The Artemis Workbench for System-level Performance Evaluation of Embedded Systems

Andy D. Pimentel, *Member, IEEE Computer Society*

***Abstract*— In this article, 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.**

***Index Terms*— System-level modelling and simulation, model refinement, performance evaluation, design space exploration, model calibration.**

I. 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 fully 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 (*SoC*s) 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.

1) *Platform architectures*
   Platform-based design [46], [52] has become a popular design method as it stresses the re-use 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 customising a chip for every application.
2) *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 [22]. Two common types of separation in the design process are: i) separating computation from communication by connecting IP processing cores via a standard (message-passing) network interface [5] and ii) separating application (what is the system supposed to do) from architecture (how it does it) [3], [23].
3) *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 [43]. 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., [3], [43]), power consumption (e.g., [6]) or cost [13] 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 article, we present an overview of the Artemis workbench, which provides high-level modelling and simulation methods

A.D. Pimentel is with the Dept. of Computer Science of the University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands. He is project coordinator of the Artemis project. Email: andy@science.uva.nl

and tools for efficient performance evaluation and exploration of heterogeneous embedded multimedia systems [43]. More specifically, we describe the Artemis system-level modelling methodology – which deploys the aforementioned principle of separation of concerns [22] – 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 article 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 III focuses on Artemis' system-level modelling and simulation techniques by describing its prototype modelling and simulation environment, called Sesame. In Section IV, we explain how Artemis facilitates gradual refinement of system-level architecture models by applying dataflow graphs. Section V 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 calibration approach. Finally, Section VI discusses related work, after which Section VII concludes the article.

II. 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 [14], [49], [51]. More specifically, Compaan transforms the sequential application specification into a Kahn Process Network (KPN) [21]. 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 [35]. In the next sections, we will elaborate on all of the above modelling, mapping, and refinement aspects in more detail.

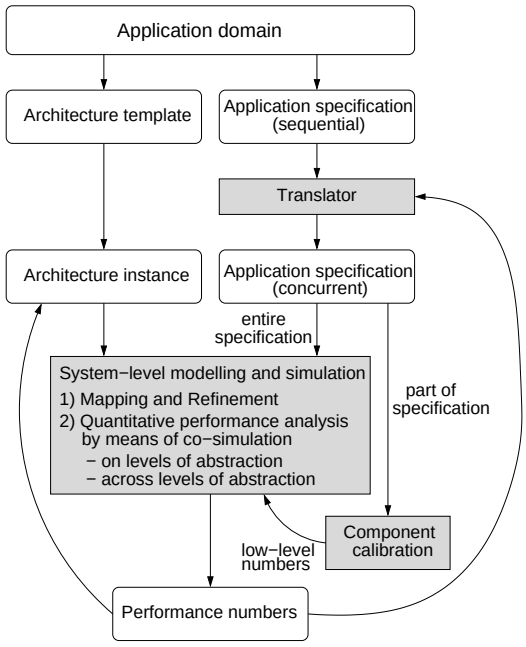


Fig. 1. The infrastructure of the Artemis workbench.

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 are 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 multi-processor system architecture as a whole. For this calibration process, the Artemis workbench uses the Laura tool-set [49], [60] and the Molen calibration platform architecture [53]–[55]. Before presenting more details on Artemis' system-level modelling and simulation techniques, which forms the bulk of this article, 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.

A. 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 multiprocessor SoC architectures that allow for exploiting task-level parallelism in applications. In contrast, models of computation

[1]Here we should note that although a signifi cant 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.

that inherently express task-level parallelism in applications and make communications explicit, such as CSP [20] and Process Networks [21], [31], 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 Matlab – which is then automatically converted into a Kahn Process Network (KPN) [21] using the Compaan tool-set [14], [49], [51]. 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 [47]. 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 [48]. 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 [49], [60], depicted on the right-hand side of Figure 2, takes a KPN as input and produces synthesizable 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.

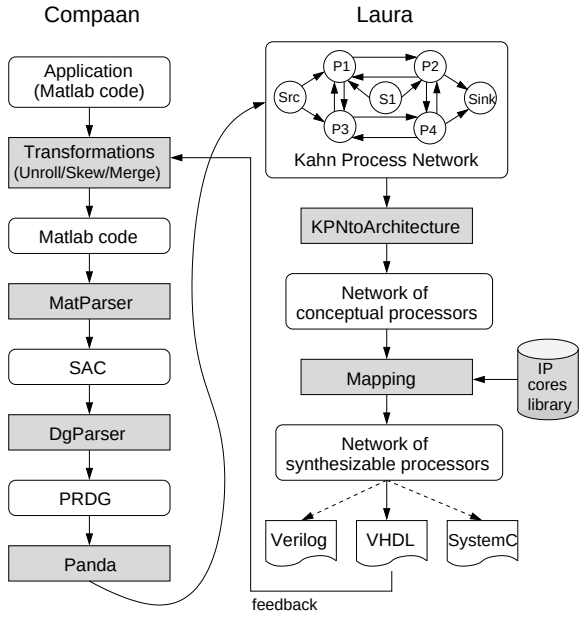


Fig. 2. The Compaan (left) and Laura (right) tool-sets.

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 synthesized 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.

B. The Molen calibration platform

Figure 3 depicts the platform architecture that is used for component calibration in Artemis. This platform architecture, called Molen [53]–[55], connects a programmable processor with a reconfigurable unit and uses microcode to incorporate architectural support for the reconfigurable unit. Instructions

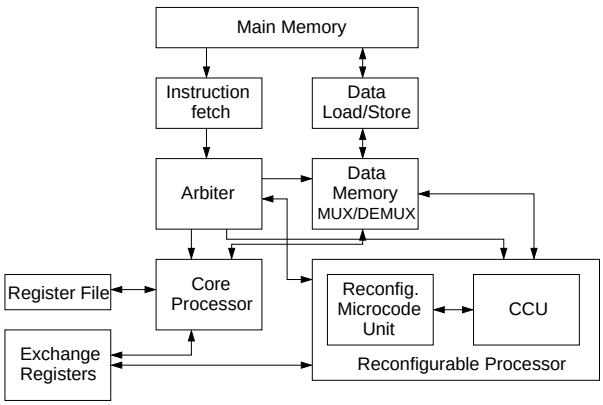


Fig. 3. The Molen calibration platform architecture.

are fetched from the memory, after which the arbiter performs a partial decoding on the instructions to determine where they should be issued [27]. 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^{TM} platform in the Molen prototype implementation [25], 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^{TM} FPGA, and a ρμ-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.

To perform the actual reconfiguration of the CCU, *reconfiguration microcode* is loaded into the ρμ-code unit and then executed (using *p-set* and *c-set* instructions) [26]. 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 synthesized 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 V, we present a case study in which this component calibration is illustrated for a DCT task in a Motion-JPEG encoder application.

III. The Sesame modelling and simulation environment

Artemis' system-level modelling and simulation environment, called Sesame [11], [44], builds upon the ground-laying work of the Spade framework [34]. This means that

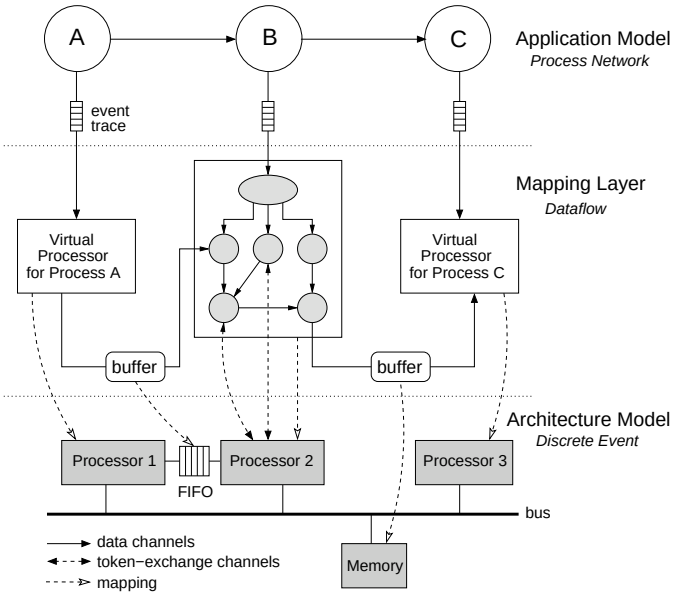


Fig. 4. Sesame's application model layer, architecture model layer, and mapping layer which interfaces between application and architecture models.

Sesame facilitates performance analysis of embedded systems architectures according to the Y-chart design approach [3], [23], 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.

A. Application modelling

For application modelling [12], Sesame uses KPN application specifications that are generated by the Compaan tool-set 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(pixel-block,channel_id)*.

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) [11]. This is an XML-based language which is similar to Ptolemy's MoML [30] 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 component 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.

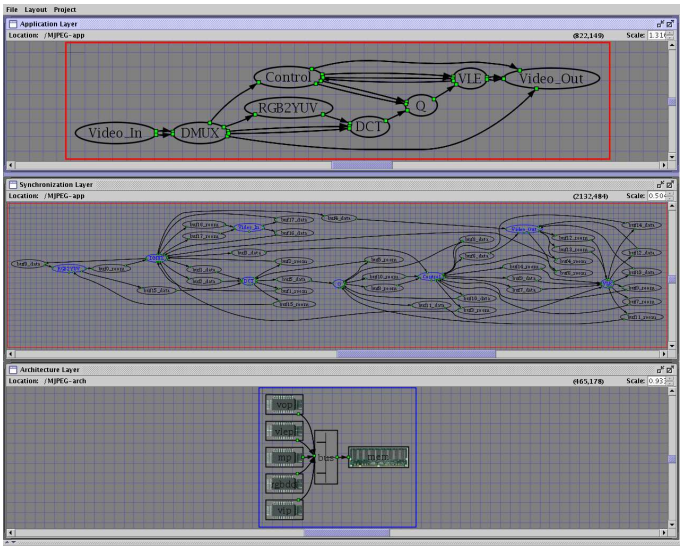


Fig. 5. A screenshot of the YML editor, showing its layered layout that corresponds to the three layers of Sesame (see Figure 4).

B. Architecture modelling

Architecture models in Sesame, which typically operate at the so-called transaction level [10], [19], 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 [38] or SystemC [1], [19]. Pearl is a small but powerful discrete-event simulation language which provides easy construction of the models and fast simulation [44]. For our SystemC architecture models, we provide an add-on library to SystemC, called SCPEx (SystemC Pearl Extension) [50], 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.

C. 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 also 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 with which application events are dispatched 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 events that originate from different event queues according to a given policy and subsequently models their timing consequences. The event scheduling supports a range of predefined policies, like FCFS and round-robin, but can also easily be customised with alternative policies. For communication events, 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 [58], an alternative mapping approach is studied. That is, while Archer and Sesame share a lot of the same application and architecture modelling techniques, Archer uses a different mapping strategy than Sesame. In Archer, Control Data Flow Graphs (CDFG) [59] 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) [57]. 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 [56].

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

D. Mapping support using multi-objective optimisation

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 [15]. 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 [61], 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 [15].

IV. Architecture model refinement through trace transformations

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 only maintains application models at a high level of abstraction (thereby optimising the potentials for re-use 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 [16], [17], [42]. In other words, the dataflow actors 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* [33], 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 [33]:

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

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

Here, refined architecture-level events *check-data**, *load-data*†, *signal-room**, *check-room**, *store-data*†, *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 [33], [42]. 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:

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

In the refined event sequence, we 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) [29] actors are deployed to realise trace transformations. Integer-controlled Data Flow (IDF) [8] actors are subsequently utilised to model repetitions and branching conditions which may be present in the application code [16]. However, as illustrated in [42], 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 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 [58]. 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. 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 [9] and has been proven to be very useful for system simulation [8]. 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 [18]. 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 [32] and [44]. Figure 6 shows the annotated C++ code for the *Quality-Control* (QC)

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

Kahn process of the M-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

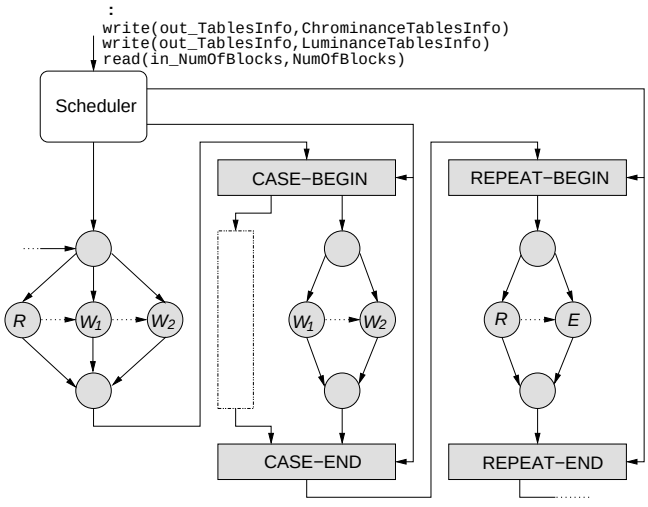


Fig. 7. IDF graph representing the QC task from Figure 6, realising high-level (unrefined) simulation at architecture level.

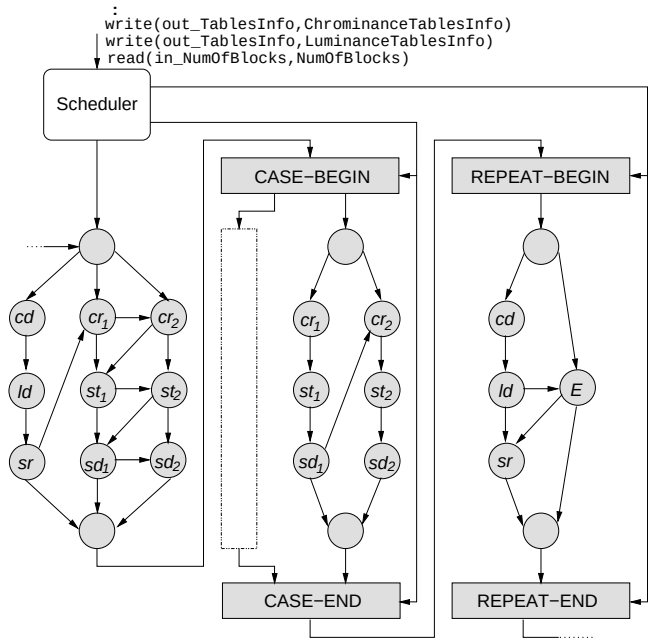


Fig. 8. IDF graph for the QC task realising communication refinement.

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 (3), noting that the right-hand side may also be specified as a partial ordering [17], [33] – this automatically yields a *valid* schedule for the IDF graph [16]. 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 [17] and like we will demonstrate in the next section, this approach also allows for refining computational behaviour.

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 performance evaluation of a specific architecture component within 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.

V. 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 encoder application from the previous section, we demonstrate how component 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 for 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, that can be mapped onto the FPGA (i.e., CCU) of the Molen platform. By mapping the remainder of the M-JPEG code onto Molen's core processor (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 article, 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.

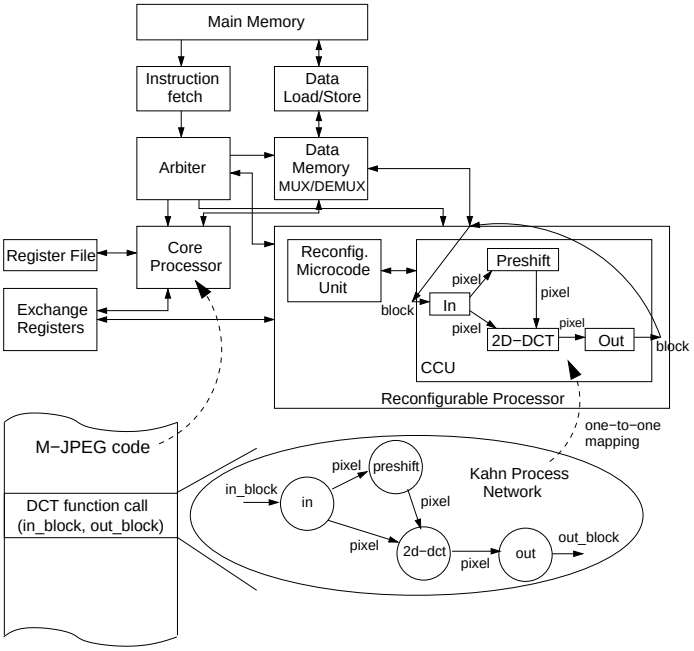


Fig. 9. Calibration of the DCT task using the Compaan/Laura frameworks and the Molen calibration platform.

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.

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

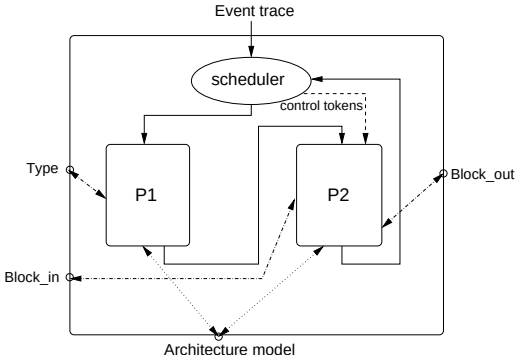


Fig. 10. Virtual processor for DCT Kahn process.

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

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

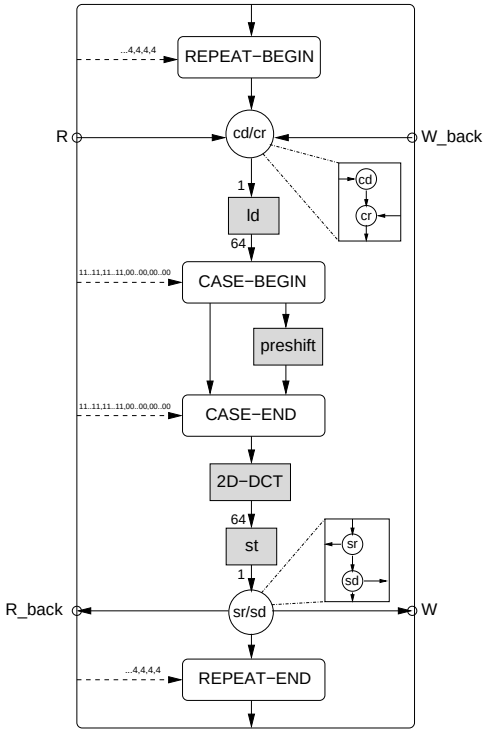


Fig. 11. IDF graph for the composite actor P2 from Figure 10.

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*).

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

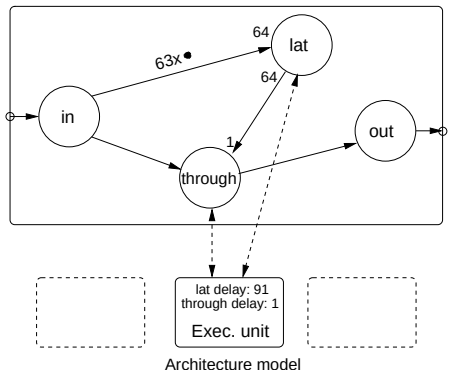


Fig. 12. SDF graph for the 2D-DCT composite actor in Figure 11, modelling an abstract pipeline.

TABLE I
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

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 [17]. This is relatively easy using YML, which allows us to describe models in a repetitive manner using loop-like constructs.

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 I shows the validation results for a sequence of sample input frames.

The results from Table I 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.

VI. RELATED WORK

There are a number of architectural exploration environments, such as (Metro)Polis [3], [4], Mescal [36], Milan [37], and various SystemC-based environments like the work of [24], 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 article, 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 [28] 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 [57], [58], 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 [34]. 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 [9] 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 [20] models, and many types of dataflow models [31]: Synchronous Dataflow, Boolean Dataflow, Integer-controlled Dataflow, Dynamic Dataflow, as well as (Kahn) Process Networks [21].

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., [37]). Although we have not addressed such traditional model calibration in this article, 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 a hardware 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 [4] and Milan frameworks [37], the work of [41], and in the context of SystemC (e.g., [24]). In [41], 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 [2], [7], [33], [39], [40], [45], various mechanisms are proposed for the refinement of application level communication primitives into more detailed implementation (architecture) primitives.

VII. Conclusions

In this article, 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 article 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 article. 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 Coffland, 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 Živković 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 article.

References

[1] "SystemC initiative," http://www.systemc.org/.

[2] S. Abdi, D. Shin, and D. Gajski, "Automatic communication refinement for system level design," in *Proc. of the Design Automation Conference (DAC)*, June 2003, pp. 300–305.

[3] 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 Publishers, 1997.

[4] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, April 2003.

[5] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–80, Jan. 2002.

[6] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. of the Int. Symposium on Computer Architecture (ISCA)*, June 2000.

[7] J. Y. Brunel, E. A. de Kock, W. Kruijtzer, H. Kenter, and W. Smits, "Communication refinement in video systems on chip," in *Proc. of the Int. Workshop on Hardware/Software Codesign (CODES)*, May 1999, pp. 142–146.

[8] J. T. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer valued control streams," in *Proc. of the Asilomar conference on Signals, Systems, and Computers*, Oct. 1994.

[9] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, vol. 4, pp. 155–182, Apr. 1994.

[10] L. Cai and D. Gajski, "Transaction level modeling: An overview," in *Proc. of the Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, Oct. 2003, pp. 19–24.

[11] J. E. Coffland and A. D. Pimentel, "A software framework for efficient system-level performance evaluation of embedded systems," in *Proc. of the ACM Symp. on Applied Computing (SAC)*, March 2003, pp. 666–671.

[12] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J. Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers, "Yapi: Application modeling for signal processing systems," in *Proc. of the Design Automation Conference (DAC)*, June 2000, pp. 402–405.

[13] J. DeBardelaben, V. Madisetti, and A. Gadient, "Incorporating cost modeling into embedded-system design," *IEEE Design & Test of Computers*, vol. 14, no. 3, Sept. 1997.

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

[15] C. Erbas, S. C. Erbas, and A. D. Pimentel, "A multiobjective optimization model for exploring multiprocessor mappings of process networks," in *Proc. of the Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, Oct. 2003, pp. 182–187.

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

[17] C. Erbas, S. Polstra, and A. D. Pimentel, "IDF models for trace transformations: A case study in computational refinement," in *Proc. of the Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS)*, July 2003, pp. 178–187.

[18] G. Gao, R. Govindarajan, and P. Panangaden, "Well-behaved dataflow programs for DSP computation," in *Proc. of the Int. Conference on Acoustics, Speech, and Signal Processin (ICASSP)*, March 1992, pp. 561–564.

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

[20] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, August 1978.

[21] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress 74*, 1974.

[22] K. Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonolization of concerns and platform-based design," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, Dec. 2000.

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

[24] 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," in *Proc. of the Int. workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, 2003, pp. 138–148.

[25] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, "The Virtex II Pro™ MOLEN processor," in *Proc. of the Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS)*, July 2004.

[26] G. K. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, "Loading ρμ-code: Design considerations," in *Proc. of the Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS)*, July 2003.

[27] G. K. Kuzmanov and S. Vassiliadis, "Arbitrating instructions in an ρμ-coded CCM," in *Proc. of the 13th Int. Conference on Field Programmable Logic and Applications (FPL)*, Sept. 2003, pp. 81–90.

[28] K. Lahiri, A. Raghunathan, and S. Dey, "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, pp. 768–783, June 2001.

[29] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept. 1987.

[30] E. A. Lee and S. Neuendorffer, "MoML - a Modeling Markup Language in XML, version 0.4," Electronics Research Lab, University of California, Berkeley, Tech. Rep. UCB/ERL M00/8, March 2000.

[31] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[32] P. Lieverse, T. Stefanov, P. van der Wolf, and E. F. Deprettere, "System level design with Spade: an M-JPEG case study," in *Proc. of the Int. Conference on Computer Aided Design (ICCAD)*, Nov. 2001, pp. 31–38.

[33] P. Lieverse, P. van der Wolf, and E. F. Deprettere, "A trace transformation technique for communication refinement," in *Proc. of the Int. Symposium on Hardware/Software Codesign (CODES)*, Apr. 2001, pp. 134–139.

[34] P. Lieverse, P. van der Wolf, E. F. Deprettere, and K. A. Vissers, "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, pp. 197–207, Nov. 2001.

[35] A. Mihal and K. Keutzer, "Mapping concurrent applications onto architectural platforms," in *Networks on Chips*, A. Jantsch and H. Tenhunen, Eds. Kluwer Academic Publishers, 2003, pp. 39–59.

[36] 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, pp. 6–16, 2002.

[37] S. Mohanty and V. K. Prasanna, "Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures," in *Proc. of the IEEE International ASIC/SOC Conference*, 2002.

[38] H. L. Muller, "Simulating computer architectures," Ph.D. dissertation, Dept. of Computer Science, Univ. of Amsterdam, Feb. 1993.

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

[40] A. Nieuwland and P. Lippens, "A heterogenous HW-SW architecture for hand-held multi-media terminals," in *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 1998, pp. 113–122.

[41] J. Peng, S. Abdi, and D. Gajski, "Automatic model refinement for fast architecture exploration," in *Proc. of the Int. Conference on VLSI Design*, Jan. 2002, pp. 332–337.

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

[43] A. D. Pimentel, P. Lieverse, P. van der Wolf, L. O. Hertzberger, and E. F. Deprettere, "Exploring embedded-systems architectures with Artemis," *IEEE Computer*, vol. 34, no. 11, pp. 57–63, Nov. 2001.

[44] 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," in *Embedded Processor Design Challenges*, E. Deprettere, J. Teich, and S. Vassiliadis, Eds. Springer, LNCS 2268, 2002, pp. 57–73.

[45] J. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design," in *Proc. of the Design Automation Conference (DAC)*, June 1997.

[46] 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.

[47] T. Stefanov and E. F. Deprettere, "Deriving process networks from weakly dynamic applications in system-level design," in *Proc. of the Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, Oct. 2003.

[48] T. Stefanov, B. Kienhuis, and E. F. Deprettere, "Algorithmic transformation techniques for efficient exploration of alternative application instances," in *Proc. of the Int. Symposium on Hardware/Software Codesign (CODES)*, May 2002, pp. 7–12.

[49] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. F. Deprettere, "System design using Kahn process networks: The Compaan/Laura approach," in *Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE)*, Feb. 2004, pp. 340–345.

[50] M. Thompson and A. D. Pimentel, "A high-level programming paradigm for SystemC," in *Proc. of the Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS)*, July 2004, pp. 530–539.

[51] A. Turjan, B. Kienhuis, and E. F. Deprettere, "Translating affine nested loop programs to process networks," in *Proc. of the Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Sept. 2004.

[52] F. Vahid and T. Givargis, "Platform tuning for embedded systems design," *IEEE Computer*, vol. 34, no. 3, March 2001.

[53] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. M. Panainte, "The Molen programming paradigm," in *Proc. of the Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS)*, July 2003.

[54] S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN micro-coded processor," in *Proc. of the Int. Conference on Field-Programmable Logic and Applications (FPL)*, August 2001, pp. 275–285.

[55] S. Vassiliadis, S. Wong, and S. D. Cotofana, "Microcode processing: Positioning and directions," *IEEE Micro*, vol. 23, no. 4, pp. 21–30, July 2003.

[56] V. Živković, E. F. Deprettere, E. de Kock, and P. van der Wolf, "Mapping specification-level primitives to ip-primitives: A case study," in *Proc. of the Int. Workshop on Systems, Architectures, MOdelling, and Simulation (SAMOS)*, July 2003.

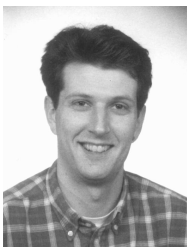
[57] V. Živković, E. F. Deprettere, P. van der Wolf, and E. de Kock, "Fast and accurate multiprocessor architecture exploration with symbolic programs," in *Proc. of the Int. Conference on Design Automation and Test in Europe (DATE)*, March 2003.

[58] V. Živković, P. van der Wolf, E. F. Deprettere, and E. A. de Kock, "Design space exploration of streaming multiprocessor architectures." in *Proc. of the IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2002.

[59] W. Wolf, *Computers as Components: Principles of Embedded Computer Systems Design*. Morgan Kaufmann Publishers, 2001.

[60] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. F. Deprettere, "LAURA: Leiden architecture research and exploration tool," in *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL)*, Sept. 2003.

[61] E. Zitzler, "Evolutionary algorithms for multiobjective optimization: Methods and applications," Ph.D. dissertation, Swiss Federal Institute of Technology Zurich, 1999.



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 Department of Computer Science. He is co-founder of the International Workshop on Systems, Architectures, Modelling, and Simulation (SAMOS). His research interests include computer architecture, embedded systems, computer architecture modelling and simulation, system-level performance analysis, and parallel computing. He is a member of the IEEE Computer Society.