The Artemis workbench for system-level performance evaluation of embedded systems

Andy D. Pimentel

Computer Systems Architecture Group,
Informatics Institute, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
E-mail: andy@science.uva.nl

Abstract: In this paper, we present an overview of the Artemis workbench, which provides modelling and simulation methods and tools for efficient performance evaluation and exploration of heterogeneous embedded multimedia systems. More specifically, we describe the Artemis system-level modelling methodology, including its support for gradual refinement of architecture performance models as well as for calibration of the system-level models. We show that this methodology allows for architectural exploration at different levels of abstraction while maintaining high-level and architecture independent application specifications. Moreover, we illustrate these modelling aspects using a case study with a Motion-JPEG application.

Keywords: system-level modelling and simulation; model refinement; performance evaluation; design space exploration; model calibration.

Reference to this paper should be made as follows: Pimentel, A.D. (2008) 'The Artemis workbench for system-level performance evaluation of embedded systems', *Int. J. Embedded Systems*, Vol. 3, No. 3, pp.181–196.

Biographical notes: Andy D. Pimentel received the MSc and PhD Degrees in Computer Science from the University of Amsterdam, where he currently is an Assistant Professor in the Informatics Institute. He is co-founder of the *International Workshop on Systems, Architectures, Modelling, and Simulation (SAMOS)* and is member of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). His research interests include computer architecture, computer architecture modelling and simulation, system-level design, design space exploration, performance analysis, embedded systems, and parallel computing. He is a member of the IEEE Computer Society.

1 Introduction

Designers of modern embedded systems are faced with a number of emerging challenges. Because embedded systems are mostly targeted for mass production and often run on batteries, they should be cheap to realise and power efficient. In addition, these systems increasingly need to support multiple applications and standards for which they should provide high, and sometimes real-time, performance. For example, digital televisions or mobile devices have to support different standards for communication and coding of digital contents. On top of this, modern embedded systems should be flexible so that they can easily be extended to support future applications and standards. Such flexible support for multiple applications calls for a high degree of programmability. However, performance requirements and constraints on cost and power consumption require substantial parts of these systems to be implemented in dedicated hardware blocks. As a result, modern embedded systems often have a *heterogeneous system architecture*, i.e., they consist of components ranging from Programmable processor cores to fully dedicated hardware components for the time-critical application tasks. Increasingly, such heterogeneous systems are integrated on a single chip. This yields heterogeneous multi-processor Systems-on-Chip (*SoCs*) that exploit task level Parallelism in applications.

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 a new design methodology, namely *system-level design*. Below, we briefly describe three important ingredients of system-level design approaches.

Platform architectures

Platform-based design (Vahid and Givargis, 2001; Sangiovanni-Vincentelli and Martin, 2001) has become a popular design method as it stresses the re-use of Intellectual Property (IP) 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 customising a chip for every application.

Separation of concerns

To even further improve the potentials for re-use of IP and to allow for effective exploration of alternative design solutions, it is widely recognised that the 'separation of concerns' is a crucial component in system-level design (Keutzer et al., 2000). Two common types of separation in the design process are:

- separating computation from communication by connecting IP processing cores via a standard (message-passing) network interface (Benini and de Micheli, 2002)
- separating application (what is the system supposed to do) from architecture (how it does it) (Kienhuis et al., 1997; Balarin et al., 1997).

High-level modelling and simulation early in the design

In system-level design, designers already start with modelling and simulating (possible) system components and their interactions in the early design stages (Pimentel et al., 2001). More specifically, system-level models typically represent application behaviour, 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 minimising the modelling effort and optimising simulation speed that is needed for targeting the early design stages. This high-level modelling allows for early verification of a design and can provide estimations on the performance (e.g., Balarin et al., 1997; Pimentel et al., 2001), power consumption (e.g., Brooks et al., 2000) or cost (DeBardelaben et al., 1997) of the design.

Design space exploration plays a crucial role in system-level design of embedded system (platform) architectures. Due to the systems' complexity, it is imperative to have good performance evaluation tools for efficiently exploring a wide range of design choices during the early design stages. In this paper, we present an overview of the Artemis workbench, which provides high-level modelling and simulation methods and tools for efficient performance evaluation and exploration of heterogeneous embedded multimedia systems (Pimentel et al., 2001). More specifically, we describe the Artemis system-level modelling methodology -which deploys the aforementioned principle of separation of concerns (Keutzer et al., 2000) – and particularly focus on the support for gradual refinement of architecture performance models as well as on the calibration of system-level models. We show that this methodology allows for architectural exploration at different levels of abstraction while maintaining high-level and architecture independent application specifications. Furthermore, we illustrate these modelling aspects using a case study with a Motion-JPEG encoder application.

The remainder of the paper is organised as follows. The next section provides a birds eye view of the Artemis workbench, briefly discussing the different tool-sets that are integrated in the workbench. Section 3 focuses on Artemis' system-level modelling and simulation techniques by describing its prototype modelling and simulation environment, called Sesame. In Section 4, we explain how Artemis facilitates gradual refinement of system-level architecture models by applying dataflow graphs. Section 5 illustrates the discussed modelling and refinement techniques using a case study with a Motion-JPEG encoder application. This section also demonstrates how our system-level models can be calibrated with low-level implementation information using an automated component synthesis approach. Finally, Section 6 discusses related work, after which Section 7 concludes the paper.

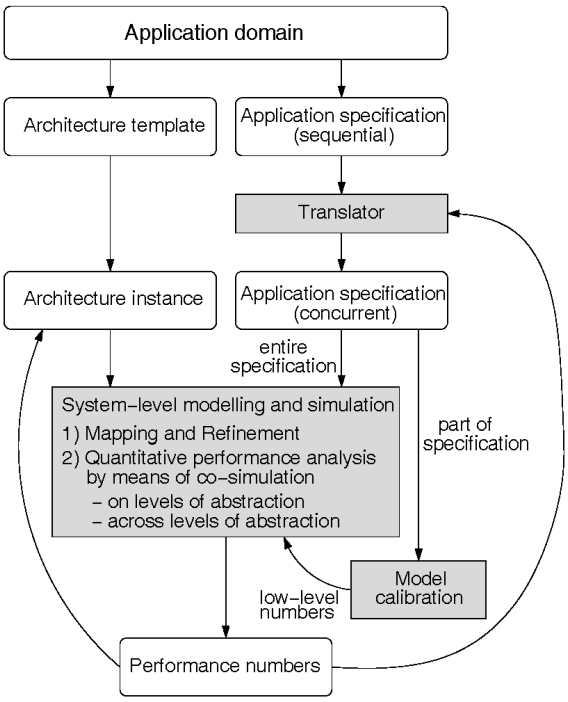
2 The Artemis workbench

The Artemis workbench consists of a set of methods and tools conceived and integrated in a framework to allow designers to model applications and SoC-based (multiprocessor) architectures at a high level of abstraction, to map the former onto the latter, and to estimate performance numbers through co-simulation of application and architecture models. Figure 1 depicts the flow of operation for the Artemis workbench, where the grey parts refer to the various tool-sets that together embody the workbench. The point of departure is an application domain (being multimedia applications for Artemis), an experimental domain-specific platform architecture and a domain-specific application specified as an executable sequential program. The platform architecture is instantiated in the architecture model layer of the workbench, while the application specification is converted to a functionally equivalent concurrent specification using a translator called Compaan (Turjan et al., 2004; Stefanov et al., 2004; Deprettere et al., 2002). More specifically, Compaan transforms the sequential application specification into a Kahn Process Network (KPN) (Kahn, 1974). In between the application and architecture layers there is a mapping layer. This mapping layer provides means to perform quantitative performance analysis on levels of abstraction, and to refine application specification components between levels of abstraction. Such refinement is required to match application specifications to the level of detail of the underlying architecture models. Effectively, the mapping layer bridges the *gap* between the application and architecture (models), sometimes referred to as the implementation gap (Mihal and Keutzer, 2003). In the next sections, we will elaborate on all of the above modelling, mapping, and refinement aspects in more detail.

Because Artemis operates at a high level of abstraction, low-level component performance numbers can be used to *calibrate* the system-level architecture models. To this end, individual processes (i.e., code segments) of a KPN application specification can be taken apart and implemented as individual low-level components (that appear in the current high-level instance of the platform architecture). This results in performance numbers – as well as in estimations on cost and power consumption – for the low-level components that the system-level modelling framework needs to provide accurate performance

estimations for the multiprocessor system architecture as a whole. For this calibration process, the Artemis workbench uses the Laura toolset (Zissulescu et al., 2003; Stefanov et al., 2004) and the Molen calibration platform architecture (Vassiliadis et al., 2001, 2003a, 2003b). Before presenting more details on Artemis' system-level modelling and simulation techniques, which forms the bulk of this paper, the remainder of this section first takes a closer look at the Compaan[1] and Laura[1] tool-sets as well as the Molen[1] calibration platform.

Figure 1 The infrastructure of the Artemis workbench



2.1 The Compaan and Laura tool-sets

Today, traditional imperative languages like C, C++ or Matlab are still dominant with respect to implementing applications for SoC-based (platform) architectures. It is, however, very difficult to map these imperative implementations, with typically a sequential model of computation, onto multi-processor SoC architectures that allow for exploiting task-level parallelism in applications. In contrast, models of computation that inherently express task-level parallelism in applications and make communications explicit, such as CSP (Hoare, 1978) and Process Networks (Lee and Parks, 1995; Kahn, 1974), allow for easier mapping onto multi-processor SoC architectures. However, specifying applications using these models of computation usually requires more implementation effort in comparison to sequential imperative solutions.

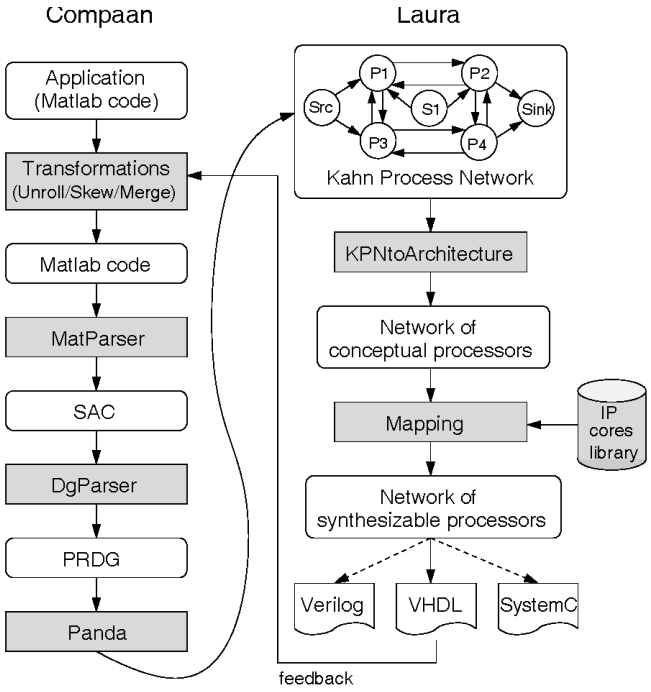
In Artemis, we use an approach in which we start from a sequential imperative application specification – more specifically an application written in a subset of Mat-lab – which is then automatically converted into a KPN (Kahn, 1974) using the Compaan tool-set (Turjan et al., 2004; Stefanov et al., 2004; Deprettere et al., 2002). This conversion is fast and correct by construction. In the KPN model of computation, parallel processes communicate with each other via unbounded FIFO channels. Reading from channels is done in a blocking manner, while writing to channels is non-blocking. We decided to use KPNs for application specifications 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, as will be discussed later on, mapping KPN processes onto SoC architecture models for quantitative performance analysis.

The infrastructure of the Compaan tool-set is illustrated on the left-hand side of Figure 2. The grey parts refer to the separate tools that are part of Compaan, while the white parts refer to the (intermediate) formats of the application specification. Starting-point is an application specification in Matlab, which needs to be specified as a parameterised static nested loop program. Recently, Compaan's scope has been extended to also include weakly-dynamic nested loop programs that allow for specifying data-dependent behaviour (Stefanov and Deprettere, 2003). On these Matlab application specifications, various source-level transformations can be applied in order to, for example, increase or decrease the amount of parallelism in the final KPN (Stefanov et al., 2002). In a next step, the Matlab code is transformed into Single Assignment Code (SAC), which resembles the Dependence Graph (DG) of the original nested loop program. Hereafter, the SAC is converted to a Polyhedral Reduced Dependency Graph (PRDG) data structure, being a compact mathematical representation of a DG in terms of polyhedra. Finally, a PRDG is converted into a KPN by associating a KPN process with each node in the PRDG. The parallel KPN processes communicate with each other according to the data dependencies given in the DG.

The Laura tool-set (Zissulescu et al., 2003; Stefanov et al., 2004), depicted on the right-hand side of Figure 2, takes a KPN as input and produces synthesisable VHDL code that implements the application specified by the KPN for a specific FPGA platform. To this end, the KPN specification is first converted into a functionally equivalent network of conceptual processors, called *hardware model.* This hardware model, which is platform independent as no information on the target FPGA platform is incorporated, defines the key components of the architecture and their attributes. It also defines the semantic model, i.e., how the various components interact with each other. Subsequently, platform specific information is added to the hardware model. This includes the addition of IP cores that implement certain functions in the original application as well as setting attributes of components such as bit-width and buffer sizes. In the final step, the hardware model is converted into VHDL. To do so, Laura supplies a piece of VHDL code for each component in the hardware model that expresses how to represent that component in the target architecture.

Using commercial tools, the VHDL code can then be synthesised and mapped onto an FPGA. As can be seen in Figure 2, the results from this automated implementation trajectory can be fed back to Compaan to explore different transformations that will, in the end, lead to different implementations.

Figure 2 The Compaan (left) and Laura (right) tool-sets.

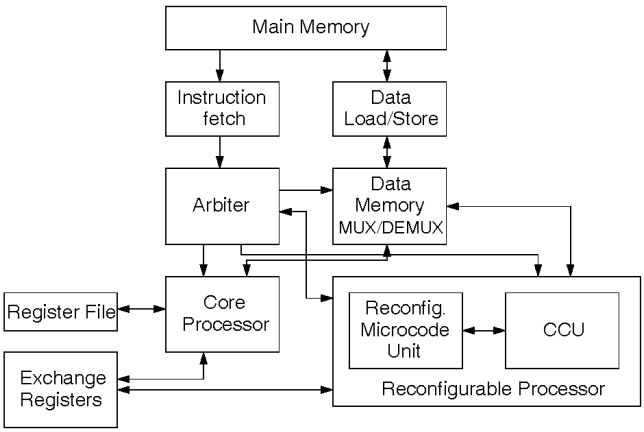


2.2 *The Molen calibration platform*

Figure 3 depicts the platform architecture that is used for model calibration in Artemis. This platform architecture, called Molen (Vassiliadis et al., 2001, 2003a, 2003b), connects a programmable processor with a reconfigurable unit and uses microcode to incorporate architectural support for the reconfigurable unit. Instructions are fetched from the memory, after which the arbiter performs a partial decoding on the instructions to determine where they should be issued (Kuzmanov and Vassiliadis, 2003). Those instructions that have been implemented in fixed hardware are issued to the Core Processing (CP) unit, which is one of the PowerPCs from a Xilinx Virtex II Pro™ platform in the Molen prototype implementation (Kuzmanov et al., 2004), while instructions for custom execution are redirected to the reconfigurable unit. The instructions entering the CP unit are further decoded and then issued to their corresponding functional units.

The reconfigurable unit consists of a Custom Configured Unit (CCU), currently implemented by the Xilinx Virtex II Pro™ FPGA, and a $\rho\mu$-code unit. The reconfigurable unit performs *operations* that can be as simple as an instruction or as complex as a piece of code describing a certain function. Molen divides an operation into two distinct phases: *set* and *execute.* The *set* phase is responsible for reconfiguring the CCU hardware, enabling the execution of an operation. Such a phase may be subdivided into two sub-phases, namely partial-set (*p-set*) and complete-set (*c-set*). The *p-set* phase covers common functions of an application or set of applications. Subsequently, the *c-set* sub-phase only reconfigures those blocks in the CCU which are not covered in the *p-set* sub-phase in order to complete the functionality of the CCU.

Figure 3 The Molen calibration platform architecture



To perform the actual reconfiguration of the CCU, *reconfiguration microcode* is loaded into the $\rho\mu$-*code* unit and then executed (using *p-set* and *c-set* instructions) (Kuzmanov et al., 2003). Hereafter, the *execute* phase is responsible for the operation execution on the CCU, performed by executing the *execution microcode.* Important in this respect is the fact that both the *set* and *execute* phases do not explicitly specify a certain operation to be performed. Instead, the *p-set, c-set* and *execute* instructions point to the memory location where the reconfiguration or execution microcode is stored.

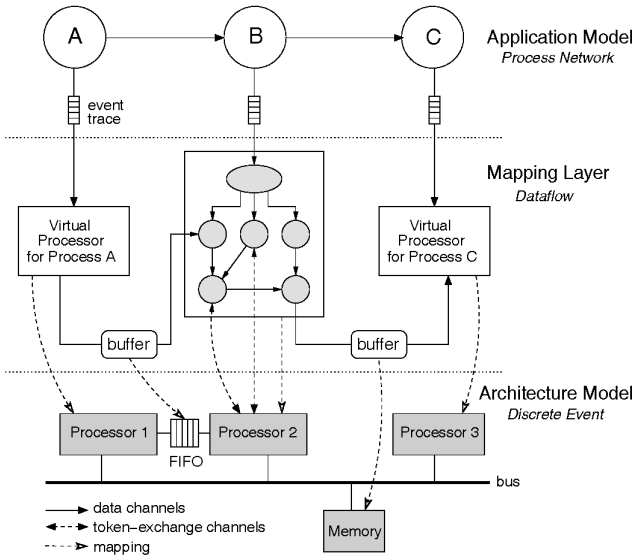
The Compaan and Laura tool-sets in combination with the Molen platform architecture provide great opportunities for the previously discussed calibration of system-level architecture models. For this purpose, Laura maps a specific component from an application specification to a hardware implementation by converting the Compaan generated KPN associated with the application component to a VHDL implementation. This VHDL code is subsequently used as reconfiguration microcode for Molen's CCU, while the remainder of the application specification (i.e., the code that has not been synthesised to a hardware implementation) is executed on Molen's Core Processor. As a result, the application component mapped onto the CCU provides low-level implementation numbers that can be used to calibrate the corresponding component in the system-level architecture model. In Section 3.4, we present a case study in which this model calibration is illustrated for a DCT task in a Motion-JPEG encoder application.

3 The Sesame environment

Artemis' system-level modelling and simulation environment, called Sesame (Pimentel et al., 2002; Coffland and Pimentel, 2003), builds upon the ground-laying work of

the Spade framework (Lieverse et al., 2001c). This means that Sesame facilitates performance analysis of embedded systems architectures according to the Y-chart design approach (Kienhuis et al., 1997; Balarin et al., 1997), recognising *separate* application and architecture models within a system simulation. An application model describes the functional behaviour of an application, including both computation and communication behaviour. An architecture model defines architecture resources and captures their performance constraints. After explicitly mapping an application model onto an architecture model, they are co-simulated via trace-driven simulation. This allows for evaluation of the system performance of a particular application, mapping, and underlying architecture. Essential in this modelling methodology is that an application model is independent from architectural specifics, assumptions on hardware/software partitioning, and timing characteristics. As a result, 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 system architectures or simply modelling the same system architecture at various levels of abstraction. The layered infrastructure of Sesame is shown in Figure 4.

Figure 4 Sesame's infrastructure



3.1 Application modelling

For application modelling (de Kock et al., 2000), Sesame uses KPN application specifications that are generated by the Compaan toolset or have been derived by hand. The computational behaviour of an application is captured by instrumenting the code of each Kahn process with annotations that describe the application's computational actions. The reading from or writing to Kahn channels represents the communication behaviour of a process within the application model. By executing the Kahn model, each process records its actions in order to generate its own trace of application events, which is necessary for driving an architecture model. These application events typically are coarse grained, such as *execute(DCT)* or *read(channeLid, pixel-block)*.

To execute Kahn application models, and thereby generating 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 as separate threads using the Pthreads package. Currently, the Kahn processes need to be written in C++, but C and Java support is also planned for the future. 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, but instead, it is described in a language called YML (Y-chart Modelling Language) (Coffland and Pimentel, 2003). This is an XML-based language which is similar to Ptolemy's MoML (Lee and Neuendorffer, 2000) 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 parameterised YML component descriptions that can be instantiated with the appropriate parameters, thereby fostering re-use of YML descriptions. To simplify the use of YML even further, a YML editor has also been developed to compose model descriptions using a GUI. Figure 5 gives an impression of the YML editor's GUI, showing its layered layout that corresponds to the three layers of Sesame (see Figure 4), namely the application model layer, mapping layer and architecture model layer.

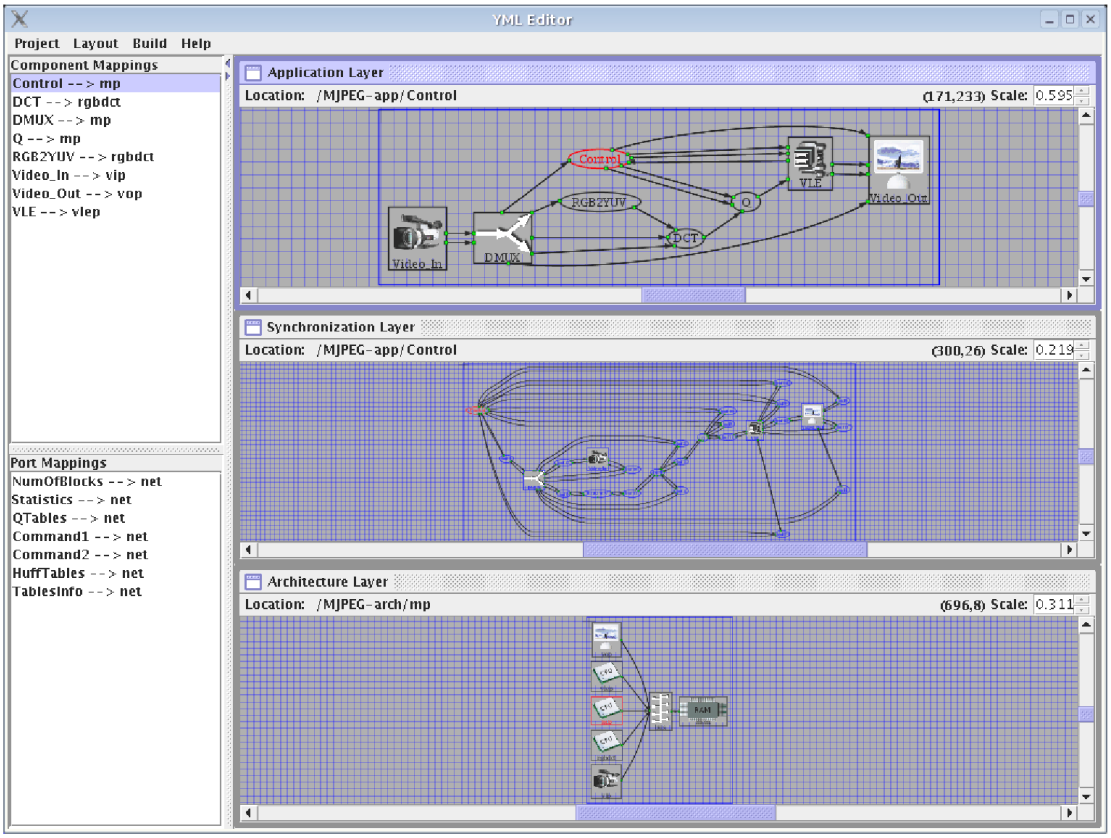
3.2 Architecture modelling

Architecture models in Sesame, which typically operate at the so-called transaction level (Cai and Gajski, 2003; Grötker et al., 2002), 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 behaviour. This is possible because the functional behaviour 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 busses) 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 (Muller, 1993) or SystemC (http://www.systemc.org/, Grötker et al., 2002). Pearl is a small but powerful discrete-event simulation language which provides

easy construction of the models and fast simulation (Pimentel et al., 2002). For our SystemC architecture models, we provide an add-on library to SystemC, called SCPEx (SystemC Pearl Extension) (Thompson and Pimentel, 2004), 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 modelling effort required for developing transaction-level architecture models and making the modelling process less prone to programming errors.

Figure 5 A screenshot of the YML editor (see online version for colours)



3.3 *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 from different event traces 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 parameterised and dependent on the modelled architecture. As the structure of the mapping layer closely resembles 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 (and manipulated using the YML editor, see Figure 5). Communication channels – i.e., the buffers in the mapping layer – are also mapped onto the architecture model. In Figure 4, 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 for dispatching application events from a virtual processor to an architecture model component guarantees deadlock-free scheduling of the application events from different event traces. 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 application events that originate from different event queues according to a given policy and subsequently models their timing consequences. The scheduling of application events supports a range of predefined policies, like FCFS and

round-robin, but can also easily be customised with alternative policies. Communication events are, however, not directly dispatched by a virtual processor. Rather, a virtual processor first consults the appropriate buffer at the mapping layer to check whether or not a 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), then communication events may be dispatched to the processor component in the architecture model. 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 a component in the architecture model, the virtual processor is blocked in simulated time until the event's latency has been simulated by the architecture model.

When architecture model components are gradually refined to include more implementation details, the virtual processors at the mapping layer are also refined. The latter is done with dataflow graphs such that it allows us to perform architectural simulation at multiple levels of abstraction without modifying the application model. Figure 4 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.

In a closely related project, called Archer (Živković et al., 2002), an alternative mapping approach is studied. That is, while Archer and Sesame share quite a few application and architecture modelling techniques, Archer uses a different mapping strategy than Sesame. In Archer, Control Data Flow Graphs (CDFG) (Wolf, 2001) are taken as a basis. However, as the CDFG notation is too complex for design space exploration, the CDFGs are lifted to a higher abstraction level, called Symbolic Programs (SP) (Živković et al., 2003b). The SPs, which in Archer are automatically derived from a KPN application specification, are CDFG-like representations of the Kahn processes. They contain control constructs like CDFGs, but unlike CDFGs, they are not directly executable since SPs only contain symbolic instructions (i.e., application events) and no real code. Therefore, SPs need extra information for execution to determine the control flow within an SP, which is supplied in terms of *control traces*. These control traces are generated by running the application with a particular set of data. At the architecture layer, SPs are executed with the control traces to generate event traces which are subsequently used to drive the resources in the architecture model. Like Sesame, Archer also supports the refinement of architecture models. It does so by transforming application-level SPs into architecture-level SPs (Živković et al., 2003a).

3.4 Mapping support

To facilitate effective design space exploration, Sesame provides some (initial) support for finding promising candidate application-to-architecture mappings to guide a designer during the system-level simulation stage. To this end, we have developed a mathematical model that captures several trade-offs faced during the process of mapping (Erbas et al., 2003). In this model, we take into account the computational and communication demands of an application as well as the properties of an architecture, in terms of computational and communication performance, power consumption, and cost. The resulting trade-offs with respect to performance, power consumption and cost are formulated as a multi-objective combinatorial optimisation problem. Using an optimisation software tool, which is based on a widely-known evolutionary algorithm (Zitzler, 1999), the mapping decision problem is solved by providing the designer with a set of approximated Pareto-optimal mapping solutions that can be further evaluated using system-level simulation. For a more detailed description of this mapping support, the interested reader is referred to (Erbas et al., 2003).

4 Architecture model refinement

Refining architecture model components in Sesame requires that the application events driving them should also be refined to match the architectural detail. Since we aim at a smooth transition between different abstraction levels, re-implementing 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 optimising 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 the 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 are subsequently used to drive the architecture model components (Pimentel and Erbas, 2003; Erbas and Pimentel, 2003; Erbas et al., 2003). In other words, the dataflow graphs consume external input (dataflow) tokens that represent high-level computational and communication application events and produce external output tokens that represent the refined architectural events associated with the application events.

Refinement of application events is denoted using *trace transformations* (Lieverse et al., 2001b), 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 synchronisations are separated from actual data transfers (Lieverse et al., 2001b):

$$R \overset{\Theta_{ref}}{\Rightarrow} cd \rightarrow ld \rightarrow sr \quad (1)$$

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

Here, refined architecture-level events *check-data*, load-data*†*, signal-room*, check-room*, store-data*†*), signal-data** are abbreviated as *cd, ld, sr, cr, st, sd,* respectively. The events marked with * refer to synchronisations while those marked with † refer to data transmissions. The above refinements allow for, for example, moving synchronisation points or reducing their number when a *pattern* of application events is transformed (Lieverse et al., 2001b; Pimentel and Erbas, 2003). 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 a '$R \rightarrow E \rightarrow W$' 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:

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

In the refined event sequence, we early check — using the *check-room* (*cr*) – 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*.

In Sesame, Synchronous Data Flow (SDF) (Lee and Messerschmitt, 1987) actors are deployed to realise trace transformations. Integer-controlled Data Flow (IDF) (Buck, 1994) actors are subsequently utilised to model repetitions and branching conditions which may be present in the application code (Erbas and Pimentel, 2003). However, as illustrated in Pimentel and Erbas (2003), they may also be used within static transformations 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 at the application layer, an IDF graph is synthesised 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 Živković et al. (2002). 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 control behaviour 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. A special characteristic of our IDF graphs is that the SDF actors are tightly coupled with the architecture model components (Pimentel and Erbas, 2003). 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 acknowledgement token from the architecture model indicating that the performance consequences of the event have been simulated within the architecture model. To give an example, an SDF actor that embodies a *write* event will block after firing until the *write* has been simulated at the architecture level.

In IDF graphs, scheduling information of IDF actors is not incorporated into the graph definition but is explicitly supplied by a scheduler. This scheduler operates on the original application event traces in order to schedule our IDF actors. The actor scheduling can be done either in a semi-static or dynamic manner. In dynamic scheduling, the application and architecture models are co-simulated using a UNIX IPC-based interface to communicate events from the application model to the scheduler. As a consequence, the scheduler only operates on a window of application events which implies that the IDF graphs cannot be analysed at compile-time. This means that, for example, it is not possible to decide at compile-time whether an IDF graph will complete its execution in finite time, or whether the execution can be performed with bounded memory. Alternatively, we can also schedule the IDF actors in a semi-static manner. To do so, the application model should first generate the entire application traces and store them into trace files (if their size permits this) prior to the architectural simulation. This static scheduling mechanism is a well-known technique in Ptolemy (Buck et al., 1994) and has been proven to be very useful for system simulation (Buck, 1994). However, in Sesame, it does not yield to a fully static scheduling. This is because of the fact that, as was previously explained, our SDF actors have a token exchange mechanism with the underlying architecture model, yielding some dynamic behaviour.

We also intend to investigate whether or not our IDF graphs can be specified as so-called *well-behaved dataflow* graphs (Gao et al., 1992). In these well-behaved dataflow graphs dynamic actors are only used as a part of two predefined clusters of actors – known as schemas – that allow for modelling conditional and repetitive behaviour. The resulting graphs have, as opposed to regular IDF graphs, many of the same attractive properties with respect to static analysis as graphs composed only of SDF actors.

To illustrate how IDF graphs are constructed and applied for event refinement, we use an example taken from a Motion-JPEG encoder application we studied in Lieverse et al. (2001a) and Pimentel et al. (2002). Figure 6 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 quantising each frame in the video stream, according to the image statistics and the obtained compression bitrate of the previous video frame. In Figure 7, an IDF graph for the QC process is given, realising a high-level (unrefined) simulation. That is, the architecture-level events embodied by the SDF actors (depicted as circles) directly represent the application-level *R*(ead), *E*(xecute) and *W*(rite) events. The SDF actors drive the architecture model components by the aforementioned token exchange mechanism, although Figure 7 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 neighbouring 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 is available and when room is available. The IDF actors CASE-BEGIN, CASE-END, REPEAT-BEGIN, and REPEAT-END model conditional and repetition structures that are present in the application code. The scheduler reads the application trace from the Kahn process in question and executes the IDF graph by scheduling the IDF actors accordingly by sending the appropriate control tokens. In Figure 7, there are (horizontal) dotted token channels between the SDF actors, denoting 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. Like all models in Sesame, the structure of our IDF graphs is also described using YML.

In Figure 8, an IDF graph for the QC process is shown that implements the aforementioned communication refinement in which the application-level *R*(ead) and *W*(rite) events are refined such that the synchronisation 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 neighbouring virtual processors in Figure 8, but in reality *cd* actors have, for example, an incoming token channel from an *sd* actor of a remote IDF graph. 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 (equation (3)), noting that the right-hand side may also be specified as a partial ordering (Lieverse et al., 2001b; Erbas et al., 2003) – this automatically yields a *valid* schedule for the IDF graph (Erbas and Pimentel, 2003). Here, we also recall that the level of parallelism between 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 to evaluate the performance of different communication behaviours at architecture level while the application model remains unaffected. As shown in Erbas et al. (2003) and like we will demonstrate in the next section, this approach allows for refining computational behaviour as well.

Figure 6 An annotated C++ code fragment

```
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
  }
}
```

Figure 7 IDF graph for high-level (unrefined) simulation

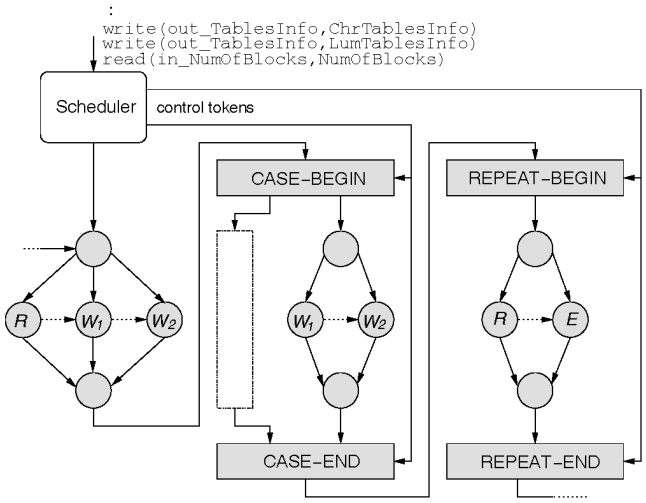
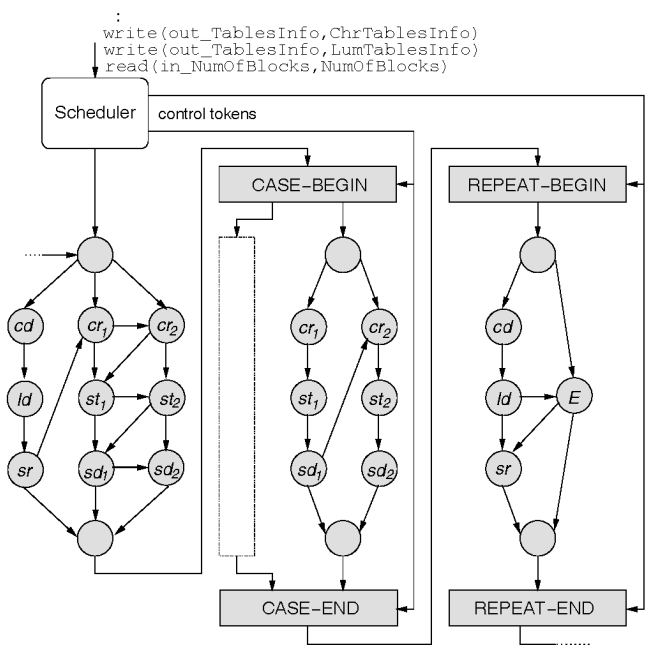


Figure 8 IDF graph realising communication refinement



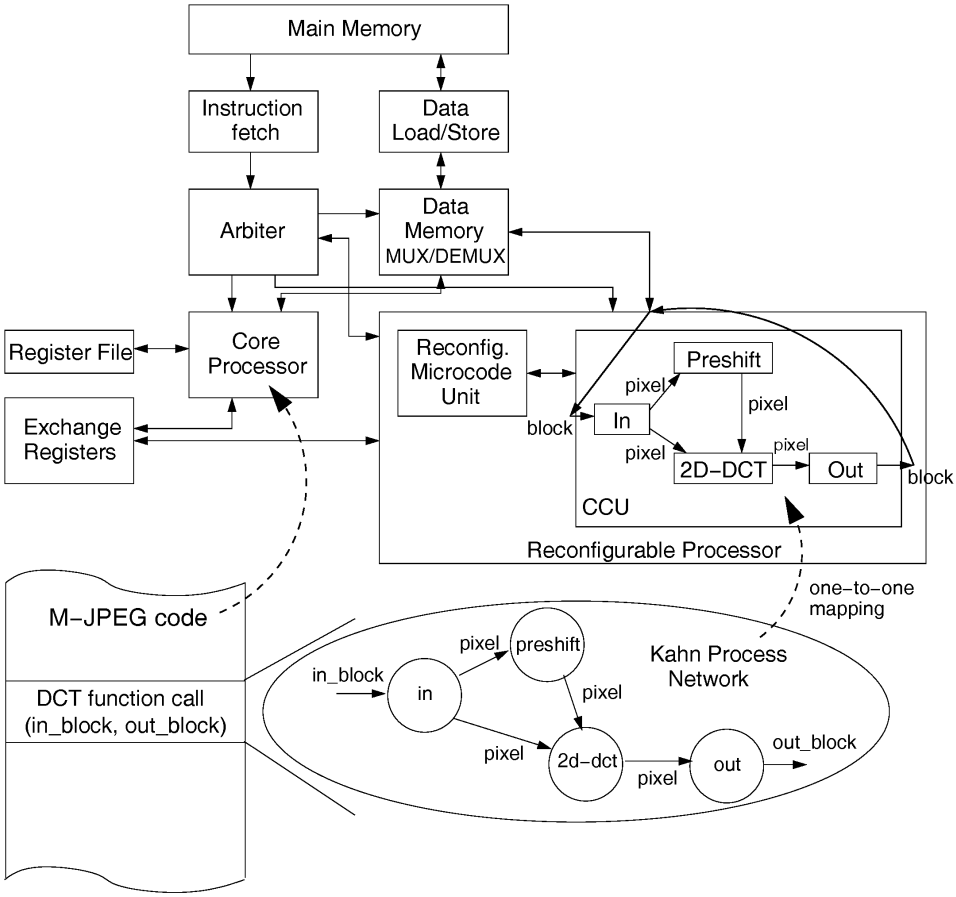
The IDF-based refinement approach also permits *mixed-level simulations,* in which only parts of the architecture model are refined while the other parts remain at the higher level of abstraction. This will be demonstrated in the next section too. These mixed-level simulations enable more detailed performance evaluation of a specific architecture component in the context of the behaviour of the whole system. 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.

5 Motion-JPEG case study

This section presents an experiment that illustrates some of the important aspects of Artemis' flow of operation as depicted in Figure 1. More specifically, using the Motion-JPEG (M-JPEG) encoder application from the previous section, we demonstrate how model calibration can be performed by means of the Compaan/Laura tool-sets and the Molen platform. Furthermore, we describe the system-level modelling and simulation aspects of the M-JPEG experiment, emphasising on the IDF-based architecture model refinement that was performed.

In the experiment, we selected the DCT task from the M-JPEG application to be used for model calibration. This means that the DCT task is taken 'all the way down' to a hardware implementation in order to study its low-level performance aspects. To do so, the following steps were taken, which are integrally shown in Figure 9. The DCT was first isolated from the sequential M-JPEG code and used as input to the Compaan tool-set. Subsequently, Compaan generated a KPN application specification for the DCT task. This DCT KPN is internally specified at pixel level but has in- and output tasks that operate at the level of pixel blocks because the original M-JPEG application specification also operates at this block level. Using the Laura tool-set, the KPN for the DCT task was translated into a VHDL implementation, in which for example the 2D-DCT component is implemented as a 92-stage pipelined IP block. This implementation can subsequently be mapped onto the FPGA (i.e., CCU) of the Molen platform. By mapping the remainder of the M-JPEG code onto Molen's (CP), we were able to study the hardware DCT implementation in the context of the M-JPEG application. As will be explained later, the results of this exercise have been used to calibrate our system-level architecture modelling. Although being out of scope for this paper, it might be worth mentioning that the M-JPEG encoder with FPGA-implemented DCT obtained a 2.14 speedup – out of a 2.5 maximum attainable theoretical speedup – in comparison to a full software implementation. For the system-level modelling and simulation part of the experiment, we decided to model the Molen calibration platform architecture itself. This gives us the opportunity to actually validate our performance estimations against the real numbers from the implementation. The resulting system-level Molen model contains two processing components (Molen's CP and CCU) which are bi-directionally connected using two uni-directional FIFO buffers. Like in the real Laura → Molen mapping, we mapped the DCT Kahn process from our M-JPEG application model onto the CCU component in the architecture model, whereas the remaining Kahn processes were mapped onto the CP component. We also decided to refine the CCU in our architecture model such that it models the pixel-level DCT implementation used in the Compaan/Laura implementation. The CP component in our architecture model was not refined, implying that it operates (i.e., models timing consequences) at the same (pixel-block) level as the application events it receives from the application model. Hence, this yields a mixed-level simulation. We would also like to stress that the M-JPEG application model was not changed for this experiment. This means that the application events for the CCU component, referring to DCT operations on entire pixel blocks, needed to be refined to pixel-level events. In addition, at the architecture model level, the execution of these pixel-level events required to be modelled according to the pipelined execution semantics of the actual implementation. This because the Preshift and 2D-DCT blocks in the Laura-generated implementation are pipelined units.

Figure 9 Model calibration for the DCT task

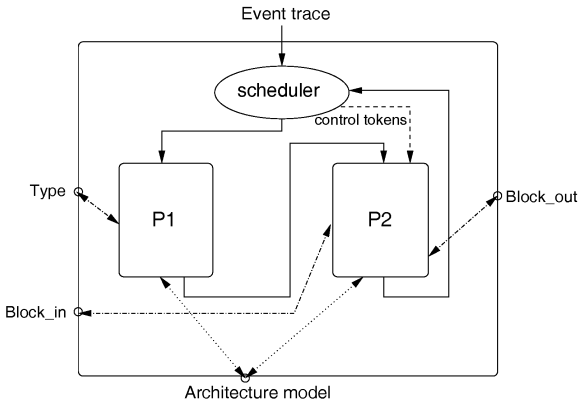


According to what was explained in the previous section, we accomplished the refinement of the CCU component in the architecture model by refining the virtual processor associated with the DCT Kahn process in the mapping layer, as this is the virtual processor that is mapped onto the CCU component. The resulting IDF graph that is embedded in the virtual processor has several levels of hierarchy, of which the top level is shown in Figure 10.

The top-level IDF graph consists of the actor scheduler and two actors, called P1 and P2. These two actors refer to the two alternating application-event patterns that the DCT process generates. One of the patterns (denoted by actor P1) results from the DCT process finding out the location (i.e., which input buffer) of the next half macro-block[3] that needs to be processed, while the other pattern (denoted by actor P2) results from the actual processing (reading, executing, and writing) of a half macro-block. As we are not interested in the first application-event pattern, actor P1 is not further refined. The channels labelled with *Type, Block-in* and *Block-out* in Figure 10 refer to the token channels to and from the remote virtual processors with which is communicated. The two dotted double-headed arrows represent the token exchange channels connected to the architecture model for modelling the latencies associated with actor firings, as was explained in the previous section.

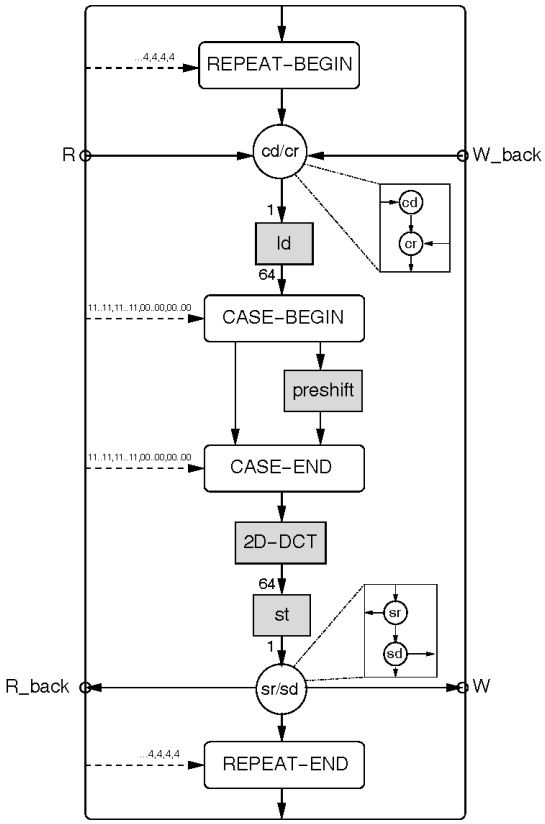
In Figure 11, we zoom in on actor P2, showing the internal IDF graph of this composite actor. Actor P2 is fired each time the scheduler at the top level (see Figure 10) recognises the processing of a half macro-block from the incoming application event trace. So, this implies that actor P2 describes the architectural behaviour of processing a half macro-block. To do so, P2 first models the processing of single pixel blocks from a half macro-block using the REPEAT actors. The REPEAT actors receive control tokens from the scheduler specifying that a half macro-block consists of four pixel blocks (2Y, 1U, 1V). For every pixel block, it is first checked whether or not the data is available in the input buffer (*cd*) and room is available to store results in the output buffer (*cr*). Subsequently, we model the reading of the pixel block from the input buffer by means of the *ld* actor, which generates 64 output tokens when fired. These tokens represent the separate pixels inside a pixel block. Here, grey actors mean that they perform a token exchange with the underlying architecture model, thereby modelling the latency of their action. According to the Compaan/Laura implementation of the DCT task (see Figure 9), we model the execution of the *preshift* and *2D-DCT* at the pixel level. Using the CASE actors, we select pixels from only the two Y blocks inside a half macro-block to fire the *preshift* actor. Next, the 2D-DCT operation is modelled for every pixel, described in more detail further on. Finally, the pixels are stored in the output buffer (*st*), and the input and output buffers are signalled that, respectively, room is available again (*sr*) and data is available (*sd*).

Figure 10 Virtual processor for DCT Kahn process



As mentioned before, the *preshift* and *2D-DCT* components in the Compaan/Laura implementation of the DCT are pipelined units. We model the pipelined execution semantics of our *preshift* and *2D-DCT* actors by embedding another SDF graph in them that models an abstract pipeline. Figure 12 depicts this abstract pipeline model for the *2D-DCT* composite actor. It models the latency and throughput behaviour of the pipeline when assuming that no pipeline bubbles occur within the processing of a single pixel block. We would like to note that we also could have modelled the pipeline in more detail, accurately accounting for pipeline stalls, by explicitly modelling all of the pipeline stages, like was done in Erbas et al. (2003). This is relatively easy using YML, which allows us to describe models in a repetitive manner using loop-like constructs.

Figure 11 IDF graph for actor P2 from Figure 10



For each pixel in a pixel block, the *in* actor in our abstract pipeline model fires a token to the *lat* and *through* actors. The token channel between the *in* and *lat* actor contains an initial number of 63 tokens. This means that after the first pixel from a pixel block, the *lat* actor will fire. This actor performs a token exchange with the underlying architecture model, where the latency of the *lat* actor equals to 91 cycles. Since the *2D-DCT* -component pipeline in reality contains 92 stages, this 91-cycle latency means that we model the first pixel from the pixel block traversing through the pipeline until the last stage. After this, the *through* actor will be fired 64 times, each with a latency of a single cycle, representing the 64 pixels leaving the pipeline one after the other.

Notably, we have been using low-level information – such as pipeline depth of units, latencies for reading/writing a pixel from/to a buffer and so on – from the Compaan/Laura/Molen implementation to calibrate our system-level model. To check whether or not the resulting model, which was calibrated with low-level information, produces accurate performance estimates at the system level, we compared the performance of the M-JPEG encoder application executed on the real Molen platform with the results from our system-level performance model. Table 1 shows the validation results for a sequence of sample input frames.

Figure 12 SDF graph modelling an abstract pipeline

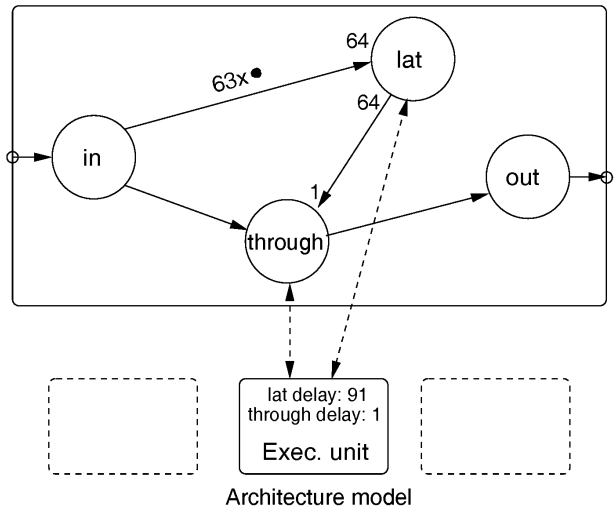


Table 1 Validation results of M-JPEG experiment

	Real Molen (*cycles*)	*Sesame simulation* (*cycles*)	*Error* (%)
Full SW implementation	84581250	85024000	0.5
DCT mapped onto CCU	39369970	40107869	1.9

The results from Table 1 include both the cases in which all application tasks are performed in software (i.e., they are mapped onto Molen's CP) and in which the DCT task is mapped onto Molen's CCU. Here, we would like to stress

that we did not perform any tuning of our system-level model with Molen's M-JPEG execution results. As can be seen from the results, Sesame's system-level performance estimations are relatively accurate. This indicates that our technique for architecture model refinement, facilitating architectural exploration while keeping the application model unchanged, shows significant promise.

6 Related work

There are a number of architectural exploration environments, such as (Metro)Polis (Balarin et al., 1997, 2003), Mescal (Mihal et al., 2002), and Milan (Mohanty and Prasanna, 2002), and various SystemC-based environments like the work of Kogel et al. (2003), that facilitate flexible system-level performance evaluation by providing support for mapping a behavioural application specification to an architecture specification. In Artemis, we try to push the separation of modelling application behaviour and modelling architectural constraints at the system level to even greater extents. This is achieved by architecture-independent application models, application-independent architecture models and a mapping step that relates these models for (trace-driven) co-simulation. Moreover, within Artemis, we use multiple models of computation, specifically chosen in accordance with the task to be achieved. As already shown in this paper, we use process networks for application modelling, dataflow networks for certain tasks at the mapping layer (e.g., trace transformations) and a discrete-event simulator for fast simulation of our architecture models.

The work of Lahiri et al. (2001) also uses a trace-driven approach, but this is done to extract communication behaviour for studying on-chip communication architectures. Rather than using the traces as input to an architecture simulator, their traces are analysed statically. In addition, a traditional hardware/software co-simulation stage is required in order to generate the traces. The Archer project (Živković et al., 2002, 2003b), which was already mentioned before, shows a lot of similarities with the Sesame framework. This is due to the fact that both Sesame and Archer stem from the earlier Spade project (Lieverse et al., 2001c). A major difference is, however, that Archer follows a different application-to-architecture mapping approach. Instead of using event-traces, it maps Symbolic Programs, which are derived from the application model, onto architecture model resources.

Ptolemy (Buck et al., 1994) is an environment for simulation and prototyping of heterogeneous systems. It allows for using multiple models of computation within a single system simulation. It does so by supporting domains to build sub-systems each conforming to a different model of computation. Ptolemy supports an increasing set of models of computation, including discrete event models, finite state machine models, CSP (Hoare, 1978) models, and many types of dataflow models (Lee and Parks, 1995): Synchronous Dataflow, Boolean Dataflow, Integer-controlled Dataflow, Dynamic Dataflow, as well as (Kahn) Process Networks (Kahn, 1974).

Calibration of high-level simulation models using more accurate lower-level simulations is a well-known technique. For a system-level architecture model, this could, for example, mean that an instruction-set simulator is used to calibrate an abstract (system-level) model of a programmable processor (e.g., Mohanty and Prasanna, 2002). Although we have not addressed such traditional model calibration in this paper, it is applicable to Artemis. In addition to that, Artemis also allows for selecting an application task after which the Compaan/Laura tool-chain automatically maps this task to an FPGA-based implementation. Such an automated implementation trajectory can rapidly produce valuable low-level information for calibrating our system-level models.

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 (Balarin et al., 2003) and Milan frameworks (Mohanty and Prasanna, 2002), the work of (Peng et al., 2002), and in the context of SystemC (e.g., Kogel et al., 2003). In Peng et al. (2002), for example, a methodology is proposed in which architecture-independent specification models are transformed (i.e., refined) into architecture models to facilitate architectural exploration. Although being 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 during this refinement process the separation between application and architecture is retained as much as possible to allow effective exploration of alternative solutions. In addition to this, the majority of the work in this field has focused on communication refinement only. For example, in Abdi et al. (2003), Lieverse et al. (2001b), Nicolescu et al. (2001), Brunel et al. (1999), Nieuwland and Lippens (1998) and Rowson and Sangiovanni-Vincentelli (1997), various mechanisms are proposed for the refinement of application level communication primitives into more detailed implementation (architecture) primitives.

7 Conclusions

In this paper, we provided an overview of the Artemis workbench, which allows designers to model (multimedia) applications and SoC-based (multiprocessor) architectures at a high level of abstraction, to map the former onto the latter, and to estimate performance numbers through co-simulation of application and architecture models. Moreover, we presented an approach for calibrating our (system-level) architecture performance models with low-level information derived from an automated implementation trajectory that can map specific application components onto an FPGA platform. A significant part of this paper was however dedicated to the architecture model refinement methodology of Artemis. We explained how

Artemis bridges the abstraction gap between application and architecture models by applying dataflow actors in the intermediate mapping layer, transforming coarse-grained application events into finer grained architecture events that drive the architecture model components. This event refinement technique allows for architectural exploration at different levels of abstraction while maintaining high-level and architecture independent application models. Using an experiment with a Motion-JPEG encoder application, we illustrated the system-level modelling, model refinement and model calibration aspects of the Artemis workbench.

Credits

A large number of people are responsible for, or have contributed to, the work described in this paper. The Compaan and Laura tool-sets have been developed at Leiden University by the group of Ed Deprettere. Main contributors of these tools-sets are Alexandru Turjan, Bart Kienhuis, Edwin Rijpkema, Todor Stefanov, Claudiu Zissulescu, and Ed Deprettere. The Molen platform has been designed and developed at Delft University of Technology by the group of Stamatis Vassiliadis. We especially would like to mention the following Molen contributors: Stephan Wong, Georgi Kuzmanov, Georgi Gaydadjiev and Stamatis Vassiliadis. The Sesame modelling and simulation framework has been developed at the University of Amsterdam by the group of Andy Pimentel. The main contributors of Sesame are: Berry van Halderen, Simon Polstra, Frank Terpstra, Joseph Cofnand, Cagkan Erbas, and Andy Pimentel. In addition, we would like to give credit to Paul Lieverse, Bart Kienhuis, Ed Deprettere, Kees Vissers, Pieter van der Wolf, and Vladimir Zivkovic for their ground-laying work with respect to the modelling methodology applied in Artemis.

Acknowledgements

This research is supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW. We thank Alexandru Turjan, Claudiu Zissulescu, Todor Stefanov, Georgi Kuzmanov, Cagkan Erbas and Simon Polstra for providing valuable input to this paper.

References

Abdi, S., Shin, D. and Gajski, D. (2003) 'Automatic communication refinement for system level design', *Proc. Design Automation Conference (DAC)*, June, pp.300–305.

Balarin, F., Sentovich, E., Chiodo, M., Giusto, P., Hsieh, H., Tabbara, B., Jurecska, A., Lavagno, L., Passerone, C., Suzuki, K. and Sangiovanni-Vincentelli, A. (1997) *Hardware-Software Co-Design of Embedded Systems – The POLIS Approach*, Kluwer Academic Publishers.

Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C. and Sangiovanni-Vincentelli, A. (2003) 'Metropolis: an integrated electronic system design environment', *IEEE Computer*, Vol. 36, No. 4, April.

Benini, L. and de Micheli, G. (2002) 'Networks on chips: a new SoC paradigm', *IEEE Computer*, Vol. 35, No. l, January, pp.70–80.

Brooks, D., Tiwari, V. and Martonosi, M. (2000) 'Wattch: a framework for architectural-level power analysis and optimizations', *Proc. Int. Symposium on Computer Architecture (ISCA)*, June.

Brunel, J.Y., de Kock, E.A., Kruijtzer, W.M., Kenter, H.J.H.N. and Smits, W.J.M. (1999) 'Communication refinement in video systems on chip', *Proc. Int. Workshop on Hardware/Software C'odesign (CODES)*, May, pp.142–146.

Buck, J.T. (1994) 'Static scheduling and code generation from dynamic dataflow graphs with integer valued control streams', *Proc. Asilomar Conference on Signals, Systems, and Computers*, October.

Buck, J.T., Ha, S., Lee, E.A. and Messerschmitt, D.G. (1994) 'Ptolemy: a framework for simulating and prototyping heterogeneous systems', *Int. Journal of Computer Simulation*, Vol. 4, April, pp.155–182.

Cai, L. and Gajski, D. (2003) 'Transaction level modeling: An overview', *Proc. Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, October, pp.19–24.

Coffland, J.E. and Pimentel, A.D. (2003) 'A software frame work for efficient system-level performance evaluation of embedded systems', *Proc. ACM Symp. on Applied Computing (SAC)*, pp.666–671, March. http://sesamesim. sourceforge.net/

de Kock, E.A., Essink, G., Smits, W.J.M., van der Wolf, P., Brunei, J.Y., Kruijtzer, W.M., Lieverse, P. and Vissers, K.A. (2000) 'Yapi: application modeling for signal processing systems', *Proc. Design Automation Conference (DAC)*, June, pp.402–405.

DeBardelaben, J., Madisetti, V. and Gadient, A. (1997) 'Incorporating cost modeling into embedded-system design', *IEEE Design and Test of Computers*, September, Vol. 14, No. 3.

Deprettere, E.F., Rijpkema, E. and Kienhuis, B. (2002) 'Translating imperative affine nested loop programs to process networks', in Deprettere, E.F., Teich, J. and Vassiliadis, S. (Eds.): *Embedded Processor Design Challenges*, Springer, LNCS 2268, pp.89–111.

Erbas, C., Erbas, S.C. and Pimentel, A.D. (2003) 'A multi-objective optimization model for exploring multiprocessor mappings of process networks', *Proc. Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, October, pp.182–187.

Erbas, C. and Pimentel, A.D. (2003) 'Utilizing synthesis methods in accurate system-level exploration of heterogeneous embedded systems', *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, August, pp.310–315.

Erbas, C., Polstra, S. and Pimentel, A.D. (2003) 'IDF models for trace transformations: a case study in computational refinement', *Proc. Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, July, pp.178–187.

Gao, G.R., Govindarajan, R. and Panangaden, P. (1992) 'Well-behaved dataflow programs for DSP computation', *Proc. Int. Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, March, pp.561–564.

Grötker, T., Liao, S., Martin, G. and Swan, S. (2002) *System Design with SystemC,* Kluwer Academic Publishers, The Netherlands.

Hoare, C.A.R. (1978) 'Communicating sequential processes', *Communications of the ACM*, Vol. 21, No. 8, August.

Kahn, G. (1974) 'The semantics of a simple language for parallel programming', *Proc. IFIP Congress 74*.

Keutzer, K., Malik, S., Newton, A., Rabaey, J. and Sangiovanni-Vincentelli, A. (2000) 'System level design: orthog-onalization of concerns and platform-based design', *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 19, No. 12, December.

Kienhuis, B., Deprettere, E.F. Vissers, K.A. and van der Wolf, P. (1997) 'An approach for quantitative analysis of application-specific dataflow architectures', *Proc. Int. Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, July.

Kogel, T., Wieferink, A., Leupers, R., Ascheid, G., Meyr, H., Bussaglia, D. and Ariyamparambath, M. (2003) 'Virtual architecture mapping: a SystemC based methodology for architectural exploration of system-on-chip designs', *Proc. Int. workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, pp.138–148.

Kuzmanov, G., Gaydadjiev, G.N. and Vassiliadis, S. (2004) 'The virtex II Pro™ MOLEN processor', *Proc. Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, July, pp.192–202.

Kuzmanov, G.K., Gaydadjiev, G.N. and Vassiliadis, S. (2003) 'Loading p/x-code: design considerations', *Proc. Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, July, pp.11–19.

Kuzmanov, G.K. and Vassiliadis, S. (2003) 'Arbitrating instructions in an pμ-coded CCM', *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL)*, September, pp.81–90.

Lahiri, K., Raghunathan, A. and Dey, S. (2001) 'System-level performance analysis for designing on-chip communication architectures', *IEEE Trans, on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 6, June, pp.768–783.

Lee, E.A. and Messerschmitt, D.G. (1987) 'Synchronous dataflow', *Proceedings of the IEEE*, Vol. 75, No. 9, September, pp.1235–1245.

Lee, E.A. and Neuendorffer, S. (2000) *MoML – A Modeling Markup Language in XML, Version 0.4*, Technical Report UCB/ERL MOO/8, Electronics Research Lab, March, University of California, Berkeley.

Lee, E.A. and Parks, T.M. (1995) 'Dataflow process networks', *Proc. IEEE*, Vol. 83, No. 5, May, pp.773–801.

Lieverse, P., Stefanov, T., van der Wolf, P. and de Prettere, E.F. (2001a) 'System level design with spade: an M-JPEG case study', *Proc. Int. Conference on Computer Aided Design (ICCAD)*, November, pp.31–38.

Lieverse, P., van der Wolf, P. and Deprettere, E.F. (2001b) 'A trace transformation technique for communication refinement', *Proc. Int. Symposium, on Hardware/Software Codesign (CODES)*, April, pp.134–139.

Lieverse, P., van der Wolf, P., Deprettere, E.F. and Vissers, K.A. (2001c) 'A methodology for architecture exploration of heterogeneous signal processing systems', *Journal of VLSI Signal Processing for Signal, Image and Video Technology*, Vol. 29, No. 3, November, pp.197–207.

Mihal, A. and Keutzer, K. (2003) 'Mapping concurrent applications onto architectural platforms', in Jantsch, A. and Tenhunen, H. (Eds.): *Networks on Chips*, pp.39–59, Kluwer Academic Publishers.

Mihal, A., Kulkarni, C., Sauer, C., Vissers, K., Moskewicz, M., Tsai, M., Shah, N., Weber, S., Jin, Y., Keutzer, K. and Malik, S. (2002) 'Developing architectural platforms: a disciplined approach', *IEEE Design and Test of Computers*, Vol. 19, pp.6–16.

Mohanty, S. and Prasanna, V.K (2002) 'Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures', *Proc. IEEE International ASIC/SOC Conference.*

Muller, H.L. (1993) *Simulating Computer Architectures*, PhD thesis, Dept. of Computer Science, Univ. of Amsterdam, February.

Nicolescu, G., Yoo, S. and Jerraya, A.A. (2001) 'Mixed-level cosimulation for fine gradual refinement of communication in SoC design', *Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE)*, March.

Nieuwland, A. and Lippens, P. (1998) 'A heterogeneous HW-SW architecture for hand-held multi-media terminals', *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, October, pp.113–122.

Peng, J., Abdi, S. and Gajski, D. (2002) 'Automatic model refinement for fast architecture exploration', *Proc. Int.Conference on VLSI Design*, January, pp.332–337.

Pimentel, A.D. and Erbas, C. (2003) 'An IDF-based trace transformation method for communication refinement', *Proc. Design Automation Conference (DAC)*, June, pp.402–407.

Pimentel, A.D., Lieverse, P., van der Wolf, P., Hertzberger, L.O. and Deprettere, E.F. (2001) 'Exploring embedded systems architectures with Artemis', *IEEE Computer*, Vol. 34, No. 11, November, pp.57–63.

Pimentel, A.D., Polstra, S., Terpstra, F., van Halderen, A.W., Coffland, J.E. and Hertzberger, L.O. (2002) 'Towards efficient design space exploration of heterogeneous embedded media systems', in Deprettere, E.F., Teich, J. and Vassiliadis, S. (Eds.): *Embedded Processor Design Challenges*, Springer, LNCS 2268, pp.57–73.

Rowson, J.A. and Sangiovanni-Vincentelli, A. (1997) 'Interface-based design', *Proc. Design Automation Conference (DAC)*, June.

Sangiovanni-Vincentelli, A. and Martin, G. (2001) 'Platform-based design and software design methodology for embedded systems', *IEEE Design and Test of Computers*, Vol. 18, No. 6, pp.23–33.

Stefanov, T. and Deprettere, E.F. (2003) 'Deriving process networks from weakly dynamic applications in system-level design', *Proc. Int. Conference on HW/SW Code-sign and System Synthesis (CODES-ISSS)*, October.

Stefanov, T., Kienhuis, B. and Deprettere, E.F. (2002) 'Algorithmic transformation techniques for efficient exploration of alternative application instances', *Proc. Int. Symposium on Hardware/Software Codesign (CODES)*, May, pp.7–12.

Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B. and Deprettere, E.F. (2004) 'System design using Kahn process networks: the Compaan/Laura approach', *Proc. Int. Conference on Design, Automation and Test in Europe (DATE)*, February, pp.340–345.

Thompson, M. and Pimentel, A.D. (2004) 'A high-level programming paradigm for SystemC', *Proc. Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, July, pp.530–539.

Turjan, A., Kienhuis, B. and Deprettere, E.F. (2004) 'Translating affine nested loop programs to process networks', *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, September.

Vahid, F. and Givargis, T. (2001) 'Platform tuning for embedded systems design', *IEEE Computer*, Vol. 34, No. 3, March.

Vassiliadis, S., Wong, S. and Cotofana, S. (2001) 'The MOLEN micro-coded processor', *Proc. Int. Conference on Field-Programmable Logic and Applications (FPL)*, August, pp.275–285.

Vassiliadis, S., Gaydadjiev, G.N., Bertels, K. and Moscu Panainte, E. (2003a). 'The Molen programming paradigm', *Proc. Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, July, pp.1–10.

Vassiliadis, S., Wong, S. and Cotofana, S.D. (2003b) 'Microcode processing: positioning and directions', *IEEE Micro*, Vol. 23, No. 4, July, pp.21–30.

Živković, V., Deprettere, E.F., de Kock, E. and van der Wolf, P. (2003a) 'Mapping specification-level primitives to ip-primitives: a case study', *Proc. Int. Workshop on Systems, Architectures, Modelling, and Simulation (SAMOS)*, July.

Živković, V., Deprettere, E.F., van der Wolf, P. and de Kock, E. (2003b) 'Fast and accurate multiprocessor architecture exploration with symbolic programs', *Proc. Int. Conference on Design Automation and Test in Europe (DATE)*, March.

Živković, V., van der Wolf, P., Deprettere, E.F. and de Kock, E.A. (2002) 'Design space exploration of streaming multiprocessor architectures', *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, October.

Wolf, W. (2001) *Computers as Components: Principles of Embedded Computer Systems Design*, Morgan Kaufmann Publishers.

Zissulescu, C., Stefanov, T., Kienhuis, B. and Deprettere, E.F. (2003) 'LAURA: Leiden architecture research and exploration tool', *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL)*, September.

Zitzler, E. (1999) *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*, PhD thesis, Swiss Federal Institute of Technology Zurich.

Website

SystemC initiative, http://www.systemc.org/

Notes

[1]Here we should note that although a significant amount of work has been performed on Compaan, Laura and Molen in the context of the Artemis project, including the integration of these research efforts into a single framework, they do not have their origin in Artemis.

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

[3]In our M-JPEG application, we use 4:2:2 YUV macro-blocks.