A Formal Approach to MpSoC Performance Verification

A new technology uses event model interfaces and a novel event flow mechanism that extends formal analysis approaches from real-time system design into the multiprocessor system on chip domain.

Kai Richter
Marek Jersak
Rolf Ernst
Technical University of Braunschweig

Multiprocessor system on chip designs use complex on-chip networks to integrate multiple programmable processor cores, specialized memories, and other intellectual property (IP) components on a single chip. MpSoCs have become the architecture of choice in industries such as network processing, consumer electronics, and automotive systems. Their heterogeneity inevitably increases with IP integration and component specialization, which designers use to optimize performance at low power consumption and competitive cost.

Figure 1 shows an example MpSoC, the Viper processor for multimedia applications.[1] Based on the Philips Nexperia platform, it includes many key components that are either reused or supplied externally, such as the MIPS and TriMedia processor cores. Tomorrow's MpSoCs will be even more complex, and using such IP library elements in a "cut-and-paste" design style is the only way to reach the necessary design productivity.

Systems integration is becoming the major challenge in MpSoC design. Complex hardware and software component interactions pose a serious threat to all kinds of performance pitfalls, including transient overloads, memory overflow, data loss, and missed deadlines. The *International Technology Roadmap for Semiconductors, 2001 Edition*, (http://public.itrs.net/files/2001itrs/design.pdf) names system-level performance verification as one of the top three codesign issues.

PERFORMANCE SIMULATION: CAN IT GET THE JOB DONE?

Simulation is state of the art in MpSoC performance verification. Tools such as Mentor Graphics' Seamless-CVE or Axys Design Automation's MaxSim support cycle-accurate cosimulation of a complete hardware and software system. The cosimulation times are extensive, but developers can use the same simulation environment, simulation patterns, and benchmarks in both function and performance verification. Simulation-based performance verification, however, has conceptual disadvantages that become disabling as complexity increases.

MpSoC hardware and software component integration involves resource sharing that is based on operating systems and network protocols. Resource sharing results in a confusing variety of performance runtime dependencies. For example, Figure 2 shows a CPU subsystem executing three processes. Although the operating system activates P_1, P_2, and P_3 strictly periodically (with periods T_1, T_2, and T_3, respectively), the resulting execution sequence is complex and leads to output bursts.

As Figure 2 shows, P_1 can delay several executions of P_3. After P_1 completes, P_3—with its input buffers filled—temporarily runs in burst mode with the exe-

Published by the IEEE Computer Society
0018-9162/03/$17.00 © 2003 IEEE

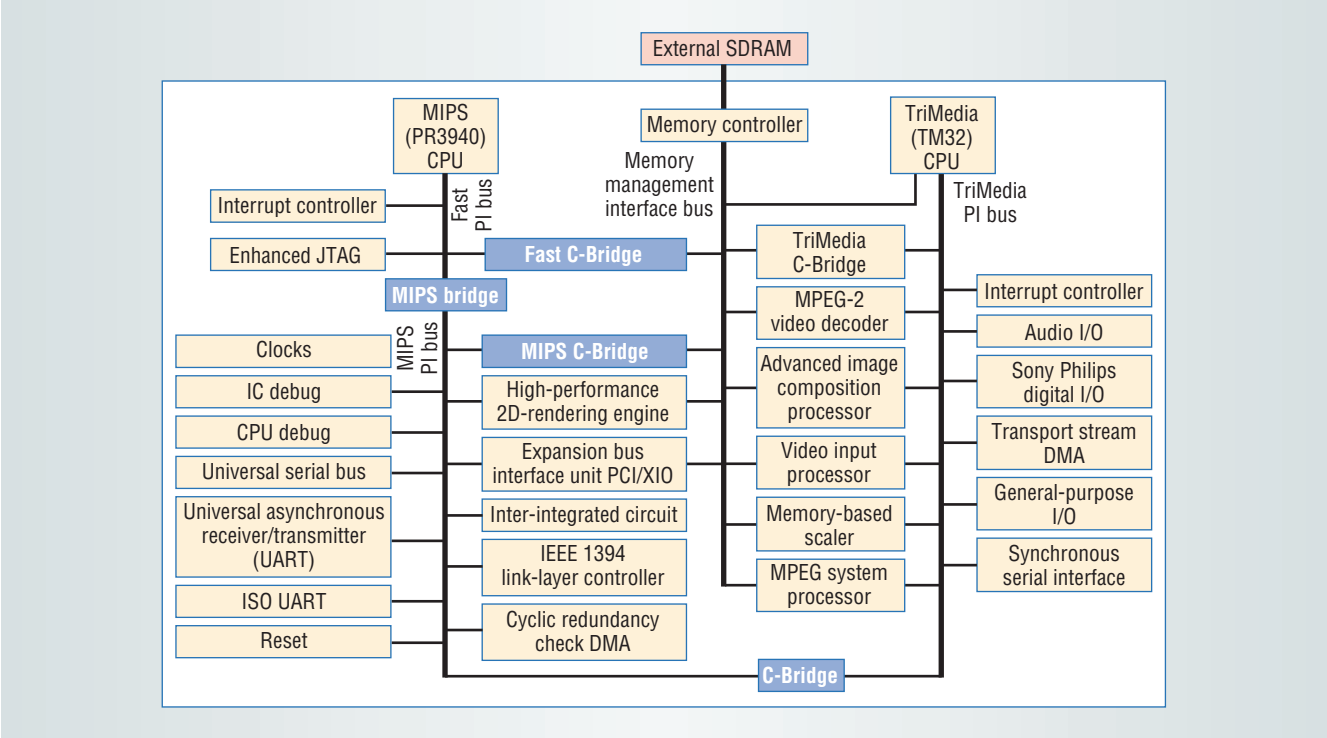

Figure 1. The Viper processor combines a MIPS RISC processor, a TriMedia TM32 VLIW DSP, weakly programmable coprocessors, and fixed function coprocessors, as well as various memories and caches omitted in the figure. A complex network of bridged high-speed and peripheral buses connects these components.

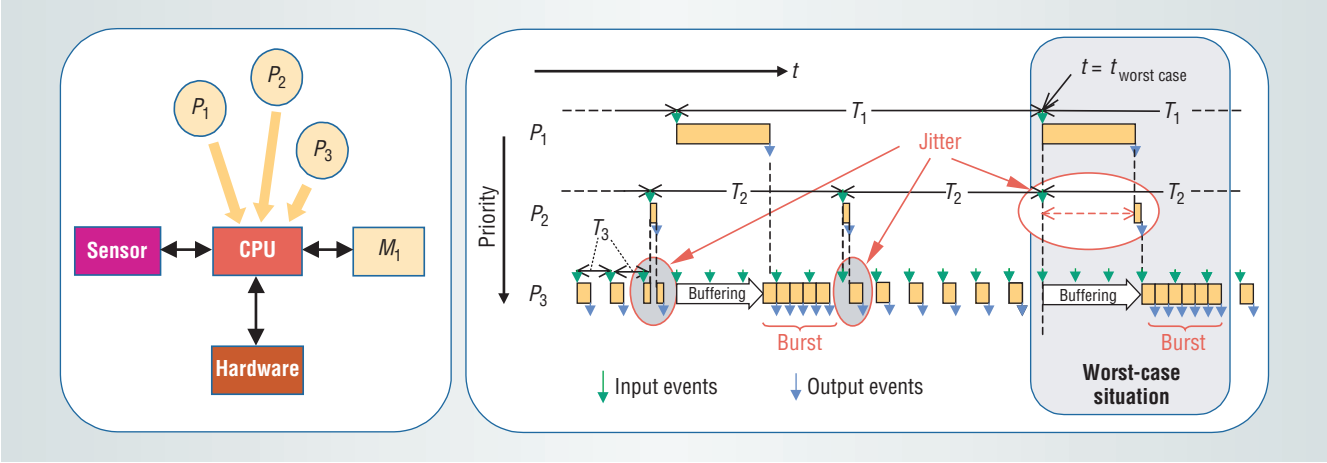

Figure 2. CPU subsystem with three tasks that are scheduled periodically. Scheduling is preemptive and follows static priorities. The highest-priority process P_1 preempts P_2 and P_3, resulting in a complex execution scenario exhibiting jitter and burst process outputs.

cution frequency limited only by the available processor performance. This leads to transient P_3 output burst, which is modulated by P_1's execution.

Figure 2 does not even include data-dependent process execution times, which are typical for software systems, and operating system overhead is neglected. Both effects further complicate the problem. Yet finding simulation patterns—or use cases—that lead to worst-case situations as highlighted in Figure 2 is already challenging.

Network arbitration introduces additional performance dependencies. Figure 3 shows an example. The green arrows indicate performance dependencies between the CPU and DSP subsystems that the system function does not reflect. These dependencies can turn component or subsystem best-case performance into system worst-case performance—a so-called *scheduling anomaly*. Recall the P_3 bursts from Figure 2 and consider that P_3's execution time can vary from one execution to the next. There are two critical execution scenarios, called *corner cases*: The minimum execution time for P_3 corresponds to the maximum transient bus load, slowing down other

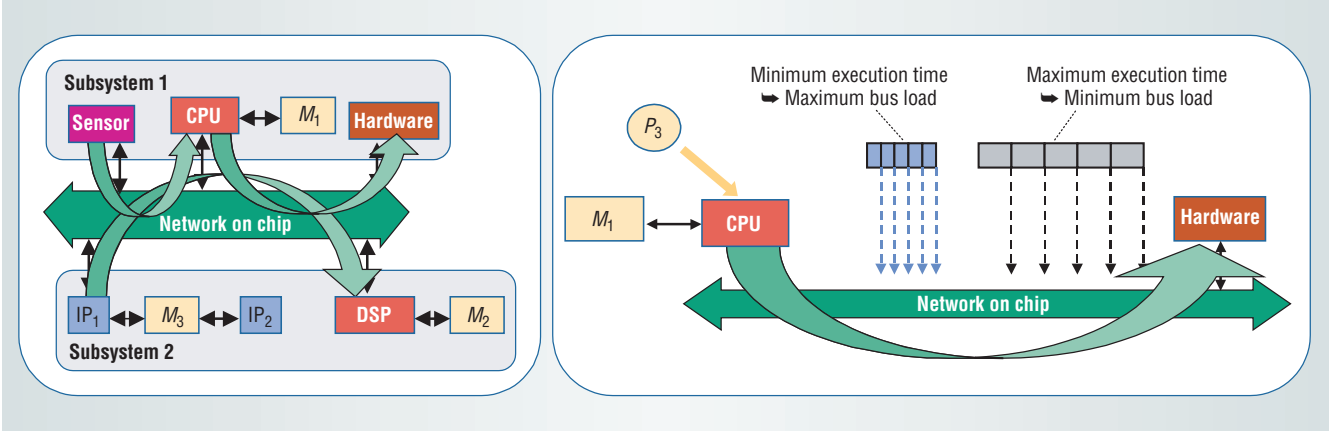

Figure 3. Scheduling anomaly resulting from a performance dependency between the CPU and DSP subsystems. P_3 is a driver process reading data from M_1. During bursts, P_3 iterates at maximum speed. Because P_3 also has a nonconstant execution time, the shorter the process execution time, the shorter the distance that packets travel over the bus and the higher the transient bus load.

components' communication, and vice versa.

The transient runtime effects shown in Figures 2 and 3 lead to complex system-level corner cases. The designer must provide a simulation pattern that reaches each corner case during simulation. Essentially, if *all* corner cases satisfy the given performance constraints, then the system is guaranteed to satisfy its constraints under all possible operation conditions. However, such corner cases are extremely difficult to find and debug, and it is even more difficult to find simulation patterns to cover them all.

Abstract and scheduling-aware performance simulation tools, such as Cadence VCC, can provide quick, rough estimates of average system performance but do not help in reliably covering system-level corner cases. The core problem is that VCC only uses typical execution times rather than considering intervals defined by the best-case and worst-case bounds.

Simulation-pattern generation

Where do we get the stimuli to cover system-level corner cases like those in Figure 3? Reusing function verification patterns is not sufficient because they do not cover the complex nonfunctional performance dependencies that resource sharing introduces. Reusing component and subsystem verification patterns is not sufficient because they do not consider the complex component and subsystem interactions.

The system integrator might be able to develop additional simulation patterns, but only for simple systems in which the component behavior is well understood. Manual corner case identification and pattern selection is not practical for complex MpSoCs with layered software architectures, dynamic bus protocols, and operating systems.

In short, it is becoming quite clear that today's simulation-based approaches to MpSoC performance verification will run out of steam soon.

Industrial consequences

As embedded system design style gradually moves from core-centric SoCs to communication-centric MpSoCs, the demands for flexibility and on-chip interconnect reactivity also increase. To meet these requirements at acceptable cost, some researchers have proposed sophisticated multihop dynamic communication network protocols, optimized using communication statistics.[2]

In practice, an opposite development has emerged, toward conservative and less efficient communication protocols like time division multiple access (TDMA) that minimize nonfunctional component dependencies like those in Figure 3. The Sonics SiliconBackplane MicroNetwork is an example of these protocols, which make communication timing straightforward and predictable. Distributed systems exhibit a similar trend toward conservative protocols, such as the time-triggered protocol in automotive and aerospace electronics.

Conservative protocols enforce static bus-access patterns and thus support independent verification of each communication's runtime behavior. This integration simplicity, however, comes at a significant performance price—a price that increases with system complexity. Buffer sizing requirements increase, along with response times. In addition, such protocols do not adapt to the dynamically changing load situations that are typical for reactive embedded systems. The conservative approach will therefore not scale well to future communication-centric MpSoCs with complex network protocols.

FORMAL TECHNIQUES: A PROMISING ALTERNATIVE

When simulation falls short, formal approaches become more attractive, offering systematic verification based on well-defined models. Formal analysis guarantees full performance corner-case coverage and bounds for critical performance parameters. The example of hardware circuit verifica-

tion, for which industry has widely adapted formal techniques as a supplement to hardware simulation, shows the practicability of formal approaches.

On the system level, the formal performance or timing verification literature distinguishes two problem areas:

- formal process and task performance analysis, usually in the form of process execution time analysis; and
- resource-sharing analysis, also known as scheduling or schedulability analysis, which is based on process execution times.

Process execution time analysis

Formal process execution time analysis has roots both in real-time system analysis for software processes and in hardware/software codesign for rapid hardware performance estimates. It includes two parts:

- program path analysis to find out what is going to execute; and
- architecture modeling, including pipelines and caches, to determine the execution time spent on this path.

This field has progressed enormously over the past 10 years and includes large-scale industrial applications in the aircraft industry, such as the AbsInt tool (www.absint.com), which calculates the execution time of C-processes using a technique called abstract interpretation. Details of process execution time analysis are beyond the scope of this article, but the technology currently provides conservative upper and lower bounds (intervals) for individual process execution times as well as bounds for the communication between processes.[3] These bounds lay the foundation for the scheduling analysis on shared resources.

Scheduling analysis

Real-time systems research has addressed scheduling analysis for processors and buses for decades, and many popular scheduling analysis techniques are available. Examples include rate-monotonic scheduling and earliest deadline first, using both static and dynamic priorities;[4] time-slicing mechanisms like TDMA or round-robin;[5] and static order scheduling.[6] Many extensions have found their way into commercial analysis tools, such as TriPacific's Rapid RMA, LiveDevices's Real-Time Architect, and TimeSys's TimeWiz.

The techniques rely on a simple yet powerful abstraction of task activation and communication. Instead of considering each event individually, as simulation does, formal scheduling analysis abstracts from individual events to event streams. The analysis requires only a few simple characteristics of event streams, such as an event period or a maximum jitter. From these parameters, the analysis systematically derives worst-case scheduling scenarios, which safely bound the worst-case process or communication response times.

Instead of considering each event individually, as simulation does, formal scheduling analysis abstracts from individual events to event streams.

The event-stream model displays the consequences of resource sharing, as Figure 2 shows. The scheduling transforms a periodic input event stream (process activation) into an event stream with burst at the component outputs. A larger system with more priority levels generates even more complex event sequences.

TAMING EVENT STREAM COMPLEXITY

Complex runtime interdependencies can change the event stream characteristics from component inputs to component outputs. In Figure 3, for example, the CPU output event stream is no longer exactly periodic. It propagates into the network input event stream, where communication scheduling on the shared network distorts it further.

To solve the system-level performance verification problem, we could look for analysis techniques that deal with the input event streams at hand and propagate them through the component network. However, protocols and corresponding analysis techniques that can handle such complex input event streams efficiently are rare. Furthermore, they produce even more distorted output streams, which then enter the connected receivers, and so on. The event sequences in the corresponding merged communication streams quickly become too complex for existing scheduling and analysis techniques.

In effect, global scheduling analysis of complex systems is currently not possible, since designers cannot reasonably combine the known subsystem techniques, mainly due to input-output event stream incompatibilities. A few "holistic" analysis approaches provide solutions for special classes of distributed systems,[7-8] but their scope and scalability are limited.

Recently, a different view of global scheduling analysis has emerged. The individual components and subsystems are seen as entities that interact, or communicate, via event streams. Schedulability analysis then becomes a flow-analysis problem for event streams that, in principle, event stream prop-

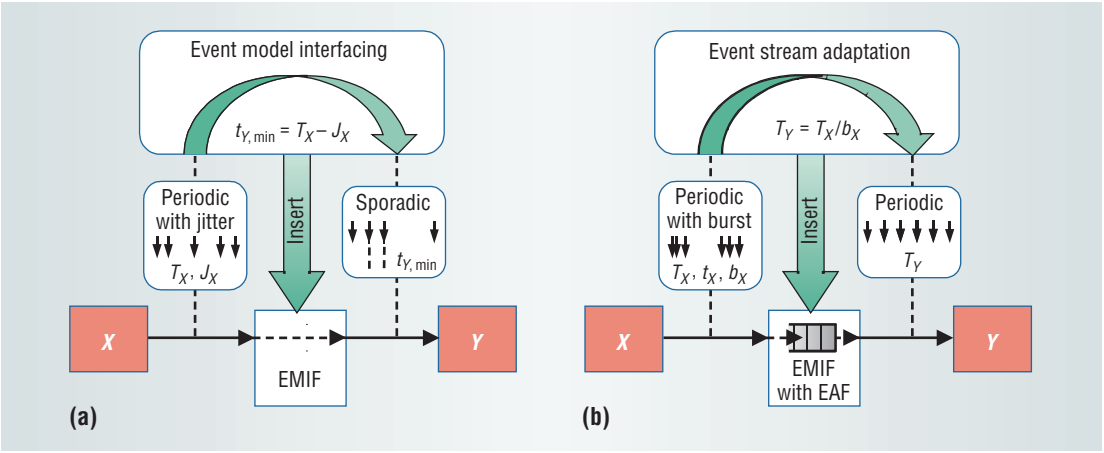

Figure 4. Event stream interfacing. (a) An event model interface (EMIF) transforms periodic events with jitter into the sporadic event model, and (b) an event adaptation function (EAF) adapts periodic bursts into the purely periodic event stream.

agation can solve iteratively. One approach to taming event stream complexity is to generalize the event model either in an event vector system[9] or with upper- and lower-bound arrival curves.[10] However, both these approaches introduce a new event stream representation and thus require new scheduling analysis techniques for the local components.

However, we don't necessarily need to develop new local analysis techniques if we can benefit from the host of work in real-time scheduling analysis. For example, in Figure 2, even if input and output streams seem to have totally different characteristics, the number of P_3's output events can be easily bounded over a longer time interval. The bursts only occur temporarily, representing a transient overload within a generally periodic event stream. In other words, some key characteristics of the original periodic stream remain even in the presence of heavy distortion.

We have developed a technology that lets us extract this key information from a given schedule and automatically interface or adapt the event stream so that designers and analysts can safely apply existing subsystem analysis techniques.

EVENT STREAM INTERFACING

Figure 4a shows a relatively simple event stream interfacing scenario, converting a periodic event stream with jitter into a sporadic stream. Some analysis techniques need this transformation, which requires only a minimum of math. The jitter events are characterized by a period (T_X) and a jitter (J_X). The jitter bounds the maximum time deviation of each event with respect to a virtual reference period. In other words, each event can be at most $J_X/2$ earlier or later than the reference event.

The required sporadic input event model has only one parameter, the minimum interarrival time ($t_{Y,\min}$) between any two events in the stream, thus bounding the maximum transient event frequency.

Imagine two successive events in the original stream, the first being as late as possible ($t + J_X/2$) and the second as early as possible ($t + T_X - J_X/2$). The minimum distance between two dedicated events in the output stream is thus $t_{Y,\min} = T_X - J_X$.

With respect to existing analysis techniques and practicably important input stream characteristics, we can identify two event model classes:

- *Periodic* event models are particularly useful when the event timing shows a generally periodic behavior, as in typical signal processing and control applications. Periodic events with jitter originate from a purely periodic source but the timing has been distorted by process preemption or network congestion, as in the case of P_2's output jitter in Figure 2. Jitters that exceed the original period lead to event bursts.
- *Sporadic* event models define situations in which the source is not periodic, such as seemingly irregular user-generated service requests. To conduct performance analysis, however, we need some information to bound the overall system load—for example, the minimum interarrival time between any two events. Besides the basic sporadic events model, we also model sporadic events with bursts that illustrate high transient event frequencies.

Our technology supports adaptation for all possible event model combinations, helping system integrators control complex event stream dependencies and understand and optimize the dynamic behavior of component interactions. We base the interfacing and adaptation on mathematical relations established between the involved streams or models.[11]

We can see the practical impact of event stream adaptation on system design and analysis in Figure 4a. The event stream itself—the timing properties of actual events—remains unchanged while only the mathematical representation—the underlying event model—is transformed. We refer to such transformations as event model interfaces (EMIFs). Such transformations require that the target model parameters directly encompass the timing of any event sequence possible in the source model.

If such direct model transformation is not possible, then our technology adapts the actual timing of

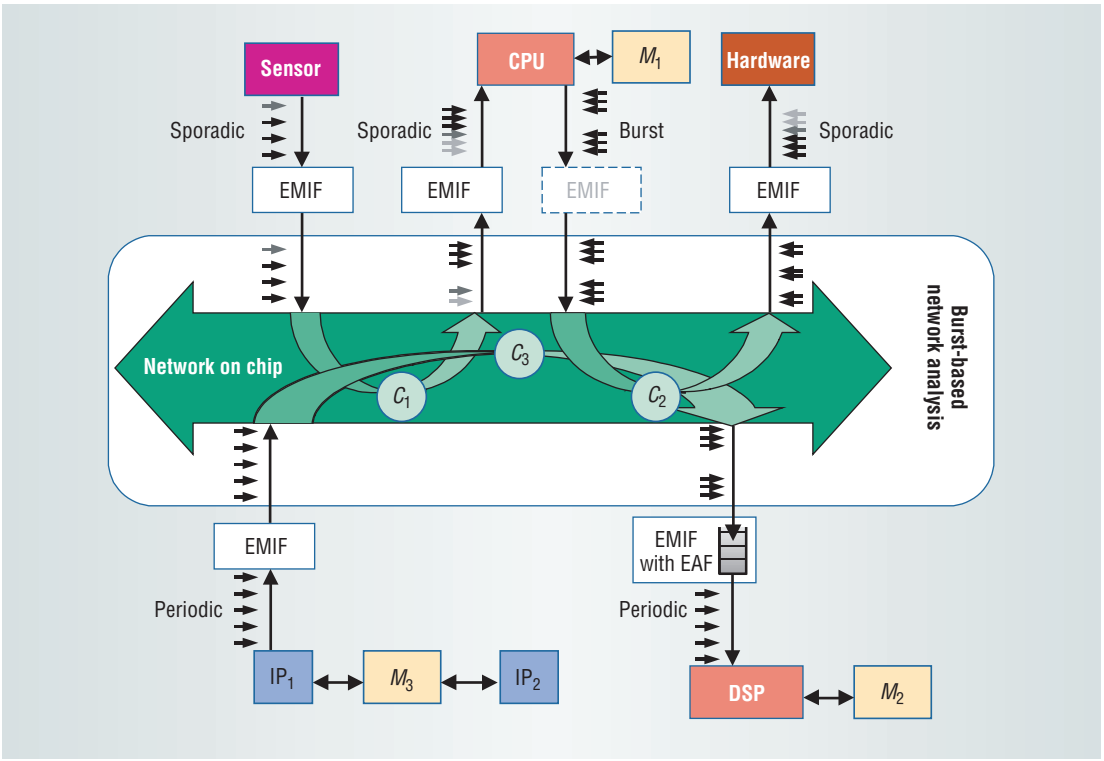

Figure 5. Event stream view of complex component interactions after integration. The merged network traffic on the logical channels C_1, C_2, and C_3 consists of periodic, bursty, and sporadic event streams. EMIFs and EAFs adapt the event streams to the components' requirements, so the designer can use known analysis techniques and efficient component implementations.

the stream events. An example is a periodic stream with jitter or burst that enters a component expecting purely periodic events. Figure 4b shows this situation, in which our technology automatically inserts an event adaptation function (EAF) in an EMIF to make the streams match. Practically speaking, EAFs correspond to buffers that are inserted at the component interface to make the system working and analyzable. Using EAFs, buffer sizing and buffering-delay calculation is automatically performed during adaptation. This is important in global system analysis.

Again, the math for the resynchronization shown in Figure 4b is relatively simple. The sought-after parameter T_Y of the purely periodic stream is the average period of the bursty stream, given by $T_Y = T_X/b_X$. The burst event model[7] captures a number of b_X events within a period of T_X.

Likewise, sophisticated interfaces and adaptations—possibly requiring appropriate design elements—are available for all other event model combinations. Furthermore, designers can easily extend the library of existing EMIFs and EAFs to other event models and streams to support the design of sophisticated high-performance subsystems or to integrate complex IP components. These EMIFs and EAFs form the foundation for a novel and very promising system-level performance analysis procedure.

INTERFACE AND PROPAGATE

We can now reliably verify the performance of the heterogeneous system in Figure 3. The sensor uses the logical channel C_1 to send new data sporadically to P_1, while P_3 sends bursts of requests through C_2 to the fixed-function hardware component. Simultaneously, the DSP subsystem is also using the network. IP_1 periodically sends data over channel C_3 to the DSP, which implements a periodic scheduling to efficiently run a set of signal processing applications.

The network can implement any protocol for which an appropriate analysis technique is available. This freedom widens the design space, because real-time analysis covers many network protocols, including complex dynamic arbitration schemes.

Assume that to use a known analysis technique, the input event streams to the network must comply with the model of periodic events with burst. Figure 5 illustrates the required EMIFs at the network inputs. Only the burst stream from the CPU already meets the required model; the other two input streams require an EMIF. Next, the designer can analyze the network using the known technique and obtain the distorted output event streams. In other words, the input streams are interfaced and propagated through the network analysis.

Finally, the output event streams are interfaced to the input models that the individual receiver components require. The DSP requires periodic input data to fit the given implementation of purely periodic scheduling. We already know this situation from Figure 4b, so we insert an EMIF with an appropriate EAF at the network output.

EMIFs and EAFs give system integrators tremendous assistance in system-level performance analysis for complex event stream dependencies spanning

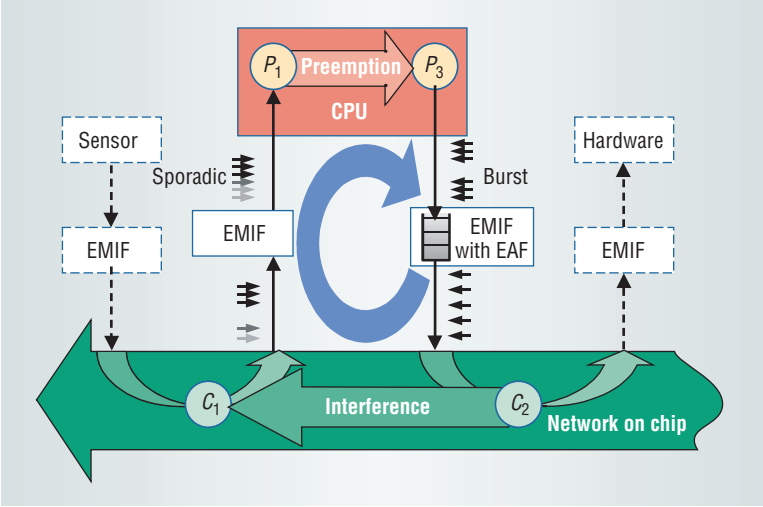

Figure 6. Cyclic event stream dependencies introduced in system integration and not reflected in the system function. Appropriate buffering when inserting EMIFs breaks up such cycles. Putting the buffers in the right place significantly improves system performance and memory optimization.

heterogeneous components. The required event models are either constrained by the local component and subsystem analysis techniques such as the network in Figure 5 or given by a component's implementation—for instance, the periodic DSP schedule or the maximum frequency (sporadic event model) of the hardware component.

CYCLIC EVENT STREAM DEPENDENCIES

An additional performance pitfall in MpSoC design occurs when cyclic event stream dependencies are introduced during system integration. These dependences are subtle and difficult to detect if the integration process does not consider component details. Figure 6 highlights a nonfunctional event stream dependency cycle in the system of Figure 5 that is only introduced by communication sharing. Upon receipt of new sensor data, the CPU activates process P_1, which preempts P_3 and thus affects the execution timing of P_3. Figure 2 illustrates this preemption.

P_3's output, in turn, enters the network on channel C_2, where it now interferes with the arriving sensor data on C_1. The interference of the two functionally independent channels, C_1 and C_2, closes the dependency cycle because the subsystem in Figure 2 was originally cycle-free.

Such cycles are analyzed by iterative propagation of event streams until the event stream parameters converge or until a process misses a deadline or exceeds a buffer limit. This iteration process terminates because the event timing uncertainty—that is, the best-case to worst-case event timing interval—grows monotonically with every iteration.

For cases in which no convergence occurs automatically, we have developed a mechanism that uses EAFs to break up the dependency cycle and enforce convergence by reducing the timing uncertainty. We have thoroughly investigated cyclic dependencies.[12] Note that the event flow cycles are not an artificial result of global analysis but exist in practice as the example demonstrates.

APPLICATIONS

We have defined the methods and developed a simple tool for analysis interfacing as well as a library with analysis algorithms for configuring a global analysis model.

We have also applied the technology to three case studies in cooperation with industry partners in telecommunications, multimedia, and automobile manufacturing. Each case had a very different focus. In the telecommunications project, we resolved a severe transient-fault system integration problem that not even prototyping could solve. In the multimedia case study, we modeled and analyzed a complex two-stage dynamic memory scheduler to derive maximum response times for buffer sizing and priority assignment. In the automotive study, we showed how the technology enables a formal software certification procedure.

The case studies have demonstrated the power and wide applicability of the event flow interfacing approach. It allows designers to apply scheduling analysis techniques to programmable cores and their software as well as to hardware components, as MpSoC verification requires. We consider this approach to be a serious alternative to performance simulation. The new technology allows comprehensive system integration and provides much more reliable performance analysis results at far less computation time.

Global application of the analysis technique still requires some expert knowledge to guide the process, but we are working on an automated system using libraries of analysis techniques. We can profit here from the host of work already completed in real-time systems analysis. ■

References

1. S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A Multiprocessor SoC for Advanced Set-Top Box and Digital TV Systems," *IEEE Design & Test of Computers*, Sept.-Oct. 2001, pp. 21-31.
2. L. Benini and G. DeMicheli, "Networks on Chips: A New SoC Paradigm," *Computer*, Jan. 2002, pp. 70-78.
3. F. Wolf, *Behavioral Intervals in Embedded Software*, Kluwer Academic, 2002.
4. C.L. Liu and J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, 1973, pp. 46-61.
5. E. Jensen, C. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Sys-

tems, *Proc. 6th IEEE Real-Time Systems Symp.* (RTSS1985), IEEE CS Press, 1985, pp. 112-122.
6. E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Computers*, Jan. 1987, pp. 24-35.
7. K. Tindell and J. Clark, "Holistic Schedulability for Distributed Hard Real-Time Systems, *Euromicro J.*, vol. 40, 1994, pp. 117-134.
8. P. Pop, P. Eles, and Z. Peng, "Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems," *Proc. Int'l Symp. Hardware/Software Codesign* (CODES2002), ACM Press, 2002, pp. 187-192.
9. K. Gresser, "An Event Model for Deadline Verification of Hard Real-Time Systems," *Proc. 5th Euromicro Workshop on Real-Time Systems*, IEEE CS Press, 1993, pp. 118-123.
10. L. Thiele, S. Chakraborty, and M. Naedele, "Real-time Calculus for Scheduling Hard Real-Time Systems," *Proc. Int'l Symp. Circuits and Systems* (ISCAS 2000), IEEE CS Press, 2000, pp. 101-104.
11. K. Richter and R. Ernst, "Event Model Interfaces for Heterogeneous System Analysis," *Proc. Design, Automation and Test in Europe* (DATE2002), IEEE CS Press, 2002, pp. 506-513.
12. K. Richter et al., "Model Composition for Scheduling Analysis in Platform Design," *Proc. Design Automation Conf.* (DAC2002), ACM Press, 2002, pp. 287-292.

Kai Richter *is a PhD candidate at the Technical University of Braunschweig, Germany, and a research staff member of its Institute of Computer and Communication Network Engineering. His research interests include real-time systems, performance analysis, and heterogeneous hardware/software platforms. He received a Diploma (Dipl.-Ing.) in electrical engineering from the University of Braunschweig. Contact him at kair@ ida.ing.tu-bs.de.*

Marek Jersak *is a PhD candidate at the Technical University of Braunschweig, Germany, and a research staff member of its Institute of Computer and Communication Network Engineering. His research interests include real-time embedded systems, multilanguage design, and model transformations. He received a Dipl.-Ing. in electrical engineering from the Aachen Institute of Technology, Germany. Contact him at marek@ ida.ing.tu-bs.de.*

Rolf Ernst *is a full professor at the Technical University of Braunschweig, Germany, where he heads the Institute of Computer and Communication Network Engineering. His main research interests are embedded system design and embedded system design automation. He received a PhD in electrical engineering from the University of Erlangen-Nürnberg, Germany. Contact him at ernst@ida.ing.tu-bs.de.*