A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels

Andy D. Pimentel, *Member*, *IEEE Computer Society*,
Cagkan Erbas, *Student Member*, *IEEE*, and Simon Polstra

Abstract—The sheer complexity of today's embedded systems forces designers to start with modeling and simulating system components and their interactions in the very early design stages. It is therefore imperative to have good tools for exploring a wide range of design choices, especially during the early design stages, where the design space is at its largest. This paper presents an overview of the Sesame framework, which provides high-level modeling and simulation methods and tools for system-level performance evaluation and exploration of heterogeneous embedded systems. More specifically, we describe Sesame's modeling methodology and trajectory. It takes a designer systematically along the path from selecting candidate architectures, using analytical modeling and multiobjective optimization, to simulating these candidate architectures with our system-level simulation environment. This simulation environment subsequently allows for architectural exploration at different levels of abstraction while maintaining high-level and architecture-independent application specifications. We illustrate all these aspects using a case study in which we traverse Sesame's exploration trajectory for a Motion-JPEG encoder application.

Index Terms—Modeling of computer architecture, real-time and embedded systems, simulation, modeling techniques, performance analysis and design aids.

1 Introduction

Advances in chip technology according to Moore's Law, allowing more and more functionality to be integrated on a single chip, have led to the emergence of *Systems on Chip* (SoCs). These SoCs are nowadays key to the development of advanced embedded computing systems, such as set-top boxes, digital televisions, and 3G cell phones. Designers of these SoC-based embedded systems are typically faced with conflicting design requirements regarding performance, flexibility, power consumption, and cost. As a result, SoC-based embedded systems often have a *heterogeneous system architecture*, consisting of components that range from fully programmable processor cores to fully dedicated hardware blocks. Programmable processor technology is used for realizing flexibility, for example, to support multiple applications and future extensions, while dedicated hardware is used to optimize designs in time-critical areas and for power and cost minimization.

The heterogeneity of modern embedded systems and the varying demands of their target applications greatly complicate the system design. It is widely agreed upon that traditional design methods fall short for the design of these systems as such methods cannot deal with the systems' complexity and flexibility. This has led to the notion of *system-level design*, in which aspects such as *platform architectures*, *separation of concerns*, and *high-level modeling and simulation* play an important role. Platform-based design [1], [2] stresses the reuse of IP (Intellectual Property) blocks. In this design approach, a single hardware platform is used as a "hardware denominator" that is shared across multiple applications in a given domain and is accompanied by a range of methods and tools for design and development. This increases production volume and reduces cost compared to customizing a chip for every application.

To even further improve the potentials for reuse of IP and to allow for effective exploration of alternative design solutions, it is also widely recognized that "separation of concerns" [3] is a crucial component in system-level design. Two common types of separation in the design process are: 1) separating computation from communication by connecting IP processing cores via a standard network interface and 2) separating application (what is the system supposed to do) from architecture (how it does it).

Moreover, system-level design methodologies typically urge designers to start with modeling and simulating (possible) system components and their interactions in the early design stages [4]. Such system-level models usually represent application behavior, architecture characteristics, and the relation (e.g., mapping, hardware-software partitioning) between application(s) and architecture. These models do so at a high level of abstraction, thereby minimizing the modeling effort and optimizing simulation speed that is needed for targeting the early design stages. This high-level modeling allows for early verification of a design and can provide estimations on the performance, power consumption, or cost of the design.

• *The authors are with the Informatics Institute, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands.*
E-mail: {andy, cagkan, spolstra}@science.uva.nl.

Manuscript received 30 Nov. 2004; revised 18 Feb. 2005; accepted 9 May 2005; published online 21 Dec. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0385-1104.

The above ingredients for system-level design should be accompanied by a proper methodology for effective and efficient *design space exploration*. Due to the systems' complexity, it is imperative to have good (performance evaluation) tools for exploring a wide range of design choices, especially during the early design stages, where the design space is at its largest. In the context of the Artemis project [4], [5], we are developing the Sesame[1] framework, which provides high-level modeling and simulation methods and tools for efficient system-level performance evaluation and exploration of heterogeneous SoC-based embedded systems targeting the multimedia application domain. This paper presents an overview of Sesame. More specifically, we describe Sesame's modeling methodology and trajectory, in which the aforementioned principle of separation of concerns [3] is deployed. Sesame takes a designer systematically along the path from selecting candidate architectures, using analytical modeling and multiobjective optimization, to simulating these candidate architectures with a system-level simulation environment. This simulation environment subsequently allows for architectural exploration at different levels of abstraction (i.e., permits gradual refinement of the architecture performance models) while maintaining high-level and architecture-independent application specifications. We illustrate all these aspects using a case study in which we traverse Sesame's exploration trajectory for a Motion-JPEG encoder application.

The remainder of the paper is organized as follows: The next section discusses related work, after which Section 3 describes the ingredients that we believe are important to efficient system-level design space exploration. Section 4 describes the analytical methods and multiobjective optimization techniques we apply to efficiently select promising target architectures that can be further studied using simulation. In Section 5, we subsequently explain how selected architectures can be modeled and simulated for performance evaluation using our system-level simulation framework. Section 6 describes how Sesame facilitates gradual refinement of system-level architecture models by applying dataflow graphs. Section 7 illustrates the discussed aspects of Sesame's modeling methodology and trajectory using a case study with a Motion-JPEG encoder application. Finally, Section 8 concludes the paper.

2 Related Work

There are a number of architectural exploration environments, such as (Metro)Polis [6], [7], Mescal [8], MESH [9], Milan [10], and various SystemC-based environments, like the work of [11], that facilitate flexible system-level performance evaluation by providing support for mapping a behavioral application specification to an architecture specification. For example, in MESH [9], a high-level simulation technique based on frequency interleaving is used to map logical events (referring to application functionality) to physical events (referring to hardware resources). In [12], an excellent survey is presented of various methods, tools, and environments for early design space exploration. In comparison to most related efforts, Sesame tries to push the separation of modeling application behavior and modeling architectural constraints at the system level to even greater extents. As will be explained, this is achieved by architecture-independent application models, application-independent architecture models, and a mapping step that relates these models for trace-driven cosimulation. Moreover, within Sesame, we use multiple models of computation, specifically chosen in accordance with the task to be achieved. For example, we use process networks for application modeling, dataflow graphs to facilitate model refinement, and a discrete-event simulator for fast simulation of our architecture models.

The work of [13] also uses a trace-driven approach, but this is done to extract communication behavior for studying on-chip communication architectures. Rather than using the traces as input to an architecture simulator, their traces are analyzed statically. In addition, a traditional hardware/software cosimulation stage is required in order to generate the traces. Archer [14] shows similarities with the Sesame framework due to the fact that both Sesame and Archer stem from the earlier Spade project [15]. A major difference is, however, that Archer follows a different application-to-architecture mapping approach. Instead of using event traces, it maps so-called Symbolic Programs, which are derived from the application model, onto architecture model resources. Moreover, unlike Sesame, Archer does not include support for rapidly pruning the design space.

Ptolemy [16] is an environment for the simulation and prototyping of heterogeneous systems. It allows for using multiple models of computation (e.g., discrete event, finite state machines, CSP [17], dataflow [18], Kahn Process Networks [19], etc.) within a single system simulation. It does so by supporting domains to build subsystems, each conforming to a different model of computation.

In the domain of hardware/software codesign of embedded systems, multiobjective optimization studies have been performed extensively for system-level synthesis (e.g., [20], [21], [22]) and platform configuration (e.g., [23], [24], [25]). The former refers to the problem of optimally mapping a task-level specification onto a heterogeneous hardware/software architecture, while the latter includes tuning the platform architecture parameters and exploring its configuration space. In the Sesame framework, we also apply multiobjective optimization methods, although we do not primarily target the problem of system synthesis. Rather, our primary objective is to develop a methodology that allows for evaluating a large architectural design space and to steer the designer in the exploration process by providing a number of approximated Pareto-optimal solutions. These solutions are then fed to our simulation framework for further evaluation. After simulation, system-level performance numbers (e.g., utilization of components, data throughput, communication contention, etc.) are provided to the designer which may subsequently be used for a new exploration iteration. The Milan framework [10] follows a similar steering approach, but uses symbolic analysis methods to reduce the design space that needs to be explored using simulation.

Research on the gradual refinement of (abstract) system-level architecture performance models is still in its infancy. There are several attempts being made to address this issue, such as in the Metropolis [7] and Milan frameworks [10], the work of [26], and in the context of SystemC (e.g., [11]). In [26], for example, a methodology is proposed in which architecture-independent specification models are transformed (i.e., refined) into architecture models to facilitate

1. Sesame stands for Simulation of Embedded System Architectures for Multilevel Exploration.

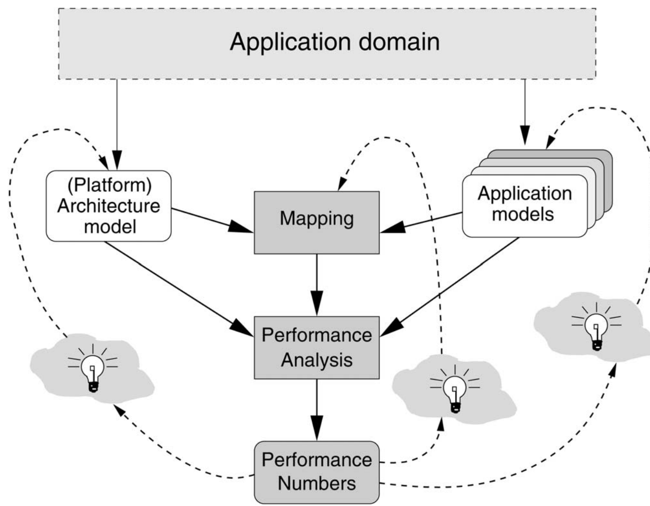

Fig. 1. Y-chart-based design space exploration [6], [32], [33].

architectural exploration. Although promising, these efforts generally do not offer a clear methodology accompanied with tool-support that allows a designer to gradually refine high-level architecture performance models, while retaining the separation between application and architecture as much as possible to allow effective exploration. In addition to this, the majority of the work in this field has focused on communication refinement only. For example, in [27], [28], [29], [30], [31], various mechanisms are proposed for the refinement of application-level communication primitives into more detailed implementation (architecture) primitives.

3 Toward Efficient System-Level Design Space Exploration

Nowadays, it is widely recognized that the separation-of-concerns concept [3] is key to achieving efficient system-level design space exploration of complex embedded systems. In this respect, we advocate the use of the increasingly popular Y-chart design methodology [6], [32], [33] as a basis for (early) design space exploration. This implies that, in Sesame, we separate *application models* and *architecture (performance) models* while also recognizing an explicit *mapping step* to map application tasks onto architecture resources. This is illustrated in Fig. 1. In this approach, an application model—derived from a specific application domain—describes the functional behavior of an application in a timing and architecture independent manner. A (platform) architecture model—which has been defined with the application domain in mind—defines architecture resources and captures their performance constraints. To perform quantitative performance analysis, application models are first mapped onto and then cosimulated with the architecture model under investigation, after which the performance of each application-architecture combination can be evaluated. Subsequently, the resulting performance numbers may inspire the designer to improve the architecture, restructure/adapt the application(s), or modify the mapping of the application(s). These designer actions are illustrated by the light bulbs in Fig. 1.

Essential in this methodology is that an application model is independent of architectural specifics, assumptions on hardware/software partitioning, and timing characteristics. As a result, application models can be reused in the exploration cycle. For example, and as will be demonstrated in this paper, a single application model can be used to exercise different hardware/software partitionings and can be mapped onto a range of architecture models, possibly representing different architecture designs or modeling the same architecture design at various levels of abstraction. With the latter, we refer to the *gradual refinement* of architecture performance models. As design decisions are made, a designer typically wants to descend in abstraction level by disclosing more and more implementation details in an architecture performance model. Eventually, such refinements should bring an initially abstract architecture model closer to the level of detail where it is possible to synthesize an implementation.

Although such system-level modeling and simulation allows for efficiently evaluating different application/architecture combinations, it would fail to explore large parts—let alone the entire span—of the design space. This is because system-level simulation is simply too slow for comprehensively exploring the design space, which is at its largest during the early stages of design. For this reason, it is of the utmost importance that effective steering be provided to the system-level simulation which is capable of guiding the designer toward promising system architectures and which therefore allows for *pruning* the design space. To quickly find promising candidate application-to-architecture mappings in Sesame and thereby reducing the points in the design space that need to be explored with system-level simulation, we have developed an analytical model that captures several trade-offs faced during the process of mapping.

4 Analytical Formulation of the Mapping Problem

As already mentioned in Section 3, Sesame supports separate application and architecture models within its exploration framework. This separation implies an explicit mapping step for cosimulation of the two models. Since the enumeration of all possible mappings grows exponentially, a designer usually needs a subset of best candidate mappings for further evaluation in terms of cosimulation. Therefore, in summary, the mapping problem in Sesame is the optimal mapping of an application model onto a (platform) architecture model. This problem formulation could, for example, take three objectives into account [34]: maximum processing time in the system, total power consumption of the system, and the cost of the architecture. This section describes how we formulate the mapping problem as a multiobjective optimization problem in such a way as to quickly search for promising candidate system architectures with respect to the above three objectives.

Application Modeling. The application models in Sesame are process networks which can be represented by a graph $AP = (V_K, E_K)$, where the sets V_K and E_K refer to the nodes (i.e., processes) and the directed channels between these nodes, respectively. For each node in the application model, a computation requirement (workload imposed by the node onto a particular component in the architecture model) and an allele set (the processors that it can be mapped onto) are defined. For each channel in the application model, a

communication requirement is defined only if that channel is mapped onto an external memory element. Hence, we neglect internal communications (within the same processor) and only consider external (interprocessor) communications.

Architecture Modeling. The architecture models in Sesame can also be represented by a graph $AR = (V_A, E_A)$, where the sets V_A and E_A denote the architecture components and the connections between them, respectively. For each processor in an architecture model, we define the parameters processing capacity, power consumption during execution, and a fixed cost.

Having defined more abstract mathematical models for Sesame's application and architecture model components, we have the following optimization problem:

Definition 1 (MMPN problem [34]). *The Multiprocessor Mappings of Process Networks (MMPN) problem is:*

$$\begin{array}{rl} \min & \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x})) \\ \text{subject to} & g_i(\mathbf{x}), i \in \{1, \cdots, n\}, \mathbf{x} \in X_f, \end{array}$$

where

- f_1 *is the maximum processing time,*
- f_2 *is the total power consumption, and*
- f_3 *is the total cost of the system.*

The functions g_i are the constraints and $\mathbf{x} \in X_f$ are the decision variables. These variables represent decisions like which processes are mapped onto which processors or which processors are used in a particular architecture instance. The constraints of the problem make sure that the decision variables are valid, i.e., X_f is the feasible set. For example, all processes need to be mapped onto a processor from their allele sets or, if two communicating processes are mapped onto the same processor, the channel(s) between them must also be mapped onto the same processor and so on. The optimization goal is to identify a set of solutions which are superior to all other solutions when all three objective functions are minimized.

Here, we have provided an overview of the MMPN problem. The exact mathematical modeling and formulation can be found in [34].

4.1 Multiobjective Optimization

To solve the above multiobjective optimization problem, we use the (improved) Strength Pareto Evolutionary Algorithm (SPEA2) [35] that finds a set of approximated Pareto-optimal mapping solutions, i.e., solutions that are *not dominated* in terms of quality (performance, power, and cost) by any other solution in the feasible set. To this end, SPEA2 maintains an external set to preserve the nondominated solutions encountered so far besides the original population. Each mapping solution is represented by an individual encoding, i.e., a chromosome in which the genes encode the values of parameters. SPEA2 uses the concept of dominance to assign fitness values to individuals. It does so by taking into account how many individuals a solution dominates and is dominated by. Distinct fitness assignment schemes are defined for the population and the external set to always ensure that better fitness values are assigned to individuals in the external set. Additionally, SPEA2 performs *clustering* to limit the number of individuals in the external set (without losing the boundary solutions) while also maintaining diversity among them. For selection, it uses binary tournament with replacement. Finally, only the external nondominated set takes part in selection. In our SPEA2 implementation, we have also introduced a repair mechanism [34] to handle infeasible solutions. The repair takes place before the individuals enter evaluation to make sure that only valid individuals are evaluated.

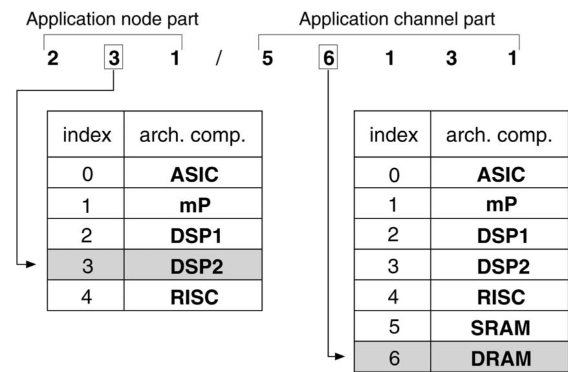

Fig. 2. An example individual encoding.

For our mapping problem, an individual encoding consists of two parts: a part for application nodes and a part for application channels. Each value (i.e., gene) in the individual encoding has its own allele set which is determined by the type of the gene and the constraints of the problem. For genes representing application nodes, only the set of processors in the architecture model form the allele set, while, for genes representing the application channels, both the sets of processors and memories constitute the allele set. The constraints of the problem may include some limitations which should be considered in the individual encoding. For example, if there exists a dedicated architecture component for a specific application process, then only this architecture component has to be included in the allele set of that application process. In Fig. 2, an example of an individual encoding is given. The first three genes are those for application nodes, while the remaining genes are for application channels. For this encoding, the second application process is mapped onto DSP2 and the second application channel is mapped onto DRAM. We also see that the allele sets for these two genes are different.

In [34], we have shown that an SPEA implementation to heuristically solve the multiobjective optimization problem can provide the designer with good insight on the quality of candidate system architectures. This knowledge can subsequently be used to select an initial (platform) architecture to start the system-level simulation phase or to guide a designer in finding alternative architectures when system-level simulation indicates that the architecture under investigation does not fulfill the requirements. The latter is an example of design feedback, as represented by the light bulbs in Fig. 1.

5 System-Level Performance Modeling and Simulation

Sesame's system-level modeling and simulation environment [36], [37] builds upon the ground-laying work of the Spade framework [15]. This means that Sesame facilitates performance analysis according to the Y-chart design

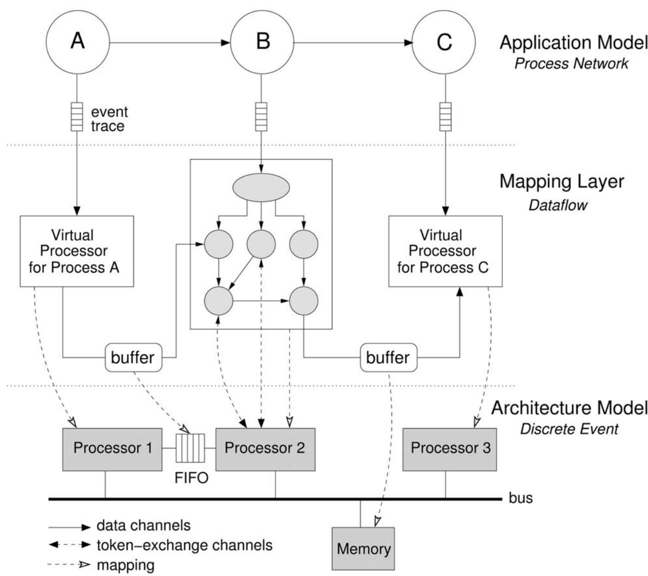

Fig. 3. Sesame's application model layer, architecture model layer, and mapping layer which interfaces between application and architecture models.

approach [6], [32], [33] as discussed in Section 3, recognizing separate application and architecture models. The layered infrastructure of Sesame's modeling and simulation framework is shown in Fig. 3. Sesame maps application models onto architecture models for cosimulation by means of *trace-driven simulation*, while using an intermediate mapping layer for scheduling and event-refinement purposes. The remainder of this section provides an overview of each of the layers as shown in Fig. 3.

5.1 Application Modeling

For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [19] in which parallel processes communicate with each other via unbounded FIFO channels. In the Kahn paradigm, reading from channels is done in a blocking manner, while writing is nonblocking. We use KPNs for application modeling because they nicely fit with the targeted media-processing application domain and they are deterministic. The latter implies that the same application input always results in the same application output, irrespective of the scheduling of the KPN processes. This provides us with a lot of scheduling freedom when mapping KPN processes onto architecture models for quantitative performance analysis. However, because KPN semantics disallow, for example, the modeling of interrupts, our ability to model applications with time-dependent behavior is currently limited.

The Kahn application models used in Sesame are either generated by a framework called Compaan [38], [39] or are derived by hand from sequential C/C++ code. The Compaan framework allows for automatically converting a sequential imperative application specification—more specifically, an application written in a subset of Matlab—into a KPN. The workload of an application is captured by (partly manually) instrumenting the code of each Kahn process with annotations that describe the application's computational and communication actions. By executing the Kahn model, these annotations cause the Kahn processes to generate traces of *application events* which subsequently drive the underlying architecture model. There are three types of application events: the communication events *read* and *write* and the computational event *execute*. These application events typically are coarse grained, such as *execute(DCT)* or *read(channel_id,pixel-block)*.

To execute Kahn application models and thereby generate the application events that represent the workload imposed on the architecture, Sesame features a process network execution engine supporting Kahn semantics. This execution engine runs the Kahn processes, which are written in C++, as separate threads using the Pthreads package. To allow for rapid creation and modification of models, the structure of the application models (i.e., which processes are used in the model and how they are connected to each other) is not hard-coded in the C++ implementation of the processes. Instead, it is described in a language called YML (Y-chart Modeling Language) [37]. This is an XML-based language which is similar to Ptolemy's MoML [40], but is slightly less generic in the sense that YML only needs to support a few simulation domains. As a consequence, YML supports a subset of MoML's features. However, YML provides one additional feature in comparison to MoML as it contains built-in scripting support. This allows for loop-like constructs, mapping, and connectivity functions, and so on, which facilitate the description of large and complex models. In addition, it enables the creation of libraries of parameterized YML component descriptions that can be instantiated with the appropriate parameters, thereby fostering reuse of component descriptions. To simplify the use of YML even further, a YML editor has also been developed to compose model descriptions using a GUI.

5.2 Architecture Modeling

Architecture models in Sesame, which typically operate at the so-called transaction level [41], [42], simulate the performance consequences of the computation and communication events generated by an application model. These architecture models solely account for architectural performance constraints and do not need to model functional behavior. This is possible because the functional behavior is already captured in the application models, which subsequently drive the architecture simulation. An architecture model is constructed from generic building blocks provided by a library, which contains template performance models for processing cores, communication media (like buses), and various types of memory. The structure of architecture models—specifying which building blocks are used from the library and the way they are connected—is also described in YML.

Sesame's architecture models are implemented using either Pearl [43] or SystemC [42]. Pearl is a small but powerful discrete-event simulation language which provides easy construction of the models and fast simulation [36]. For our SystemC architecture models, we provide an add-on library to SystemC, called SCPEx (SystemC Pearl Extension) [44], which extends SystemC's programming model with Pearl's message-passing paradigm and which provides SystemC with YML support. SCPEx raises the abstraction level of SystemC models, thereby reducing the modeling effort required for developing transaction-level architecture models and making the modeling process less prone to programming errors.

5.3 Mapping

To map Kahn processes (i.e., their event traces) from an application model onto architecture model components and to support the scheduling of application events when multiple Kahn processes are mapped onto a single architecture component (e.g., a programmable processor), Sesame provides an intermediate *mapping layer*. This layer consists of virtual processor components and FIFO buffers for communication between the virtual processors. There is a one-to-one relationship between the Kahn processes in the application model and the virtual processors in the mapping layer. This is also true for the Kahn channels and the FIFO buffers in the mapping layer, except for the fact that the latter are limited in size. Their size is parameterized and dependent on the modeled architecture. As the structure of the mapping layer is equivalent to the structure of the application model under investigation, Sesame provides a tool that is able to automatically generate the mapping layer from the YML description of an application model.

A virtual processor in the mapping layer reads in an application trace from a Kahn process via a trace event queue and dispatches the events to a processing component in the architecture model. The mapping of a virtual processor onto a processing component in the architecture model is freely adjustable, facilitated by the fact that the mapping layer and its mapping onto the architecture model are described in YML. Communication channels—i.e., the buffers in the mapping layer—are also mapped onto the architecture model. In Fig. 3, for example, one buffer is placed in shared memory,[2] while the other buffer is mapped onto a point-to-point FIFO channel between processors 1 and 2.

The mechanism used to dispatch application events from a virtual processor to an architecture model component guarantees deadlock-free scheduling of the application events from different event traces [36]. In this mechanism, computation events are always directly dispatched by a virtual processor to the architecture component onto which it is mapped. The latter schedules incoming events that originate from different event queues according to a given policy (FCFS, round-robin, or customized) and subsequently models their timing consequences. Communication events, however, are not directly dispatched to the underlying architecture model. Instead, a virtual processor that receives a communication event first consults the appropriate buffer at the mapping layer to check whether or not the communication is safe to take place so that no deadlock can occur. Only if it is found to be safe (i.e., for read events, the data should be available and, for write events, there should be room in the target buffer) communication events may be dispatched. As long as a communication event cannot be dispatched, the virtual processor blocks. This is possible because the mapping layer executes in the same simulation as the architecture model. Therefore, both the mapping layer and the architecture model share the same simulation-time domain. This also implies that, each time a virtual processor dispatches an application event (either computation or communication) to an architecture model component, the virtual processor is blocked in simulated time until the event's latency has been simulated by the architecture model. In other words, virtual processors can be seen as abstract representations of application processes at the system architecture level.

When architecture model components are gradually refined to disclose more implementation details, Sesame follows an approach in which the virtual processors at the mapping layer are also refined. The latter is done by incorporating dataflow graphs in virtual processors such that it allows us to perform architectural simulation at multiple levels of abstraction without modifying the application model. Fig. 3 illustrates this dataflow-based refinement by refining the virtual processor for process B with a fictive dataflow graph. In this approach, the application event traces specify *what* a virtual processor executes and *with whom* it communicates, while the internal dataflow graph of a virtual processor specifies *how* the computations and communications take place at the architecture level. In the next section, we provide more insight on how this refinement approach works by explaining the relation between trace transformations for refinement and dataflow actors at the mapping layer.

6 Architecture Model Refinement through Trace Transformations

Refining architecture model components in our system-level simulation framework requires that the application events driving these components should also be refined to match the architectural detail. Since we aim at a smooth transition between different abstraction levels, reimplementing or transforming (parts of) the application models for each abstraction level is undesirable. Instead, Sesame maintains only application models at a high level of abstraction (thereby optimizing the potentials for reuse of application models) and bridges the abstraction gap between application models and underlying architecture models at the mapping layer. As will be explained in this section, bridging this abstraction gap is accomplished by refining the virtual processors in the mapping layer with dataflow actors that transform coarse-grained application events into finer grained events at the desired abstraction level which subsequently drive the architecture model components [45], [46], [47]. In other words, the dataflow graph inside a virtual processor consumes external input (dataflow) tokens that represent high-level computational and communication application events and produces external output tokens that represent the refined architectural events associated with the application events.

Refinement of application events is denoted using *trace transformations* [28] in which the left-hand side contains the coarse-grained application events that need to be refined and the right-hand side the resulting architecture-level events. Furthermore, "$\rightarrow$" symbols in trace transformations denote the "followed by" ordering relation. To give an example, the following trace transformations refine R(ead) and W(rite) application events such that the synchronizations are separated from actual data transfers [28]:

$$R \stackrel{\Theta_{ref}}{\Longrightarrow} cd \rightarrow ld \rightarrow sr, \qquad (1)$$

2. The architecture model accounts for the modeling of bus activity (arbitration, transfers, etc.) when accessing this buffer.

$$W \overset{\Theta_{ref}}{\Longrightarrow} cr \rightarrow st \rightarrow sd. \quad (2)$$

Here, refined architecture-level events *check-data**, *load-data*†, *signal-room**, *check-room**, *store-data*†, and *signal-data** are abbreviated as cd, ld, sr, cr, st, and sd, respectively. The events marked with * refer to synchronizations while those marked with † refer to data transmissions. We further assume that the cd and cr events are blocking, i.e., block until, respectively, data or room is available in a buffer. The above refinements allow, for example, for moving synchronization points or reducing their number when a *pattern* of application events is transformed [28], [45]. Consider, for example, an application process that reads a block of data from an input buffer, performs some computation on it, and writes the results to an output buffer. This would generate an "$R \rightarrow E \rightarrow W$" application-event pattern, in which the E(xecute) refers to the computation on the block of data. Assuming that this application process is mapped onto a processing component that does not have local storage but operates directly on its input and output buffers, we need the following trace transformation:

$$R \rightarrow E \rightarrow W \overset{\Theta_{ref}}{\Longrightarrow} \mathrm{cd} \rightarrow \mathrm{cr} \rightarrow ld \rightarrow E \rightarrow st \rightarrow sr \rightarrow sd. \quad (3)$$

In the refined event sequence, we check early—using the *check-room* (*cr*)—to see if there is room in the output buffer before fetching the data (*ld*) from the input buffer because the processing component cannot temporarily store results locally. In addition, the input buffer must remain available until the processing component has finished operating on it (i.e., after writing the results to the output buffer). Therefore, the *signal-room* (*sr*) is scheduled after the st. Note that this behavior could not have been modeled at the level of the coarse-grained, atomic R(ead) and W(rite) application events.

6.1 Event Refinement Using Dataflow Graphs

In Sesame, Synchronous Data Flow (SDF) [48] and Integer-controlled Data Flow (IDF) [49] actors are deployed to realize trace transformations. As we will explain in this section, the SDF actors perform the actual event refinement while dynamic IDF actors are utilized to model repetitions and branching conditions that are present in the application code [46]. In addition, as illustrated in [45], IDF actors may also be used to achieve less complicated (in terms of the number of actors and channels) dataflow graphs.

Refining application event traces by means of dataflow actors works as follows: For each Kahn process in the application model, an IDF graph is synthesized at the mapping layer and embedded in the corresponding virtual processor. As a result, each virtual processor is equipped with an abstract representation of the application code from its corresponding Kahn process, similar to the concept of Symbolic Programs from [14]. Sesame's IDF graphs consist of static SDF actors (due to the fact that SDF is a subset of IDF) embodying the architecture events that are the—possibly transformed—representation of application events at the architecture level. In addition, to capture the control behavior of the Kahn processes, the IDF graphs also contain dynamic actors for conditional jumps and repetitions. The IDF graphs are executable as the actors have an execution mechanism called *firing rules* which specify when an actor can fire. When firing an actor, it consumes the required tokens from its input token channels and produces a specified number of tokens on its output channels.

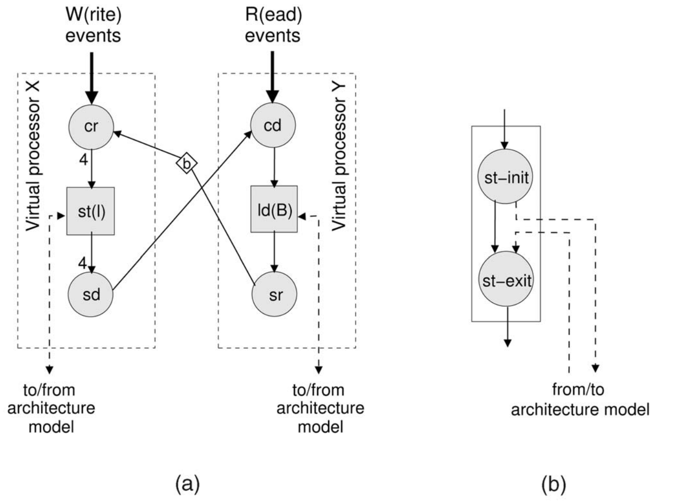

Fig. 4. (a) Refining blocks to lines. (b) Decomposition of the st actor.

To illustrate how SDF actors can be used to transform (i.e., refine) application events, consider the following example in which two Kahn application processes act as a producer-consumer pair communicating pixel blocks: Let us assume that the producing Kahn process is mapped onto an architectural processing element (e.g., IP core) that produces *lines* rather than blocks and where four lines equal one block. Fig. 4a shows the SDF graphs inside virtual processors X and Y that represent the behavior of, respectively, the producer and consumer application processes at the architecture level. The SDF actors inside virtual processor X refine the W(rite) application events—that operate at block level—from the producing Kahn process according to the following transformation:

$$W \overset{\Theta_{ref}}{\Longrightarrow} cr \rightarrow \mathrm{st(l)} \rightarrow \mathrm{st(l)} \rightarrow \mathrm{st(l)} \rightarrow \mathrm{st(l)} \rightarrow sd, \quad (4)$$

where $st(l)$ refers to a "store line." Since the consuming processing element reads entire pixel blocks ($ld(B)$ in Fig. 4a), synchronization is handled at block level, i.e., cr and cd events check for the availability of (room for) entire blocks.

The names of the actors in Fig. 4a represent their functionality. Some actors are depicted by boxes rather than circles to indicate that they are, as will be explained further on, compound actors. If no explicit number is specified at a channel of an actor, then it is assumed that a single token on that particular channel is consumed/produced. Focusing on the SDF graph in virtual processor X in Fig. 4a, the cr actor fires when it receives a W(rite) application event and has at least one token on its channel from the sr actor of virtual processor Y. Here, the $\diamondsuit$ symbol denotes the *delay* of the channel, specifying the initial number of tokens that are present on the channel. This means that a delay of b tokens on the channel to the cr actor in virtual processor X models a FIFO buffer of b elements between virtual processors X and Y. Firing the cr actor produces four tokens for the $st(l)$ actor which subsequently fires four times in a row, where each firing produces a single token. Finally, the sd actor consumes four tokens to fire, after which it produces a token for the cd actor in virtual processor Y to signal the availability of a new pixel block.

A special characteristic of our SDF actors is that they can be coupled (i.e., mapped) to architecture model components. This means that a firing SDF actor may send a token to the architecture model to initiate the simulation of an event. The SDF actor in question is then blocked until it receives an acknowledgment token from the architecture model indicating that the performance consequences of the event have been simulated. In the example of Fig. 4a, the *st* and *ld* actors are mapped onto the architecture model (indicated by the boxed representation of the actors). For these mapped actors, we use special compound actors. Taking the *st* compound actor of virtual processor X as an example, its composition is shown in Fig. 4b. It is composed of an *st-init* and *st-exit* actor. Firing the *st-init* actor produces a token for the architecture model indicating that the performance consequences of the *st* event need to be simulated. When this has been done, the architecture model sends an acknowledgment token to the *st-exit* actor, triggering it to fire and thereby finalizing the firing of the entire compound actor. Because the virtual processors with their dataflow graphs and the underlying architecture model are executed within the same simulation (and, thus, are sharing the virtual clock), the token exchange with the architecture model yields *timed dataflow graphs*. For example, the time delay of the *ld* and *st* actors in Fig. 4a depends on the simulation of these events in the underlying architecture model. Here, we would like to add that synchronization events can also be modeled using the compound actor of Fig. 4b in order to model a latency that is associated with *executing* synchronization operations.

6.2 IDF Graphs for Event Trace Transformations

As mentioned before, we apply dynamic IDF actors (in addition to SDF actors) to capture dynamic behavior in our architecture-level dataflow representation of application behavior. To illustrate how these IDF graphs are constructed and applied for event trace transformation, we use an example taken from a Motion-JPEG encoder application that we studied in [36]. Fig. 5 shows an annotated C++ code fragment from the *Quality-Control* (QC) Kahn process of the Motion-JPEG application. The QC process dynamically computes the tables for Huffman encoding as well as those required for quantizing each frame in the video stream, according to the image statistics and the obtained compression bitrate of the previous video frame. In Fig. 6, an IDF graph for the QC process is given, realizing a high-level (unrefined) simulation. That is, the architecture-level events embodied by the SDF actors directly represent the application-level *R*(ead), *E*(xecute), and *W*(rite) events. The SDF actors drive the architecture model components using the aforementioned token exchange mechanism, although Fig. 6 does not depict the architecture model nor the token exchange channels for the sake of simplicity. Also not shown are the token channels to and from the IDF graphs of neighboring virtual processors with which is communicated. For example, the *R*(ead) actors are, in reality, connected to a *W*(rite) actor from a remote virtual processor in order to signal when data or room is available. The (horizontal) dotted token channels between the SDF actors in Fig. 6 denote dependencies. Adding these token channels to the graph results in sequential execution of architecture-level events, while removing them will allow for exploiting parallelism by the underlying architecture model. The IDF actors CASE-BEGIN, CASE-END, REPEAT-BEGIN, and REPEAT-END model conditional and repetition structures that are present in the application code. Like all models in Sesame, the structure of our IDF graphs is also described using YML.

```
while(1) {
  read(in_NumOfBlocks,NumOfBlocks);

  // code omitted

  write(out_TablesInfo,LumTablesInfo);
  write(out_TablesInfo,ChrTablesInfo);
  switch(TablesChangeFlag) {
    case HuffTablesChanged:
      write(out_HuffTables,LumHuffTables);
      write(out_HuffTables,ChrHuffTables);
      write(out_Command1,OldTables);
      write(out_Command2,NewTables);
      break;
    case QandHuffTablesChanged:

      // code omitted

    default:
      write(out_Command1,OldTables);
      write(out_Command2,OldTables);
      break;
  }

  // code omitted

  for(int i=1;i<(NumOfBlocks/2);i++) {

    // code omitted

    read(in_Statistics,Statistics);
    execute("op_AccStatistics");

    // code omitted
  }
}
```

Fig. 5. An annotated C++ code fragment taken from a Quality-Control (QC) task in a Motion-JPEG encoder application.

In the IDF graphs, scheduling information of actors is not incorporated into the graph definition, but is explicitly supplied by a scheduler. As can be seen in Fig. 6, this scheduler operates on the original application event trace in order to schedule the IDF actors by producing the appropriate control tokens. The actor scheduling is typically done in a dynamic manner. This means that the application and architecture models are cosimulated using a UNIX IPC-based interface to communicate events from the application model to the scheduler, implying that the scheduler only operates on a window of application events. For a more detailed discussion on the scheduling of our IDF actors, we refer the interested reader to [5].

Fig. 7 shows an IDF graph for the QC process that implements the aforementioned communication refinement in which the application-level *R*(ead) and *W*(rite) events are refined such that the synchronization and data-transfer parts become explicit. The computational *E*(xecute) events remain unrefined in this example. We again omitted the token channels to/from IDF graphs of neighboring virtual processors in Fig. 7, but, in reality, *cd* actors have, for example, an incoming token channel from an *sd* actor of a remote IDF graph (as illustrated in Fig. 4). By firing the refined SDF actors (*cd*, *cr*, etc.) in the IDF graph according to the order in which they appear on the right-hand side of a trace transformation—see, for example, transformation (3), noting that the right-hand side may also be specified as a

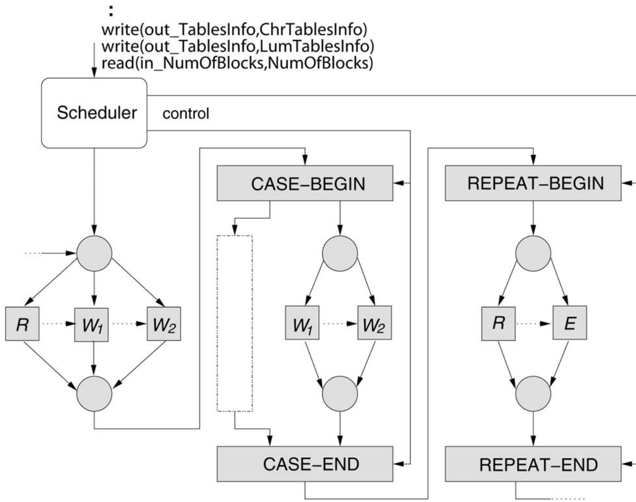

Fig. 6. IDF graph representing the QC task from Fig. 5, realizing high-level (unrefined) simulation at architecture level.

partial ordering [28], [47]—this automatically yields a *valid* schedule for the IDF graph [46]. Here, we also recall that the level of parallelism of the architecture-level events is specified by the presence or absence of token channels between SDF actors. To conclude, communication refinement is accomplished by simply replacing SDF actors with refined ones, allowing for evaluating the performance of different communication behaviors at the architecture level while the application model remains unaffected. As shown in [47] and as we will demonstrate in the next section, this approach allows for refining computational behavior as well.

The IDF-based refinement approach also permits *mixed-level simulations* in which only parts of the architecture model are refined while the other parts remain at the higher level of abstraction. This will be demonstrated in the next section, too. These mixed-level simulations enable a more detailed performance evaluation of a specific architecture component in a system-level context. They therefore avoid the need for building a completely refined architecture model during the early design stages. Moreover, mixed-level simulations do not suffer from deteriorated system evaluation efficiency caused by unnecessarily refined parts of the architecture model.

7 A Motion-JPEG Case Study

In this section, we use the aforementioned Motion-JPEG (M-JPEG) encoder application to illustrate Sesame's modeling methodology and trajectory. More specifically, we demonstrate how Sesame allows a designer to systematically traverse the path from selecting candidate architectures, using analytical modeling and multiobjective optimization, to simulating these candidate architectures with our system-level simulation environment. Subsequently, architectural exploration is performed at different levels of abstraction (facilitated by Sesame's gradual refinement of architecture performance models) while maintaining high-level and architecture-independent application specifications.

The application model of the M-JPEG encoder is shown in Fig. 8. Our M-JPEG encoder differs from traditional encoders in two ways: It can operate on video data in both YUV and

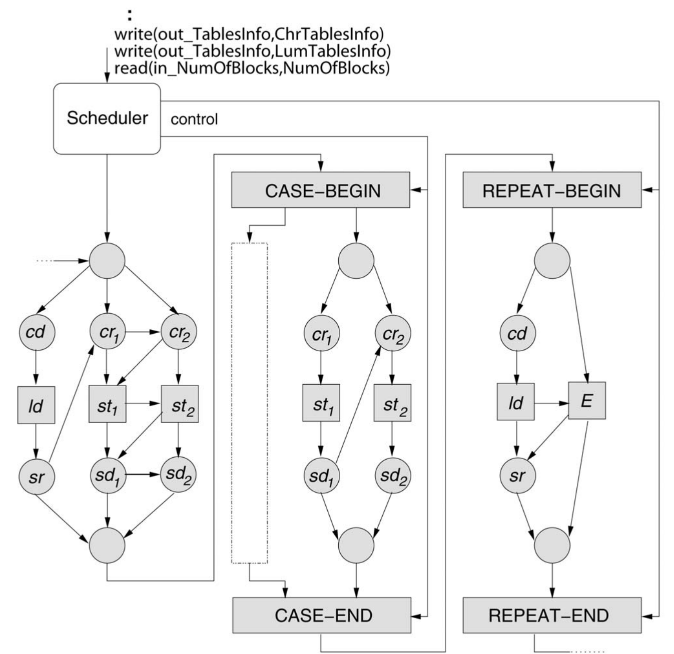

Fig. 7. IDF graph for the QC task realizing communication refinement.

RGB formats on a per-frame basis and it includes dynamic quality control by means of on-the-fly generation of quantization and Huffman tables. Fig. 9 depicts the studied (meta)platform architecture, consisting of five processors and four memories connected by two shared buses and point-to-point links. Using our analytical modeling and multiobjective optimization methods, we intend to find promising instances of this (meta)platform that allow a good mapping (in terms of performance, power, and cost) of the M-JPEG application. Two of these candidate platform instances will then be further studied using system-level simulation.

In Table 1, we provide the processor and memory characteristics that have been used during the multiobjective optimization process. In this exploration phase, we used relative processing capacities (x), power consumption during execution (y) and communication (z), and cost (k) values for each processor and memory in the platform. We have implemented the MMPN problem as a PISA module [50] and subsequently used the already available SPEA2 optimizer to solve it. The mapping may impose some constraints on, e.g., which application process can be mapped onto which processor. Such constraints are specified in the allele sets and constraint violations are handled by a repair mechanism, as was explained in Section 4.1. In Table 2, we show the

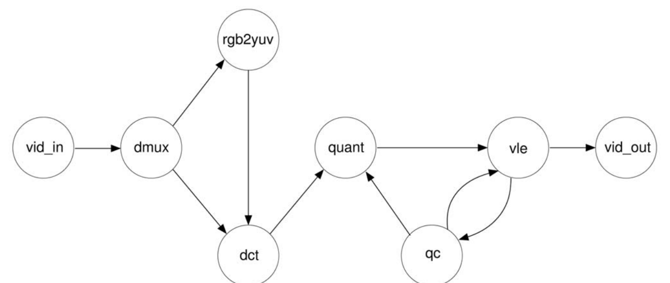

Fig. 8. Application model of the Motion-JPEG encoder.

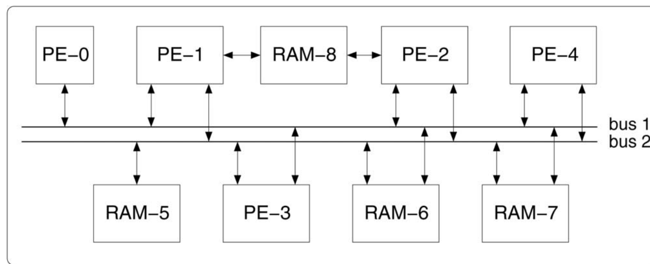

Fig. 9. (Meta)platform architecture model.

configuration used in our experiments. Additional SPEA2 parameters are as follows:

- population size = 100,
- number of generations = 1,000,
- mutation probability = 0.5,
- bit mutation probability = 0.01, and
- crossover probability = 0.8.

In Fig. 10, the nondominated front is shown as obtained by plotting 17 nondominated solutions that were found by SPEA2 in a single run. Before plotting, the objective function values were normalized so that they fall into the $[0, 1]$ interval. The latter normalization procedure is usually applied for better visualization as objective functions scale independently. This particular search took roughly 5 seconds on a 2.8GHz Pentium-4 machine, but search times can be dependent on factors such as the number of elements involved (application processes and channels and architecture resources) and feasibility (e.g., allele sets). For example, in a different study, we have also experimented with larger application models, consisting of up to 26 processes and 75 channels, which required a search time of about 25 seconds on the same Pentium-4 machine.

We have selected two nondominated solutions for further investigation by means of simulation. These solutions and their objective function values are given in Table 3. The *sol 1* solution is a faster implementation which uses three processor cores (PE-1, PE-2, and PE-3), while *sol 2* is a cheaper implementation using only two processing cores (PE-0 and PE-1). In both platform instances, the processors are connected to a single common bus and communicate via shared memory (RAM-6). We modeled these platform instances with our system-level simulation environment, initially using nonrefined architecture model components, i.e., the simulated architecture events are identical to the generated application events. In the architecture models, the processing cores such as PE-0 and PE-1 are modeled by (the performance

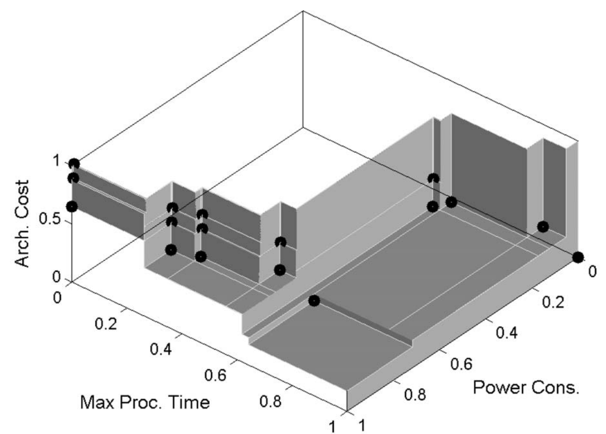

Fig. 10. Nondominated front obtained by SPEA2.

behavior of) representative types of architecture components like microprocessors, DSPs, or ASICs.

The estimated cycle counts for the two platform instances, as derived from system-level simulation, are shown in Table 4. For these experiments, we simulated the encoding of 11 frames with a resolution of 352×288 pixels. The simulations predict that the first target architecture (*sol 1*) will be faster than the second architecture (*sol 2*), which corresponds to the results from the analytical model in Table 3. Besides the cycle count, the simulation produces a multitude of other statistics on, for example, component utilization (see Fig. 12), intercomponent communication, and critical path analysis, of which the latter two are not shown here due to space limitations. Table 4 also shows the wall-clock time for the simulations on a 2.8MHz Pentium-4 machine, which is approximately 64 seconds for both simulations. We further note that, although the absolute cycle counts are not highly relevant in this illustrative case study, we performed several validation experiments in different studies of which the results demonstrate that our high-level models can yield good accuracy [5], [51]. In these experiments, we mapped various application models,

TABLE 1
Processor and Memory Characteristics - I

Processor	Processing cap. (comp,comm)	Power cons. (comp,comm)	Cost
PE-0	(2x,2x)	(y,z)	k
PE-1	(5x,5x)	(4y,3z)	6k
PE-2	(3x,3x)	(3y,3z)	4k
PE-3	(3x,3x)	(3y,3z)	4k
PE-4	(3x,3x)	(3y,3z)	4k
Memory	**Processing cap.**	**Power cons.**	**Cost**
RAM-5	3x	5y	2k
RAM-{6-8}	x	2y	k

TABLE 2
Processor Characteristics - II

Processor	Allele set for the processor
PE-0	DCT
PE-1	QC, DCT, DMUX, QUANT, RGB2YUV, VLE, VID_IN, VID_OUT
PE-2	QC, DCT, DMUX, QUANT, RGB2YUV, VLE, VID_IN, VID_OUT
PE-3	QC, DCT, DMUX, QUANT, RGB2YUV, VLE, VID_IN, VID_OUT
PE-4	QC, DCT, DMUX, QUANT, RGB2YUV, VLE, VID_IN, VID_OUT

TABLE 3
Two Solutions Chosen for Simulation

Solution	Max processing time	Power cons.	Cost
sol 1	129338	1166	160
sol 2	193252	936	90

TABLE 4
Simulation of the Two Selected Platform Architectures

Solution	Estimated cycle count	Wall-clock time (secs)
sol 1	40585833	≈64
sol 2	49816109	≈64

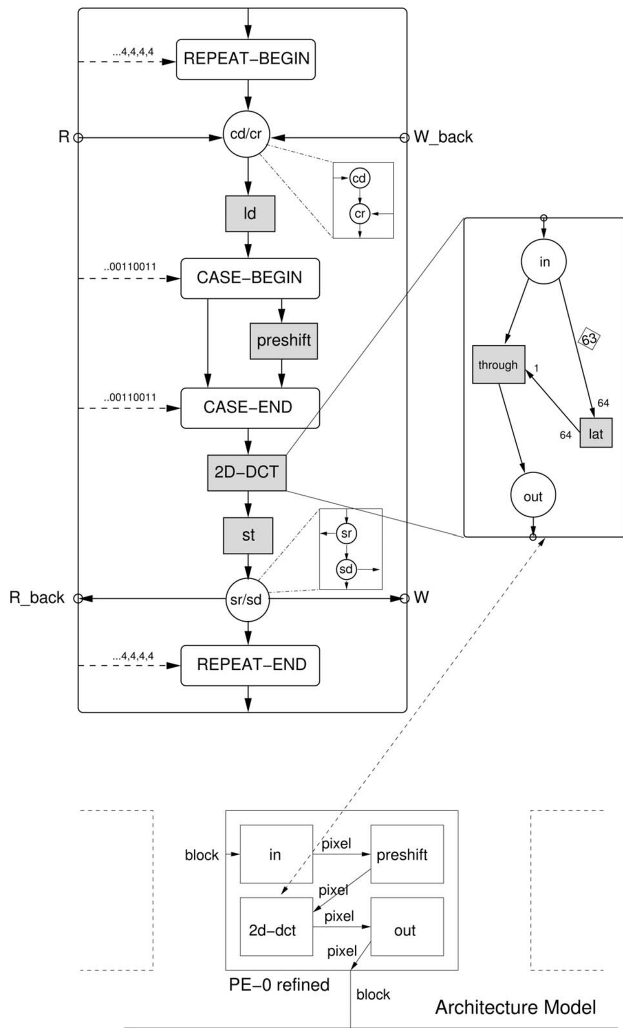

Fig. 11. IDF graph refining the DCT task mapped on a processor without local memory.

including M-JPEG, onto models of existing architecture implementations and compared our performance estimates with the timings of the actual implementations. For these experiments, the errors of our estimations remained below the five percent.

In our subsequent exploration, we mainly focus on the DCT task in the M-JPEG application. This DCT application task operates on half (2:1:1) macroblocks consisting of two luminance (Y) blocks and two chrominance (U and V) blocks. Regarding this DCT, we would like to model more implementation details at the architectural level. To this end, we first refine the PE onto which the DCT task is mapped to reflect the fact that the luminance blocks need to be preshifted before a DCT is performed. This requires a

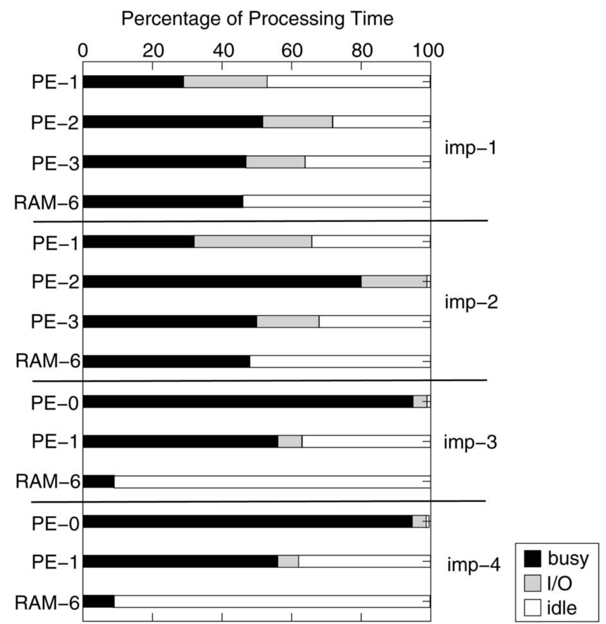

Fig. 12. Simulation results showing the utilization of architecture components.

refinement of the PE's architecture-level events, which now must include a *preshift* event. Since the application DCT task will still generate course-grained DCT events, we need to refine the application event trace in the mapping layer. This is done with the IDF graph shown in the top half of Fig. 11. The IDF scheduler activates this dataflow graph when the incoming application event trace indicates the processing of a half macroblock, i.e., the scheduler has encountered four $R \rightarrow E \rightarrow W$ event sequences that each refer to the processing of a single block inside a 2:1:1 half macroblock. Subsequently, the IDF graph will generate preshift and 2D-DCT architecture-level events for the first two luminance blocks and only 2D-DCT events for the two following chrominance blocks. The gray actors in Fig. 11 indicate that they perform a token exchange with the architecture model, thereby modeling the latency of their action.

The IDF graph also makes synchronization explicit using transformations (1) and (2) from Section 6. Fig. 11 shows the refinement graph for a PE without local memory. This is modeled by checking for room early, as was shown in the trace transformation equation (3). Modeling a PE with local memory is done by simply reordering the synchronization actors such that the check-room (*cr*) occurs just before the store (*st*).

Table 5 gives the performance estimation for both target architectures, with and without local memory for the refined PE. The results in Table 5 show that performance is increased when the PE has local memory to perform its

TABLE 5
Architecture Refinement Results - I

Imp.	Definition	Cycle count
imp-1	sol 1, PE-2 without local memory	43523731
imp-2	sol 1, PE-2 with local memory	41210599
imp-3	sol 2, PE-0 without local memory	47865333
imp-4	sol 2, PE-0 with local memory	47656269

TABLE 6
Architecture Refinement Results - II

Imp.	Definition	Cycle count
imp-5	sol 2, PE-0 without local memory, non-refined (black-box) DCT	29647393
imp-6	sol 2, PE-0 without local memory, refined (pipelined) DCT	29673684

computations. The statistics in Fig. 12 indicate that mapping the application tasks QC and DCT onto separate fast PEs —as is done in solution 1—produces a balanced system. Solution 2 maps the computation intensive DCT task onto less powerful PE-0. This results in a cheaper system, but performance is limited by PE-0. As a result, the memory used for communication is underutilized. This also explains why applying a local memory for PE-0 only marginally improves the performance, as shown in Table 5.

These statistics point to exploring the possibilities of solution 2 with a faster application-specific processing element. The PE we target has two pixel pipelines which allow it to perform a preshift and a 2D-DCT in parallel. The pipelined PE is modeled in two different ways. In the first model, indicated by implementation 5 in Table 6, we reduced the execution latencies (to model a hardware implementation) associated with the preshift and 2D-DCT architecture-level events. These event latencies now approximate the time it takes the pipelines to process a single pixel block. This is a quick way to model the effect a pipelined PE has on the system.

Implementation 6 refines the preshift and 2D-DCT operations at the architecture level as it models their pipelines explicitly. However, it does so in an abstract manner. The top right box in Fig. 11 shows a conceptual view[3] of the dataflow graph refining the 2D-DCT actor to model an abstract pipeline. It models the latency and throughput of the pipeline at pixel level without actually modeling every single stage of the pipeline itself. The architecture-level events from the abstract pipeline model are mapped onto a refined architecture component that allows parallel execution of preshift and 2D-DCT execute events. Table 6 shows that performance estimations of both models are very close, which is not surprising since the abstract pipeline does not model pipeline stalls. Both models indicate a significant performance increase over solution 2 with the original PE. We note that, as a next step, we could model the pipeline in more detail, accurately accounting for pipeline stalls, by explicitly modeling all of the pipeline stages as was done in [47].

Regarding the wall-clock times of the refined simulations from Table 5 and Table 6, the simulation of models *imp-1* up to *imp-5* all take approximately 65 seconds. Only the simulation of implementation 6 takes 120 seconds because this model simulates the DCT operation at the pixel level.

Finally, we would like to mention that the model for implementation 6 yields a mixed-level simulation. This is because PE-0 is refined such that it models (abstract) pipelines at the pixel level while the other model components still account for latencies at the level of processing entire pixel blocks (in accordance to the granularity of the application events). Moreover, during all these experiments, the application model has been reused without any alteration.

8 Conclusions

In this paper, we presented an overview of the Sesame framework which provides high-level modeling and simulation methods and tools for system-level performance evaluation and exploration of heterogeneous SoC-based embedded media systems. We described Sesame's modeling methodology and trajectory in which a designer first selects candidate architectures using analytical modeling and multiobjective optimization. Subsequently, these candidate architectures can be simulated using Sesame's system-level simulation environment. This simulation environment allows for architectural exploration at different levels of abstraction while maintaining high-level and architecture-independent application specifications. This is accomplished by the fact that Sesame bridges the abstraction gap between application and architecture models by applying dataflow graphs in its intermediate mapping layer. These dataflow graphs take care of the runtime transformation of coarse-grained application-level events into finer grained architecture-level events that drive the architecture model components. We have illustrated all these aspects using a case study in which we traverse Sesame's exploration trajectory for a Motion-JPEG encoder application.

3. Some details have been omitted, like the translation of blocks into pixels. The interested reader is referred to [5] for more details.

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. The authors would like to thank all Sesame and Artemis group members for their contributions to this work. Special credits go to Paul Lieverse, Bart Kienhuis, Todor Stefanov, Ed Deprettere, Kees Vissers, Pieter van der Wolf, and Vladimir Živković for their ground-laying work with respect to the modeling methodology applied in Sesame.

References

[1] F. Vahid and T. Givargis, "Platform Tuning for Embedded Systems Design," *Computer*, vol. 34, no. 3, pp. 112-114, Mar. 2001.
[2] A. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design and Test of Computers*, vol. 18, no. 6, pp. 23-33, 2001.
[3] K. Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523-1543, Dec. 2000.
[4] A.D. Pimentel, P. Lieverse, P. van der Wolf, L.O. Hertzberger, and E.F. Deprettere, "Exploring Embedded-Systems Architectures with Artemis," *Computer*, vol. 34, no. 11, pp. 57-63, Nov. 2001.
[5] A.D. Pimentel, "The Artemis Workbench for System-Level Performance Evaluation of Embedded Systems," *Int'l J. Embedded Systems*, vol. 1, no. 7, 2005.
[6] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli, *Hardware-Software Co-Design of Embedded Systems—The POLIS Approach.* Kluwer Academic, 1997.
[7] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An Integrated Electronic System Design Environment," *Computer,* vol. 36, no. 4, pp. 45-52, Apr. 2003.
[8] A. Mihal, C. Kulkarni, C. Sauer, K. Vissers, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, and S. Malik, "Developing Architectural Platforms: A Disciplined Approach," *IEEE Design and Test of Computers,* vol. 19, no. 6, pp. 6-16, Nov./Dec. 2002.
[9] A. Cassidy, J. Paul, and D. Thomas, "Layered, Multi-Threaded, High-Level Performance Design," *Proc. Int'l Conf. Design, Automation and Test in Europe (DATE),* Mar. 2003.
[10] S. Mohanty and V.K. Prasanna, "Rapid System-Level Performance Evaluation and Optimization for Application Mapping onto SoC Architectures," *Proc. IEEE Int'l ASIC/SOC Conf.,* 2002.

[11] T. Kogel, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, D. Bussaglia, and M. Ariyamparambath, "Virtual Architecture Mapping: A SystemC Based Methodology for Architectural Exploration of System-on-Chip Designs," *Proc. Int'l Workshop Systems, Architectures, Modeling, and Simulation (SAMOS)*, pp. 138-148, 2003.

[12] M. Gries, "Methods for Evaluating and Covering the Design Space during Early Design Development," *Integration, the VLSI J.*, vol. 38, no. 2, pp. 131-183, 2004.

[13] K. Lahiri, A. Raghunathan, and S. Dey, "System-Level Performance Analysis for Designing On-Chip Communication Architectures," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 6, pp. 768-783, June 2001.

[14] V. Živković, E.F. Deprettere, P. van der Wolf, and E. de Kock, "Fast and Accurate Multiprocessor Architecture Exploration with Symbolic Programs," *Proc. Int'l Conf. Design, Automation and Test in Europe (DATE)*, Mar. 2003.

[15] P. Lieverse, P. van der Wolf, E.F. Deprettere, and K.A. Vissers, "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems," *J. VLSI Signal Processing for Signal, Image, and Video Technology*, vol. 29, no. 3, pp. 197-207, Nov. 2001.

[16] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int'l J. Computer Simulation*, vol. 4, pp. 155-182, Apr. 1994.

[17] C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, vol. 21, no. 8, Aug. 1978.

[18] E.A. Lee and T.M. Parks, "Dataflow Process Networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773-801, May 1995.

[19] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. IFIP Congress 74*, 1974.

[20] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli, "A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures," *Proc. Design Automation Conf. (DAC)*, June 2002.

[21] R.P. Dick and N.K. Jha, "MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-Synthesis of Distributed Embedded Systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Oct. 1998.

[22] T. Blickle, J. Teich, and L. Thiele, "System-Level Synthesis Using Evolutionary Algorithms," *Design Automation for Embedded Systems*, vol. 3, no. 1, pp. 23-58, 1998.

[23] G. Ascia, V. Catania, and M. Palesi, "A GA-Based Design Space Exploration Framework for Parameterized System-on-a-Chip Platforms," *IEEE Trans. Evolutionary Computation*, vol. 8, no. 4, pp. 329-346, 2004.

[24] T. Givargis and F. Vahid, "Platune: A Tuning Framework for System-on-a-Chip Platforms," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1317-1327, 2002.

[25] T. Givargis, F. Vahid, and J. Henkel, "System-Level Exploration for Pareto-Optimal Configurations in Parameterized System-on-a-Chip," *IEEE Trans. Very Large Scale Integration Systems*, vol. 10, no. 4, pp. 416-422, 2002.

[26] J. Peng, S. Abdi, and D. Gajski, "Automatic Model Refinement for Fast Architecture Exploration," *Proc. Int'l Conf. VLSI Design*, pp. 332-337, Jan. 2002.

[27] S. Abdi, D. Shin, and D. Gajski, "Automatic Communication Refinement for System Level Design," *Proc. Design Automation Conf. (DAC)*, pp. 300-305, June 2003.

[28] P. Lieverse, P. van der Wolf, and E.F. Deprettere, "A Trace Transformation Technique for Communication Refinement," *Proc. Int'l Symp. Hardware/Software Codesign (CODES)*, pp. 134-139, Apr. 2001.

[29] G. Nicolescu, S. Yoo, and A.A. Jerraya, "Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in SoC Design," *Proc. Int'l Conf. Design, Automation and Test in Europe (DATE)*, Mar. 2001.

[30] J.Y. Brunel, E.A. de Kock, W. Kruijtzer, H. Kenter, and W. Smits, "Communication Refinement in Video Systems on Chip," *Proc. Int'l Workshop Hardware/Software Codesign (CODES)*, pp. 142-146, May 1999.

[31] J. Rowson and A. Sangiovanni-Vincentelli, "Interface-Based Design," *Proc. Design Automation Conf. (DAC)*, June 1997.

[32] B. Kienhuis, E.F. Deprettere, K.A. Vissers, and P. van der Wolf, "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures," *Proc. Int'l Conf. Application-Specific Systems, Architectures, and Processors (ASAP)*, July 1997.

[33] B. Kienhuis, E.F. Deprettere, P. van der Wolf, and K.A. Vissers, "A Methodology to Design Programmable Embedded Systems: The Y-Chart Approach," *Embedded Processor Design Challenges*, pp. 18-37, Springer, 2002.

[34] C. Erbas, S.C. Erbas, and A.D. Pimentel, "A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Process Networks," *Proc. Int'l Conf. HW/SW Codesign and System Synthesis (CODES-ISSS)*, pp. 182-187, Oct. 2003.

[35] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization," *Evolutionary Methods for Design, Optimisation, and Control*, pp. 95-100, Barcelona: CIMNE, 2002.

[36] A.D. Pimentel, S. Polstra, F. Terpstra, A.W. van Halderen, J.E. Coffland, and L.O. Hertzberger, "Towards Efficient Design Space Exploration of Heterogeneous Embedded Media Systems," *Embedded Processor Design Challenges*, pp. 57-73, Springer, 2002.

[37] J.E. Coffland and A.D. Pimentel, "A Software Framework for Efficient System-Level Performance Evaluation of Embedded Systems," *Proc. ACM Symp. Applied Computing (SAC)*, pp. 666-671, Mar. 2003, http://sesamesim.sourceforge.net/.

[38] A. Turjan, B. Kienhuis, and E.F. Deprettere, "Translating Affine Nested Loop Programs to Process Networks," *Proc. Int'l Conf. Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, Sept. 2004.

[39] T. Stefanov and E.F. Deprettere, "Deriving Process Networks from Weakly Dynamic Applications in System-Level Design," *Proc. Int'l Conf. HW/SW Codesign and System Synthesis (CODES-ISSS)*, Oct. 2003.

[40] E.A. Lee and S. Neuendorffer, "MoML—a Modeling Markup Language in XML, version 0.4," Technical Report UCB/ERL M00/8, Electronics Research Lab, Univ. of California, Berkeley, Mar. 2000.

[41] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," *Proc. Int'l Conf. HW/SW Codesign and System Synthesis (CODES-ISSS)*, pp. 19-24, Oct. 2003.

[42] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic, 2002.

[43] H.L. Muller, "Simulating Computer Architectures," PhD thesis, Dept. of Computer Science, Univ. of Amsterdam, Feb. 1993.

[44] M. Thompson and A.D. Pimentel, "A High-Level Programming Paradigm for SystemC," *Proc. Int'l Workshop Systems, Architectures, MOdeling, and Simulation (SAMOS)*, pp. 530-539, July 2004.

[45] A.D. Pimentel and C. Erbas, "An IDF-Based Trace Transformation Method for Communication Refinement," *Proc. Design Automation Conf. (DAC)*, pp. 402-407, June 2003.

[46] C. Erbas and A.D. Pimentel, "Utilizing Synthesis Methods in Accurate System-Level Exploration of Heterogeneous Embedded Systems," *Proc. IEEE Workshop Signal Processing Systems (SiPS)*, pp. 310-315, Aug. 2003.

[47] C. Erbas, S. Polstra, and A.D. Pimentel, "IDF Models for Trace Transformations: A Case Study in Computational Refinement," *Proc. Int'l Workshop Systems, Architectures, MOdeling, and Simulation (SAMOS)*, pp. 178-187, July 2003.

[48] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235-1245, Sept. 1987.

[49] J.T. Buck, "Static Scheduling and Code Generation from Dynamic Dataflow Graphs with Integer Valued Control Streams," *Proc. Asilomar Conf. Signals, Systems, and Computers*, Oct. 1994.

[50] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA—A Platform and Programming Language Independent Interface for Search Algorithms," *Proc. Evolutionary Multi-Criterion Optimization (EMO 2003)*, pp. 494-508, 2003.

[51] A.D. Pimentel, F.P. Terpstra, S. Polstra, and J.E. Coffland, "On the Modeling of Intra-Task Parallelism in Task-Level Parallel Embedded Systems," *Domain-Specific Processors*, pp. 85-105, Marcel Dekker, 2003.

Andy D. Pimentel received the MSc and PhD degrees in computer science from the University of Amsterdam, where he is currently an assistant professor in the Informatics Institute. He is member of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). His research interests include computer architecture, computer architecture modeling and simulation, system-level design, design space exploration, performance analysis, embedded systems, and parallel computing. He is a member of the IEEE Computer Society.

Cagkan Erbas received the BSc degree in electrical engineering from Middle East Technical University, Ankara, and the MSc degree in computer engineering from Ege University, Izmir. He is currently a PhD student in computer science at the University of Amsterdam. His research interests include embedded systems, hardware/software codesign, multiobjective search algorithms, and metaheuristics. He is a student member of the IEEE.

Simon Polstra received the MSc degree in computer science from the University of Amsterdam. Currently, he is a member of the Computer Systems Architecture Group at the University of Amsterdam. He has previously been active as an engineer at the National Aerospace Lab in The Netherlands. His research interests include computer architecture and abstract system modeling.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.