMIFs do not modify the actual event streams, rather they transform the abstract representation of a single event stream. The transformations are uni-directional, and can not be found for all combinations of event models. For instance, an event stream with burst cannot be captured by the parameters of a purely periodic event model. In this case, we need to adapt the event stream itself. This can be done by adding functionality to the system, which we call event adaptation functions (EAF). A periodic event stream with jitter can be re-synchronized by means of a buffer with a periodic output issue rate. The same applies to the re-synchronization of event streams with burst. For detailed information about EMIFs and EAFs and the corresponding formalisms, we refer to [12]. There, we also derive worst-case buffer sizes for the EAFs and additional event delays which result from re-synchronization.

This interfacing of and transitioning between different event models allows to couple the individual components in a way that enables global analysis of the entire system. This can be nicely seen in the example system in Fig. 7. Before integration, the event or data flow from P3 to P4 was purely periodic. Both components were fine-tuned to a previously negotiated communication timing. This way, the timing correctness of the subsystem can be verified. The OS scheduling on each of the components was verified using existing real-time analysis techniques. This is a typical subsystem design scenario. Now—after integration—bus contention might require data to be temporarily buffered before it can be sent over the bus. Due to these usually time-variant delays, the events which arrive at P4 are not purely periodic anymore.

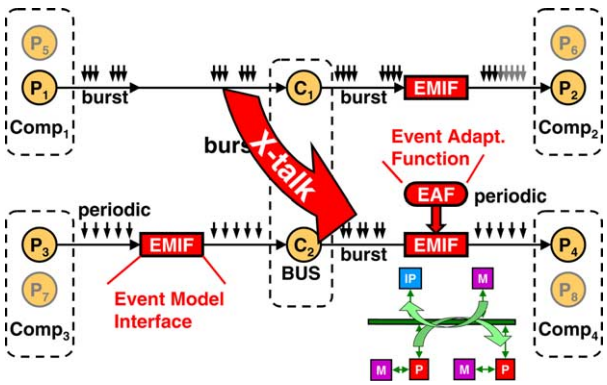

Fig. 7. Application of event model interfaces (EMIFs) and event adaptation functions (EAF) for analysis coupling, and an example for performance cross-talk.

Thus, the initial negotiation is violated, the timing is unknown, and correctness can not be guaranteed. In order to allow a safe integration of subsystems with different communication characteristics, we need to solve two problems.

The first task is to analyze the bus or network influence on the timing of communicated data. Using EMIFs and EAFs, we can adapt the given event streams to the event model required to analyze the bus, i.e. in the example periodic with burst. Then, we can analyze the network load and network latency. Furthermore, we can safely determine the required buffer sizes and possible buffering delays at the network input (the output of P1 and P3). In effect, the aggregated latencies determined the timing distortion at the other end of the bus, i.e. the input of P4. In our example, we have two streams interfering on the bus. In effect, the delay of one event stream results from the characteristics of another event stream, and vice versa. Qualitatively, the periodic events from P3 are "modulated" by the bursts coming form P1, while the P1 bursts are distorted by the periodic P3 events. In other words, both streams "cross-talk" with respect to event timing. And the analysis allows us to quantitatively determine the actual congestion-related delay.

Now, we know the influence of integration on the two communication links, and we can tackle the second problem. We need to find appropriate network or bus interfaces that allow integration without violating the existing subsystems timing requirements. Again, the EMIFs/EAFs help to apply transformation between models. This way, the original stream characteristics can be recovered at the network output, e.g. to resynchronize the bursty into a periodic stream at P4's input.

In general, we have to establish and solve the model relation between the network input event stream, the network analysis, and the required output model. Thus, different models and streams might require different interface components. While re-synchronization is absolutely necessary for P4's input, it is not for the input of P2. Here, a model of less accuracy is required, and we can conservatively derive its parameters from the bus output. In both cases, we obtain a "transparent" network, i.e. the connected components do not really "see" the network with respect to the required models, and the input/output timing behavior does not significantly change.

Models	Tools and techniques
Calculated output event models	Event model interfaces and adaptation
Required input event models	Event model propagation

7. Conclusion

In complex, realistic MP-SoC design, a single model is not sufficient to perform reliable timing analysis. Existing unified models either suffer from state-space explosion or from over-simplification. However, there exist well accepted techniques for the analysis of sub-problems. Each technique uses a specific model best suited to the problem at hand, thereby allowing very efficient solutions. A key challenge is to propagate and interface the design data through all these tools in an efficient way, while also considering IP protection requirements. In this paper, we gave an overview on models and techniques for task-, component-, and system-level performance analyses is MP-SoC design. We showed how the techniques can be applied and coupled, and how the corresponding design information can be safely exchanged in on order to facilitate efficient and reliable system-level performance verification. Abstraction and the use of intervals play a key role to obtain reliable timing analysis results.

References

[1] R. Alur, C. Courcoubeti, D. Dill, Model-checking in dense real-time, Inform. Comput. 104 (1) 1993.
[2] K. Altisen, G. Goessler, J. Sifakis, Scheduler modeling based on the controller synthesis paradigm, J. Real-Time Syst. 23 (2002).
[3] J.C.M. Baeten, J.A. Bergstra, Discrete time process algebra, Formal Asp. Comput. 8 (2) (1996) 188–208.
[4] G. Buttazzo, Real-Time Computing Systems—Predictable Scheduling Algorithms and Applications, Kluwer Academic Publishers, 2002.
[5] C.J. Fidge, Real-time schedulability tests for preemptive multitasking, J. Real-Time Syst. 14 (1998) 61–93.
[6] R.L. Graham, Bounds on multiprocessing anomalies, SIAM J. Appl. Math. 17 (1969) 263–269.
[7] K. Gresser, Echtzeitnachweis ereignisgesteuerter Realzeitsysteme, Fortschrittsberichte VDI, Reihe 10, Nr. 268, VDI Verlag, Düsseldorf, 1993.
[8] M. Jersak, K. Richter, R. Henia, R. Ernst, F. Slomka, Transformation of SDL specifications for system-level timing analysis, in: Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES02), Estes Park, CO, USA, 2002.
[9] J. Lehoczky, Fixed priority scheduling of periodic task sets with arbitrary deadlines, in: Proceedings of the Real-Time Systems Symposium, 1990.
[10] C.L. Liu, J. Layland, Scheduling algorithm for multiprogramming in a hard-real-time environment, J. ACM 20 (1) (1973) 46–61.
[11] Y.S. Li, S. Malik, Performance Analysis of Real-Time Embedded Software, Kluwer Academic Publishers, 1999.
[12] K. Richter, R. Ernst, Event model interfaces for heterogeneous system analysis, in: Proceedings of the 5th Design, Automation and Test Conference (DATE02), Paris, France, 2002, pp. 506–513.
[13] K. Tindell, H. Kopetz, F. Wolf, R. Ernst, Safe automotive software development, in: Proceedings of the Design, Automation and Test in Europe (DATE'03), Munich, Germany, March 2003.
[14] K. Richter, D. Ziegenbein, M. Jersak, R. Ernst, Model composition for scheduling analysis in platform design, in: Proceedings of the 39th Design Automation Conference (DAC'02), New Orleans, LA, USA, 2002.
[15] L. Sha, R. Rajkumar, S.S. Sathaye, Generalized rate-monotonic scheduling theory: a framework for developing real-time systems, in: Proceedings of the IEEE, vol. 82, no. 1, pp. 86–82, 1994 [SSL89]. B. Sprunt, L. Sha, J. Lehoczky, Aperiodic Task Scheduling for Hard Real-Time Systems, J. Real-Time Syst. 1 (1) (1989) 27–60.
[16] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, J. Greutert, Embedded software in network processors—models and algorithms, in: Proceedings of the First Workshop on Embedded Software (EMSOFT), Lake Tahoe, USA, 2001.
[17] F. Wolf, Behavioral Intervals in Embedded Software, Kluwer Academic Publishers, 2002.
[18] T. Yen, W. Wolf, Performance Estimation for Real-Time Distributed Embedded Systems, IEEE Trans. Parallel Distributed Syst. 9 (1998).
[19] D. Ziegenbein, M. Jersak, K. Richter, R. Ernst, Breaking down complexity for reliable system-level timing validation, in: Ninth IEEE/DATC Electronic Design Processes Workshop (EDP'02), Monterey, USA, April 2002.