Exploring Embedded-Systems Architectures with Artemis

The Artemis modeling and simulation environment aims to efficiently explore the design space of heterogeneous embedded-systems architectures at multiple abstraction levels and for a wide range of applications targeting these architectures.

Andy D. Pimentel
Louis O. Hertzberger
University of Amsterdam

Paul Lieverse
Delft University of Technology

Pieter van der Wolf
Philips Research Laboratories, Eindhoven

Ed F. Deprettere
Leiden University

Designers of modern embedded systems face several emerging challenges. Because embedded systems mostly target mass production and often run on batteries, they should be cheap to realize and power efficient. In addition, they must, increasingly, support multiple applications and standards, for which they need to provide real-time performance. For example, digital televisions or mobile devices must support different standards for communication and coding of digital contents. Further, modern embedded systems should also be flexible to enable easily extending them to support future applications and standards. Such flexible support for multiple applications calls for a high degree of programmability.

However, performance requirements as well as cost and power-consumption constraints require implementing substantial parts of these systems in dedicated hardware blocks. As a result, modern embedded systems often have a heterogeneous system architecture—they consist of components that range from fully programmable processor cores to fully dedicated hardware components for time-critical application tasks. Increasingly, such heterogeneous systems reside together on a single chip, yielding heterogeneous multiprocessor systems-on-chip that exploit task-level parallelism in applications.

The heterogeneity of these highly programmable embedded systems and the varying demands of their target applications greatly complicate system design. We must reconsider the suitability of existing design techniques for these systems for two reasons:

- Classical design methods typically start from a single-application specification, making them unsuited to highly programmable embedded systems.
- Common simulation practice for the design-space exploration of architectures is not appropriate for the early stages of heterogeneous embedded-system design.

Classical design methods that depart from a single-application specification that gradually synthesizes into an architecture implementation consisting of hardware and software components are not suitable for programmable architectures. These methods, while perhaps ideal for designing dedicated systems, lack the generalizability required to cope with the programmable system architectures suited to the efficient processing of a range of applications. In addition, classical design methods do not offer the extensibility embedded systems require to efficiently support future applications.

The increasing complexity of embedded-systems architectures makes predicting performance behavior more difficult. Therefore, having the appropriate tools to explore different choices at an early design stage is increasingly important. Currently, one common approach still uses only relatively detailed, often clock-cycle-accurate simulators for the design-space exploration of embedded-systems architectures. Building such detailed simulators requires considerable effort, making it impractical to use them in the early design stages. Moreover, their low simulation speeds significantly hamper architectural exploration.

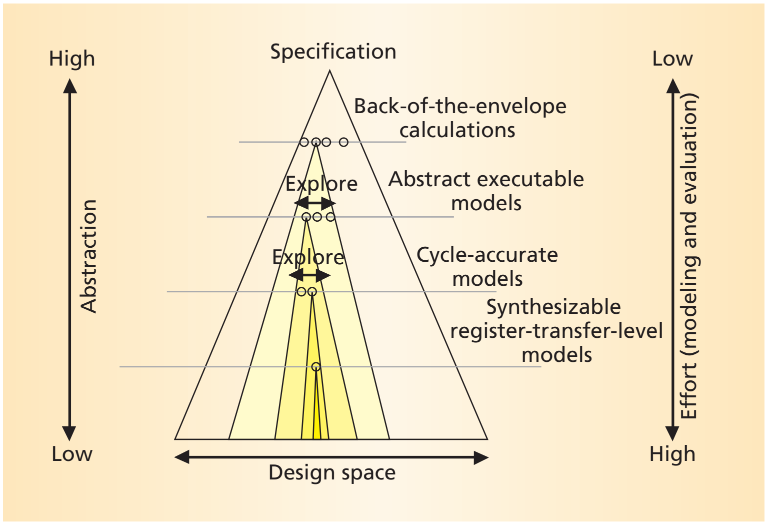
High
Abstraction
Low
Specification
Back-of-the-envelope calculations
Explore
Abstract executable models
Explore
Cycle-accurate models
Synthesizable register-transfer-level models
Design space
Low
Effort (modeling and evaluation)
High

Figure 1. System design abstraction-level pyramid. Activities closer to the top of the pyramid tend to be more abstract and require relatively little effort. Conversely, activities closer to the pyramid's base involve little abstraction and great effort.

RETHINKING DESIGN-SPACE EXPLORATION

To uncover the design space of complex embedded systems, an exploration environment should allow for the performance evaluation of embedded-systems-architecture instantiations at multiple abstraction levels for a broad range of applications. Performing simulation at multiple abstraction levels makes it possible to control the speed, required modeling effort, and attainable accuracy of the simulations. This enables using a stepwise refinement approach in which abstract simulation models efficiently explore the large design space in the early design stages. Applying more detailed models at a later stage allows focused architectural exploration, as the abstraction-level pyramid in Figure 1 shows.

An architecture design-space exploration environment should also be open to reusing intellectual property, thereby helping reduce a product's time to market. For example, reusing simulation models of architecture components, such as microprocessors, buses, and memory, must be relatively easy. This requirement calls for a high degree of modularity when building system architecture models as well as a clear separation of specifying application behavior and architecture performance constraints.

Finally, the design-space exploration of complex heterogeneous embedded systems must efficiently incorporate both the application and architecture aspects of the system under study, and it should be able to do so in the early design stages and thus at a high abstraction level. To illustrate this requirement, consider the following example of a design strategy for heterogeneous embedded-systems architectures.

As a first step in the design process, designers typically construct a functional software model to study a target application and obtain rough estimates of its performance needs—for example, the bandwidth a video application requires to achieve a certain frame rate. Such a model usually neglects most of the underlying architectural issues, such as resource contention or bandwidth constraints, involved in the application's execution.

Next, designers apply cycle-accurate instruction-level simulators and register-transfer-level (RTL) hardware simulators to study a certain architectural implementation of the embedded system. Constructing the models for these simulations requires making several design decisions. For example, designers should decide which parts of an application the dedicated hardware performs and which parts the software running on a programmable processor performs.

Two problems relate to these design decisions. First, making these decisions based on a functional software model that disregards most architectural issues is highly questionable. Second, evaluating different design decisions at the second step of the design process—such as assessing alternative hardware-software partitionings when the initial partitioning was not optimal—can require an enormous amount of remodeling effort. Further, the slow simulation speed of the detailed simulators seriously hampers such work.

ARTEMIS

To reduce the design time for highly programmable embedded multimedia systems, the Artemis (Architectures and Methods for Embedded Media Systems) project[1] focuses on solving two research challenges.

First, we are developing an architecture modeling and simulation environment that provides methods, tools, and libraries for the efficient exploration of heterogeneous embedded-systems architectures. We use the term *efficient* to indicate that the modeling and simulation environment enables rapid evaluation of different architecture designs, application-to-architecture mappings, and hardware-software partitionings at various abstraction levels for a broad range of applications. Artemis focuses on embedded multimedia systems, but developers can apply its techniques and tools to other (nonembedded) application domains that use task-level parallelism.

Second, Artemis investigates the potential of using reconfigurable embedded computer architectures as a new means of enhancing the programmability of embedded systems. Reconfigurable computing refers to an implementation style in which developers can configure a piece of hardware's exact functional behavior to efficiently process a particular task, then reconfigure it later for a different task. Reconfigurable hardware components—implemented using field-programmable gate arrays, for example—retain flexibility and have the potential to deliver high performance for specific applications, while limiting power consumption.

Figure 2 shows how both research activities integrate into the Artemis architecture workbench. Using the architecture modeling and simulation environment, we can explore potential embedded-system architectures for a set of target applications. The architecture design-space exploration performed at this stage results in recommendations for candidate architectures. In addition, analysis of application-architecture mappings, such as which application task maps onto which system architecture component, helps determine a selection of time-critical application tasks that are candidates for execution on a reconfigurable hardware component. Code fragments represent these selected tasks and subsequently act as input to the reconfigurable architecture framework's tools.

The tools from the reconfigurable architecture framework enable thorough study of task mapping onto a reconfigurable hardware component. Such a study produces accurate performance estimates for the reconfigurable hardware component, which developers can use to validate and calibrate the modeling and simulation environment's system-level architecture models. Combining the architecture modeling and simulation environment with the reconfigurable architecture framework should ultimately lead to a system architecture proposal that allows efficient processing of the target applications.

Many other groups are active in the field of modeling and simulating heterogeneous embedded systems, some of which pursue academic efforts[2-4] while others focus on commercial[5] and industrial[6] endeavors. Much work in this field combines in a single simulation the software parts mapped onto a programmable processor, the hardware components, and their interactions. Because this work makes an explicit distinction between software and hardware simulation, developers must know which application components the system will implement in software and which in hardware—before they build a system model. This requirement significantly complicates the performance evaluation of different hardware-software partitioning schemes because the assessment of each partitioning may require a new system model. Artemis uses a substantially different approach to overcome this problem.

MODELING AND SIMULATION METHODOLOGY

Designing programmable embedded systems requires making a clear distinction between applications and architectures, and the design must support

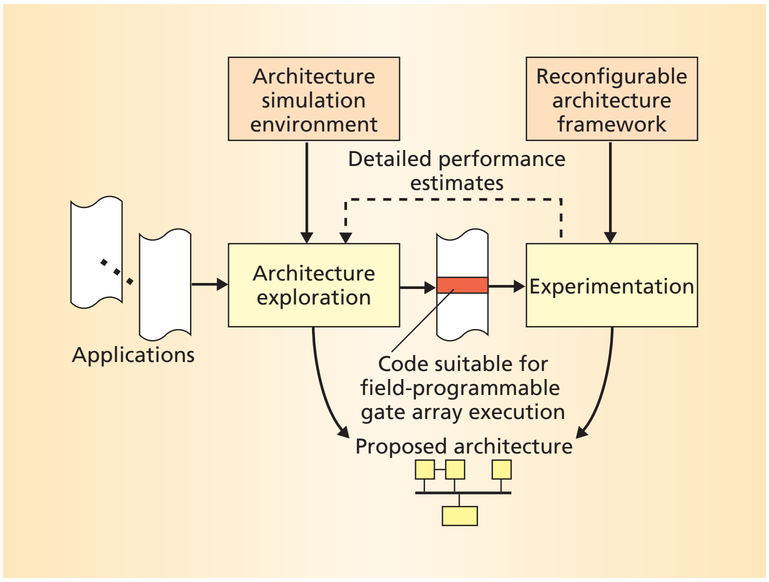
Architecture simulation environment
Reconfigurable architecture framework
Detailed performance estimates
Architecture exploration
Experimentation
Applications
Code suitable for field-programmable gate array execution
Proposed architecture

Figure 2. The Artemis architecture workbench. The workbench integrates the research activities of architecture-design-space exploration and reconfigurable architectures.

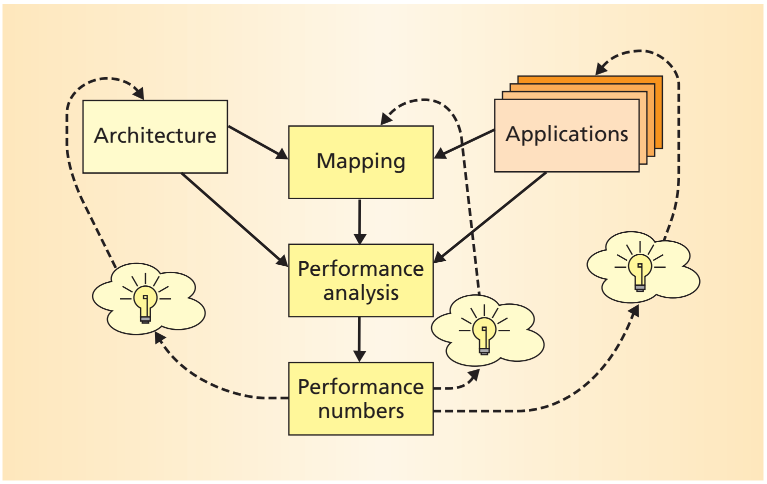

Figure 3. System design Y-chart. The upper right part of the chart consists of the application set that drives the architecture design. The designer studies these applications, proposes an architecture, then compares its effectiveness against alternatives. The light bulbs indicate that the results may inspire the designer to improve the architecture, restructure the application, or modify its mapping.

an explicit mapping step. This approach permits mapping multiple target applications, one after the other, onto candidate architectures for performance evaluation.

Figure 3 visualizes this approach, which we refer to as the Y-chart of system design.[3,7] The application set drives the architecture design. Typically, the designer studies these applications, makes some initial calculations, and proposes an architecture. The designer then evaluates and compares the architecture's effectiveness against alternative architectures. To this end, the designer uses performance analysis to quantitatively evaluate candidate architecture designs, mapping each application onto the architecture under investigation and evaluating the performance of each application-architecture combination. The resulting performance numbers may inspire the designer to improve the architecture, restructure the application, or modify its mapping.

Designers can use a single application model to exercise different hardware-software partitionings and map it onto a range of architecture models.

The Artemis modeling and simulation environment facilitates the performance analysis of embedded-systems architectures in a way that directly reflects the Y-chart design approach, recognizing separate application and architecture models for system simulation. An application model describes the application's functional behavior, including both its computational and communication behavior. The architecture model defines architecture resources and captures their performance constraints. This modeling methodology requires an application model to remain independent from architectural specifics, assumptions on hardware-software partitioning, and timing characteristics.

As a result, designers can use a single application model to exercise different hardware-software partitionings and map it onto a range of architecture models, possibly representing different system architectures or simply modeling the same system architecture at various abstraction levels. This capability clearly demonstrates the strength of decoupling application and architecture models, enabling the reuse of both model types. After mapping, the designer cosimulates an application model with an architecture model, allowing for the system-performance evaluation of a particular application, mapping, and underlying architecture.

Trace-driven cosimulation

Cosimulating application models and architecture models requires an interface that includes a mapping specification between them. For this purpose, we apply trace-driven simulation, a technique used extensively in the field of general-purpose processor design to analyze the performance of memory hierarchies. In our approach, we structure the application model as a network of concurrent communicating processes, thereby expressing the inherent task-level parallelism available in the application and making communication explicit.

Each process, when executed, produces a trace of events that represents the application workload imposed on the architecture by that particular process. Thus, the trace events refer to the computation and communication operations an application process performs. These operations can be coarse grained, such as "Compute a discrete cosine transform." Therefore, our approach differs from classical trace-driven simulation in which the events typically refer to fine-grained, instruction-level operations.

Because application models represent functional behavior, the traces correctly reflect data-dependent behavior. Consequently, the architecture models, driven by the application traces, only need to account for the application events' performance consequences, not their functional behavior.

Application modeling

For application modeling, we use the Kahn Process Network (KPN) computational model.[8] To obtain a Kahn application model, we restructure a sequential application written in C/C++ into a program that consists of parallel processes communicating with each other via unbounded FIFO channels. In the Kahn paradigm, channel reads are blocking, while writing is nonblocking.

To capture an application's computational behavior, we instrument the code of each Kahn process with annotations that describe the application's computational actions. Reading from or writing to Kahn channels represents a process's communication behavior within the application model. By executing the Kahn model, each process records its actions to generate a trace of application events, which is necessary for driving an architecture model.

Much research has been done in the field of application modeling on computational models.[9] We use KPNs because they fit nicely with our media-processing application domain and they are deterministic. This latter attribute means that the same application input always results in the same application output, making the application behavior architecture-independent and automatically guaranteeing the validity of event traces when the application and architecture simulators execute independently of each other. However, because KPN semantics disallow, for example, the modeling of interrupts, we currently cannot model applications with time-dependent behavior.

Using a separate application model also makes it possible to analyze an application's performance requirements and potential performance constraints in isolation from any architecture. This lets us investigate the application's upper performance bounds and may lead to early recognition of bottlenecks within the application itself.

Architecture modeling

An architecture model is based on components that represent processors or coprocessors, memories, buffers, buses, and so on. To evaluate an architecture's performance, we can simulate the performance consequences of an application model's generated computation and communication events. Such simulation requires an explicit mapping of a Kahn application model's processes and channels onto the architecture model's components.

A trace-event queue routes the generated trace of application events from a specific Kahn process toward a specific component inside the architecture model. The Kahn process dispatches its application events to this queue, while the designated component in the architecture model consumes them, as Figure 4 shows. Two or more Kahn process trace-event queues can be mapped onto a single architecture component—for example, several application tasks can be mapped onto a microprocessor. In this case, the architecture simulator must schedule the events from the different queues.

The underlying architecture model need not model functional behavior, because the application model already captures that behavior and subsequently drives the architecture simulation. Designers construct an architecture model from generic building blocks provided by a library. This library contains performance models for processing cores, communication media such as buses, and different memory types. Such a library-based modeling approach can greatly simplify the reuse of architecture model components.

At a high abstraction level, a processing-core model functions as a *black box* that can simulate the timing behavior of a programmable processor, reconfigurable component, or dedicated hardware unit. The architecture simulator can model such a variety of architectural implementations because it assigns parameterizable latencies to the incoming application events. For example, to model software execution of an application event, the simulator can assign a relatively high latency to it. Likewise, to model an application event that dedicated or reconfigurable hardware is executing, the simulator tags the event with a lower latency.

By simply varying the latencies for computational application events, the simulator can evaluate different hardware-software partitionings at a high abstraction level. The simulator can obtain these latencies from a lower-level architecture component model, performance-estimation tools, or the estimates of an experienced designer.

This approach uses the communication events from the application model to model the performance consequences of data transfers and synchronizations at the architecture level. These events cause the appropriate communication component within the architecture model—onto which the simulator maps the communicating Kahn channel—to account for the latencies associated with the data transfers.

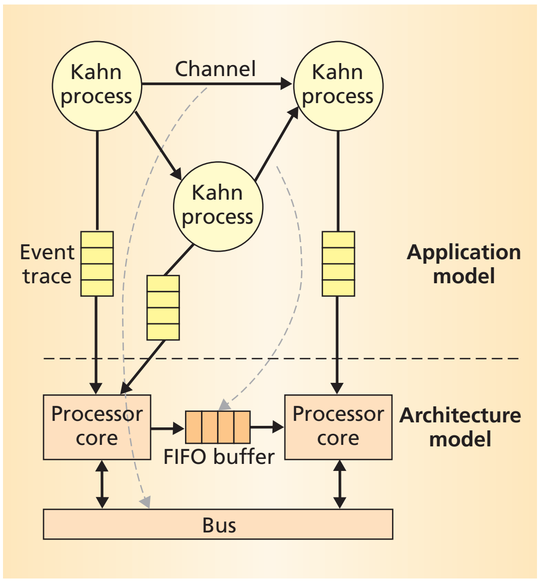

Figure 4. Mapping a Kahn application model onto an architecture model. A trace-event queue routes the generated trace of application events from a specific Kahn process toward a specific component inside the architecture model.

Unlike the application model, in which all first-in, first-out channels are unbounded, the writes at the architecture level can also be blocking, depending on the availability of resources such as buffer space.

Model refinement

The designer makes design decisions, such as hardware-software partitioning, and refines the architecture model components accordingly. This gradual adaptation implies that the architecture model begins to reflect the characteristics of a particular implementation—dedicated versus programmable hardware, for example.

To facilitate the process of model refinement, the architecture model library should include models of common architecture components at several abstraction levels. For example, the microprocessor model can have multiple instances, such as a black-box model, a model that accounts for the performance consequences of the processor's memory hierarchy, and one that accounts for the performance impact of both its memory hierarchy and data path. Moreover, the simulation should refine application model events to match the detail level present in the architecture model. Providing flexible support for such event refinement remains a largely open problem.[6]

The model refinement process can continue to the level at which it embeds detailed simulators for certain architecture components—such as instruction-level simulators or RTL simulators—into the overall system architecture simulation. Consider the example in which the designer decides that software will implement a certain application task. Instead of mapping the task's Kahn process onto a processor core's abstract architecture model, a detailed instruction-

The modeling concepts should include support for refining only parts of an architecture model, creating a mixed-level simulation model.

level simulator can emulate the application task's actual code. The process of embedding more detailed simulators can continue, gradually incorporating greater functionality into an architecture model. Ultimately, the architecture model can then provide a starting point for more traditional hardware-software cosimulation, which is composed of instruction-level simulators and RTL simulators.

ARTEMIS DRIVERS

For the Artemis architecture modeling and simulation environment's development, we are currently experimenting with two prototype simulation frameworks: Spade (system-level performance analysis and design-space exploration)[10] and Sesame (simulation of embedded-system architectures for multilevel exploration).[11] Both frameworks act as technology drivers for testing and evaluating new simulation models and methods, giving us insight into their suitability for the Artemis environment. We only incorporate those simulation models and methods that prove valid and effective into Artemis.

The Spade framework emphasizes simplicity, flexibility, and easy interfacing to more detailed simulators. It provides a small library of architecture model components consisting of a black-box model of a processing core, a generic bus model, a generic memory model, and several interfaces for connecting these model building blocks. We can use this limited library of model building blocks and interfaces to rapidly conduct a large variety of system-level performance studies.

We implemented Spade's architecture model components using the cycle-based Tool for System Simulation (TSS), a Philips in-house simulation environment. Philips has a large user community that is applying TSS to implement cycle-accurate architecture simulators. Sharing a common simulation backbone significantly simplifies the transition from high-level Spade models to detailed TSS architecture models. Currently, we are also considering alternatives for TSS, such as SystemC.

The Sesame framework studies the potential of conducting simulations at multiple abstraction levels and explores concepts for refining simulation models smoothly across different abstraction levels. For example, refinement of one component in an architecture model should not lead to a completely new implementation of the entire model. Thus, the modeling concepts should also include support for refining only parts of an architecture model, creating a mixed-level simulation model.

Mixed-level simulations enable more detailed evaluation of a specific architecture component within the context of the entire system's behavior. Therefore, such simulations avoid building a complete, detailed architecture model during the early design stages. Moreover, mixed-level simulations do not suffer from the deterioration in system-evaluation efficiency that unnecessarily refined parts of the architecture model causes.

Sesame currently provides only a library of black-box architecture models. In the near future, we will extend the library with models for architecture components, at several abstraction levels, to facilitate the performance evaluation of architectures from the black-box level to cycle-accurate models. This library will eventually be supplemented with techniques and tools to assist developers in gradually refining the models and performing mixed-level simulations. Currently, these issues largely remain open research problems.

To implement Sesame's architecture models, we use a small but powerful discrete-event simulation language that enables easy model construction and fast simulation. These characteristics greatly improve the scope of the design space the designer can explore in a reasonable time. Sesame's architecture library components are not meant to be fixed building blocks with predefined interfaces, but merely freely extendable and adaptable template models. This approach slightly increases the effort required to build architecture models, but it also achieves a high degree of flexibility, which can be helpful when refining models.

We have applied our modeling and simulation methodology to two media applications: an MPEG-2 decoder[10] and a variant of M-JPEG encoding.[11,12] Both studies, performed at the black-box architecture model level, showed promising results. In a short period, we obtained useful feedback on a wide range of design decisions involving the candidate architectures for the two applications we studied. For example, for the M-JPEG application, we experimented with a shared-memory multiprocessor architecture model. For this architecture, we evaluated different hardware-software partitionings, application to architecture mappings, processor speeds, and interconnect structures: bus, crossbar, and omega networks. All of this work, including the application and architecture modeling, took less than one person-month.

We intend to perform more case studies with industrially relevant applications to further demonstrate the power and effectiveness of our methods and tools. The validation of our simulation models also requires attention. Future research will emphasize techniques for model refinement. In particular, support for mixed-level simulation introduces many new research problems that developers must address. ✷

Acknowledgments

This research is supported by PROGRESS, the embedded-systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs, and the Technology Foundation STW. We thank all members of the Artemis team and acknowledge Bart Kienhuis and Kees Vissers for their contribution to the modeling methodology.

References

1. A.D. Pimentel et al., "The Artemis Architecture Workbench," *Proc. Progress Workshop Embedded Systems,* STW Technology Foundation, Utrecht, the Netherlands, 2000, pp. 53-62.
2. K. Hines and G. Borriello, "Dynamic Communication Models in Embedded System Cosimulation," *Proc. Design Automation Conf.*, ACM Press, New York, 1997, pp. 395-400.
3. F. Balarin et al., *Hardware-Software Codesign of Embedded Systems: The POLIS Approach*, Kluwer Academic, Dordrecht, the Netherlands, 1997.
4. S.L. Coumeri and D.E. Thomas, "A Simulation Environment for Hardware-Software Codesign," *Proc. Int'l Conf. Computer Design,* IEEE CS Press, Los Alamitos, Calif., 1995, pp. 58-63.
5. P. Dreike and J. McCoy, "Cosimulating Software and Hardware in Embedded Systems," *Embedded Systems Programming,* June 1997, pp. 12-27.
6. J-Y. Brunel et al. "Communication Refinement in Video Systems on Chip," *Proc. 7th Int'l Workshop Hardware-Software Codesign,* ACM Press, New York, 1999, pp. 142-146.
7. B. Kienhuis et al., "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures," *Proc. Int'l Conf. Application-Specific Systems, Architectures, and Processors,* IEEE CS Press, Los Alamitos, Calif., 1997, pp. 338-349.
8. G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. IFIP Congress 74,* North-Holland, Amsterdam, 1974, pp. 471-475.
9. J. Buck et al., "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int'l J. Computer Simulation,* Apr. 1994, pp. 155-182.
10. P. Lieverse et al., "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems," *J. VLSI Signal Processing for Signal, Image and Video Technology,* special issue on SiPS 99, vol. 29, no. 3, 2001, pp. 197-207.
11. F. Terpstra et al., "Rapid Evaluation of Instantiations of Embedded Systems Architectures: A Case Study," *Proc. PROGRESS Workshop Embedded Systems,* STW Technology Foundation, Utrecht, the Netherlands, 2001, pp. 251-260.
12. P. Lieverse et al., "System Level Design with Spade: An M-JPEG Case Study," to be published in *Proc. Int'l Conf. Computer-Aided Design,* IEEE CS Press, Los Alamitos, Calif., 2001.

Andy D. Pimentel *is an assistant professor in the Department of Computer Science at the University of Amsterdam. His research interests include computer architecture, embedded systems, computer architecture modeling and simulation, performance analysis, and parallel computing. Pimentel received a PhD in computer science from the University of Amsterdam. He is a member of the IEEE Computer Society. Contact him at andy@science.uva.nl.*

Louis O. Hertzberger *is a professor in the Department of Computer Science at the University of Amsterdam. His research interests are in the field of complex computer systems design, with an emphasis on parallel/distributed systems and embedded industrial systems. He received a PhD in experimental physics from the University of Amsterdam. Contact him at bob@science.uva.nl.*

Paul Lieverse *is currently completing a PhD in the Department of Information Technology and Systems/Electrical Engineering at Delft University of Technology, in close cooperation with Philips Research Laboratories, Eindhoven, the Netherlands. His research interests include system-level design and performance analysis of heterogeneous multiprocessor systems. Lieverse received an MSc in electrical engineering from Delft University of Technology. He is a student member of the IEEE and the ACM. Contact him at lieverse@ieee.org.*

Pieter van der Wolf *is a senior research scientist at Philips Research Laboratories, Eindhoven, the Netherlands. His research interests include system-level design methodologies, embedded-systems architectures, and embedded processors. He received a PhD in electrical engineering from Delft University of Technology. Van der Wolf is a member of the IEEE. Contact him at Pieter.van.der.Wolf@philips.com.*

Ed F. Deprettere *is a professor of embedded real-time computing in the Computer Science Department at Leiden University. His research interests include models, methods, and tools for the exploration of flexible, yet high-performance combinations of applications and architectures in the domains of multimedia, signal processing, and wireless communications. He is a member of the ACM and a fellow of the IEEE. Contact him at edd@liacs.nl.*