3

Tools and Methodologies for System-Level Design

Shuvra Bhattacharyya and Marilyn Wolf

CONTENTS

3.1 Introduction 40
3.2 Characteristics of Video Applications 40
3.3 Platform Characteristics 41
  3.3.1 Custom System-on-Chip Architectures 42
  3.3.2 Graphics Processing Units 42
  3.3.3 Platform FPGAs 43
3.4 Abstract Design Methodologies 43
3.5 Model-Based Design Methodologies 44
  3.5.1 Dataflow Models 44
  3.5.2 Dataflow Modeling for Video Processing 46
  3.5.3 Multidimensional Dataflow Models 46
  3.5.4 Control Flow 46
  3.5.5 Integration with Finite-State Machine and Mode-Based Modeling Methods 47
  3.5.6 Video Processing Examples 47
3.6 Languages and Tools for Model-Based Design 49
  3.6.1 CAL 49
  3.6.2 Compaan 50
  3.6.3 PREESM 50
  3.6.4 Ptolemy 51
  3.6.5 SysteMoc 52
3.1 INTRODUCTION

System-level design, once the province of board designers, has now become a central concern for chip designers. Because chip design is a less forgiving design medium—design cycles are longer and mistakes are harder to correct—system-on-chip (SoC) designers need a more extensive tool suite than may be used by board designers, and a variety of tools and methodologies have been developed for system-level design of SoCs.

System-level design is less amenable to synthesis than are logic or physical design. As a result, system-level tools concentrate on modeling, simulation, design space exploration, and design verification. The goal of modeling is to correctly capture the system's operational semantics, which helps with both implementation and verification. The study of models of computation provides a framework for the description of digital systems. Not only do we need to understand a particular style of computation, such as dataflow, but we also need to understand how different models of communication can reliably communicate with each other. Design space exploration tools, such as hardware/software codesign, develop candidate designs to understand trade-offs. Simulation can be used not only to verify functional correctness but also to supply performance and power/energy information for design analysis.

We will use video applications as examples in this chapter. Video is a leading-edge application that illustrates many important aspects of system-level design. Although some of this information is clearly specific to video, many of the lessons translate to other domains.

The next two sections briefly introduce video applications and some SoC architectures that may be the targets of system-level design tools. We will then study models of computation and languages for system-level modeling. We will then survey simulation techniques. We will close with a discussion of hardware/software codesign.

3.2 CHARACTERISTICS OF VIDEO APPLICATIONS

One of the primary uses of SoCs for multimedia today is for video encoding—both compression and decompression. In this section, we review the basic characteristics of video compression algorithms and the implications for video SoC design.

Video compression standards enable video devices to interoperate. The two major lines of video compression standards are MPEG and H.26x. The MPEG standards concentrate on broadcast applications, which allow for a more expensive compressor on the transmitter side in exchange for a simpler receiver. The H.26x standards were developed with videoconferencing in mind, in which both sides must encode and decode. The Advanced Video Codec standard, also known as H.264, was formed by the confluence of the H.26x and MPEG efforts. H.264 is widely used in consumer video systems.

Modern video compression systems combine lossy and lossless encoding methods to reduce the size of a video stream. Lossy methods throw away information such that the uncompressed video stream is not a perfect reconstruction of the original; lossless methods do allow the information provided to them to be perfectly reconstructed. Most modern standards use three major mechanisms:

1. The discrete cosine transform (DCT) together with quantization
2. Motion estimation and compensation
3. Huffman-style encoding
Quantized DCT and motion estimation are lossy, while Huffman encoding is lossless. These three methods leverage different aspects of the video stream's characteristics to more efficiently encode it.

The combination of DCT and quantization was originally developed for still images and is used in video to compress single frames. The DCT is a frequency transform that turns a set of pixels into a set of coefficients for the spatial frequencies that form the components of the image represented by the pixels. The DCT is used over other transforms because a 2D DCT can be computed using two 1D DCTs, making it more efficient. In most standards, the DCT is performed on an 8 × 8 block of pixels. The DCT does not itself lossily compress the image; rather, the quantization phase can more easily pick out information to throw away thanks to the structure of the DCT. Quantization throws out fine detail in the block of pixels, which correspond to the high-frequency coefficients in the DCT. The number of coefficients set to zero is determined on the level of compression desired.

Motion estimation and compensation exploit the relationships between frames provided by moving objects. A reference frame is used to encode later frames through a motion vector, which describes the motion of a macroblock of pixels (16 × 16 in many standards). The block is copied from the reference frame into the new position described by the motion vector. The motion vector is much smaller than the block it represents. Two-dimensional correlation is used to determine the position of the macroblock's position in the new frame; several positions in a search area are tested using 2D correlation. An error signal encodes the difference between the predicted and the actual frames; the receiver uses that signal to improve the predicted picture.

MPEG distinguishes several types of frames: I (inter) frames are not motion compensated, P (predicted) frames have been predicted from earlier frames, and B (bidirectional) frames have been predicted from both earlier and later frames.

The results of these lossy compression phases are assembled into a bit stream and compressed using lossless compression such as Huffman encoding. This process reduces the size of the representation without further compromising image quality.

It should be clear that video compression systems are actually heterogeneous collections of algorithms; this is also true of many other applications, including communications and security. A video computing platform must run several algorithms; those algorithms may perform very different types of operations, imposing very different requirements on the architecture.

This has two implications for tools: first, we need a wide variety of tools to support the design of these applications; second, the various models of computation and algorithmic styles used in different parts of an application must at some point be made to communicate to create the complete system. For example, DCT can be formulated as a dataflow problem, thanks to its butterfly computational structure, while Huffman encoding is often described in a control-oriented style.

Several studies of multimedia performance on programmable processors have remarked on the significant number of branches in multimedia code. These observations contradict the popular notion of video as regular operations on streaming data. Fritts and Wolf [1] measured the characteristics of the MediaBench benchmarks.

They used path ratio to measure the percentage of instructions in a loop body that were actually executed. They found that the average path ratio of the MediaBench suite was 78%, which indicates that a significant number of loops exercise data-dependent behavior. Talla et al. [2] found that most of the available parallelism in multimedia benchmarks came from interiteration parallelism. Exploiting the complex parallelism found in modern multimedia algorithms requires that synthesis algorithms be able to handle more complex computations than simple ideal nested loops.

### 3.3 PLATFORM CHARACTERISTICS

Many SoCs are heterogeneous multiprocessors and the architectures designed for multimedia applications are no exception. In this section, we review several SoCs, including some general-purpose SoC architectures as well as several designed specifically for multimedia applications.
Two very different types of hardware platforms have emerged for large-scale applications. On the one hand, many custom SoCs have been designed for various applications. These custom SoCs are customized by loading software onto them for execution. On the other hand, platform field-programmable gate arrays (FPGAs) provide FPGA fabrics along with CPUs and other components; the design can be customized by programming the FPGA as well as the processor(s). These two styles of architecture represent different approaches for SoC architecture and they require very different sorts of tools: custom SoCs require large-scale software support, while platform FPGAs are well suited to hardware/software codesign.

### 3.3.1 CUSTOM SYSTEM-ON-CHIP ARCHITECTURES

The TI OMAP family of processors [3] is designed for audio, industrial automation, and portable medical equipment. All members of the family include a C674x DSP; some members also include an ARM9 CPU.

The Freescale MPC574xP [4] includes 2 e200z4 CPUs operating in delayed lock step for safety checking as well as an embedded floating point unit.

### 3.3.2 GRAPHICS PROCESSING UNITS

GPUs are widely used in desktop and laptop systems as well as smartphones. GPUs are optimized for graphics rendering but have been applied to many other algorithms as well. GPUs provide SIMD-oriented architectures with floating-point support.

Figure 3.1 shows the organization of the NVIDIA Fermi [5]. Three types of processing elements are provided: cores, each of which has a floating point and an integer unit, load/store units, and special function units. A hierarchy of register files, caches, and shared memory provides very high memory bandwidth. A pair of warp processors controls operation. Each warp scheduler can control a set of 32 parallel threads.

![Organization of the Fermi graphics processing unit.](image)

© 2016 by Taylor & Francis Group, LLC
3.3.3 PLATFORM FPGAs

FPGAs [6] have been used for many years to implement logic designs. The FPGA provides a more general structure than programmable logic devices, allowing denser designs. They are less energy efficient than custom ASICs but do not require the long application specific integrated circuit (ASIC) design cycle.

Many FPGA design environments provide small, customizable soft processors that can be embedded as part of the logic in an FPGA. Examples include the Xilinx MicroBlaze and Altera Nios II. Nios II supports memory management and protection units, separate instruction and data caches, pipelined execution with dynamic branch prediction (up to 256 custom instructions), and hardware accelerators. MicroBlaze supports memory management units, instruction and data caches, pipelined operation, floating-point operations, and instruction set extensions.

The term “programmable SoC” refers to an FPGA that provides one or more hard logic CPUs in addition to a programmable FPGA fabric. Platform FPGAs provide a very different sort of heterogeneous platform than custom SoCs. The FPGA fabric allows the system designer to implement new hardware functions. While they generally do not allow the CPU itself to be modified, the FPGA logic is a closely coupled device with high throughput to the CPU and to memory. The CPU can also be programmed using standard tools to provide functions that are not well suited to FPGA implementation. For example, the Xilinx Zynq UltraScale+ family of multiprocessor systems-on-chips [7] includes an FPGA logic array, a quad-core ARM MPCore, dual-core ARM Cortex-R5, graphics processing unit, and DRAM interface.

3.4 ABSTRACT DESIGN METHODOLOGIES

Several groups have developed abstract models for system-level design methodologies. These models help us to compare concrete design methodologies.

An early influential model for design methodologies was the Gajski–Kuhn Y-chart [8]. As shown in Figure 3.2, the model covers three types of refinement (structural, behavioral, and physical) at four levels of design abstraction (transistor, gate, register transfer, and system). A design methodology could be viewed as a trajectory through the Y-chart as various refinements are performed at different levels of abstraction.

FIGURE 3.2 The Y-chart model for design methodologies. (From Gajski, D.D. and Kuhn, R.H., Computer, 16(12), 11, 1983.)
The X-chart model [9] has been proposed as a model for SoC design methodologies. As shown in Figure 3.3, a system specification is given by the combination of a behavioral description that describes the system function and a set of constraints that describes the nonfunctional requirements on the design. A synthesis procedure generates a structural implementation and a set of quality numbers by which the structure can be judged.

### 3.5 MODEL-BASED DESIGN METHODOLOGIES

Increasingly, developers of hardware and software for embedded computer systems are viewing aspects of the design and implementation processes in terms of domain-specific models of computation. Models of computation provide formal principles that govern how functional components in a computational specification operate and interact (e.g., see Reference 10). A domain-specific model of computation is designed to represent applications in a particular functional domain such as DSP and image and video processing; control system design; communication protocols or more general classes of discrete, control flow intensive decision-making processes; graphics; and device drivers. For discussions of some representative languages and tools that are specialized for these application domains, see, for example, [11–22]. For an integrated review of domain-specific programming languages for embedded systems, see Reference 23.

Processors expose a low-level Turing model at the instruction set. Traditional high-level programming languages like C, C++, and Java maintain the essential elements of that Turing model, including imperative semantics and memory-oriented operation. Mapping the semantics of modern, complex applications onto these low-level models is both time consuming and error prone. As a result, new programming languages and their associated design methodologies have been developed to support applications such as signal/image processing and communications. Compilers for these languages provide correct-by-construct translation of application-level operations to the Turing model, which both improves designer productivity and provides a stronger, more tool-oriented verification path [24].

#### 3.5.1 DATAFLOW MODELS

For most DSP applications, a significant part of the computational structure is well suited to modeling in a dataflow model of computation. In the context of programming models, dataflow refers to a modeling methodology where computations are represented as directed graphs in which vertices (actors) represent functional components and edges between actors represent
first-in-first-out (FIFO) channels that buffer data values (tokens) as they pass from an output of one actor to an input of another. Dataflow actors can represent computations of arbitrary complexity; typically in DSP design environments, they are specified using conventional languages such as C or assembly language, and their associated tasks range from simple, fine-grained functions such as addition and multiplication to coarse-grain DSP kernels or subsystems such as FFT units and adaptive filters.

The development of application modeling and analysis techniques based on dataflow graphs was inspired significantly by the computation graphs of Karp and Miller [25] and the process networks of Kahn [26]. A unified formulation of dataflow modeling principles as they apply to DSP design environment is provided by the dataflow process networks model of computation of Lee and Parks [27].

A dataflow actor is enabled for execution any time it has sufficient data on its incoming edges (i.e., in the associated FIFO channels) to perform its specified computation. An actor can execute at any time when it is enabled (data-driven execution). In general, the execution of an actor results in some number of tokens being removed (consumed) from each incoming edge and some number being placed (produced) on each outgoing edge. This production activity in general leads to the enabling of other actors.

The order in which actors execute, called the “schedule,” is not part of a dataflow specification and is constrained only by the simple principle of data-driven execution defined earlier. This is in contrast to many alternative computational models, such as those that underlie procedural languages, in which execution order is overspecified by the programmer [28]. The schedule for a dataflow specification may be determined at compile time (if sufficient static information is available), at run time, or when using a mixture of compile-time and run-time techniques. A particularly powerful class of scheduling techniques, referred to as “quasi-static scheduling” (e.g., see Reference 29), involves most, but not all, of the scheduling decisions being made at compile time.

Figure 3.4 shows an illustration of a video processing subsystem that is modeled using dataflow semantics. This is a design, developed using the Ptolemy II tool for model-based embedded system design [30], of an MPEG-2 subsystem for encoding the P frames that are processed.
by an enclosing MPEG-2 encoder system. A thorough discussion of this MPEG-2 system and its comparison to a variety of other modeling representations is presented in Reference 31. The components in the design of Figure 3.4 include actors for the DCT, zigzag scanning, quantization, motion compensation, and run length coding. The arrows in the illustration correspond to the edges in the underlying dataflow graph.

The actors and their interactions all conform to the semantics of synchronous dataflow (SDF), which is a restricted form of dataflow that is efficient for describing a broad class of DSP applications and has particularly strong formal properties and optimization advantages [32,33]. Specifically, SDF imposes the restriction that the number of tokens produced and consumed by each actor on each incident edge is constant. Many commercial DSP design tools have been developed that employ semantics that are equivalent to or closely related to SDF. Examples of such tools include Agilent’s SystemVue, Kalray’s MPPA Software Development Kit, National Instrument’s LabVIEW, and Synopsys’s SPW. Simulink®, another widely used commercial tool, also exhibits some relationships to the SDF model (e.g., see Reference 34).

3.5.2 DATAFLOW MODELING FOR VIDEO PROCESSING

In the context of video processing, SDF permits accurate representation of many useful subsystems, such as the P-frame encoder shown in Figure 3.4. However, such modeling is often restricted to a highly coarse level of granularity, where actors process individual frames or groups of successive frames on each execution. Modeling at such a coarse granularity can provide compact, top-level design representations, but greatly limits the benefits offered by the dataflow representation since most of the computation is subsumed by the general-purpose, intra-actor program representation. For example, the degree of parallel processing and memory management optimizations exposed to a dataflow-based synthesis tool becomes highly limited at such coarse levels of actor granularity. An example of a coarse-grain dataflow actor that “hides” significant amounts of parallelism is the DCT actor as shown in Figure 3.4.

3.5.3 MULTIDIMENSIONAL DATAFLOW MODELS

A number of alternative dataflow modeling methods have been introduced to address this limitation of SDF modeling for video processing and, more generally, multidimensional signal processing applications. For example, the multidimensional SDF (MD-SDF) model extends SDF semantics to allow constant-sized, n-dimensional vectors of data to be transferred across graph edges and provides support for arbitrary sampling lattices and lattice-changing operations [35]. Intuitively, a sampling lattice can be viewed as a generalization to multiple dimensions of a uniformly spaced configuration of sampling points for a 1D signal [36]; hexagonal and rectangular lattices are examples of commonly used sampling lattices for 2D signals. The computer vision (CV) SDF model is designed specifically for CV applications and provides a notion of structured buffers for decomposing video frames along graph edges; accessing neighborhoods of image data from within actors, in addition to the conventional production and consumption semantics of dataflow; and allowing actors to efficiently access previous frames of image data [37,38]. Blocked dataflow is a metamodeling technique for efficiently incorporating hierarchical, block-based processing of multidimensional data into a variety of dataflow modeling styles, including SDF and MD-SDF [31]. Windowed SDF is a model of computation that deeply integrates support for sliding window algorithms into the framework of static dataflow modeling [39]. Such support is important in the processing of images and video streams, where sliding window operations play a fundamental role.

3.5.4 CONTROL FLOW

As described previously, modern video processing applications are characterized by some degree of control flow processing for carrying out data-dependent configuration of application tasks and changes across multiple application modes. For example, in MPEG-2 video encoding,
significantly different processing is required for I frames, P frames, and B frames. Although the processing for each particular type of frame (I, P, or B) conforms to the SDF model, as illustrated for P frame processing in Figure 3.4, a layer of control flow processing is needed to efficiently integrate these three types of processing methods into a complete MPEG-2 encoder design. The SDF model is not well suited for performing this type of control flow processing and more generally for any functionality that requires dynamic communication patterns or activation/deactivation across actors.

A variety of alternative models of computation have been developed to address this limitation and integrate flexible control flow capability with the advantages of dataflow modeling. In Buck’s Boolean dataflow model [40] and the subsequent generalization as integer-controlled dataflow [41], provisions for such flexible processing were incorporated without departing from the framework of dataflow, and in a manner that facilitates the construction of efficient quasi-static schedules. In Boolean dataflow, the number of tokens produced or consumed on an edge is either fixed or is a two-valued function of a control token present on a control terminal of the same actor. It is possible to extend important SDF analysis techniques to Boolean dataflow graphs by employing symbolic variables. In particular, in constructing a schedule for Boolean dataflow actors, Buck’s techniques attempt to derive a quasi-static schedule, where each conditional actor execution is annotated with the run-time condition under which the execution should occur. Boolean dataflow is a powerful modeling technique that can express arbitrary control flow structures; however, as a result, key formal verification properties of SDF, such as bounded memory and deadlock detection, are lost (the associated analysis problems are not decidable) in the context of general Boolean dataflow specifications.

### 3.5.5 INTEGRATION WITH FINITE-STATE MACHINE AND MODE-BASED MODELING METHODS

In recent years, several modeling techniques have also been proposed that enhance expressive power by providing precise semantics for integrating dataflow or dataflow-like representations with finite-state machine (FSM) models and related methods for specifying and transitioning between different modes of actor behavior. These include El Greco [42], which evolved into the Synopsys System Studio and provides facilities for control models to dynamically configure specification parameters, *charts (pronounced "starcharts") with heterochronous dataflow as the concurrency model [43], the FunState intermediate representation [44], the DF* framework developed at K. U. Leuven [45], the control flow provisions in bounded dynamic dataflow [46], enable-invocation dataflow [47], and scenario-aware dataflow (SADF) [48].

### 3.5.6 VIDEO PROCESSING EXAMPLES

Figure 3.5 shows an illustration of a model of a complete MPEG-2 video encoder system that is constructed using Ptolemy, builds on the P-frame-processing subsystem of Figure 3.4, and employs multiple dataflow graphs nested within a FSM representation. Details on this application model can be found in Reference 31.

Figure 3.6 shows a block diagram, adapted from Reference 48, of an MPEG-4 decoder that is specified in terms of SADF. Descriptive names for the actors in this example are listed in Table 3.1, along with their SADF component types, which are either shown as “K” for kernel or “D” for detector. Intuitively, kernels correspond to data processing components of the enclosing dataflow graph, while detectors are used for control among different modes of operation (scenarios) for the kernels. The FD actor in the example of Figure 3.6 determines the frame type (I or P frame) and is designed as a detector. The other actors in the specification are kernels. For more details on this example and the underlying SADF model of computation, we refer the reader to Reference 48. The “D” symbols that appear next to some of the edges in Figure 3.6 represent dataflow delays, which correspond to initial tokens on the edges.
3.5 Model-Based Design Methodologies

**FIGURE 3.5** An MPEG-2 video encoder specification. (a) MPEG2 encoder (top); (b) inside the FSM; (c) I frame encoder; (d) P frame encoder; (e) B frame encoder.

**FIGURE 3.6** A block diagram of an MPEG-4 decoder that is specified in terms of scenario-aware dataflow. (Adapted from Theelen, B.D. et al., A scenario-aware data flow model for combined long-run average and worst-case performance analysis, in Proceedings of the International Conference on Formal Methods and Models for Codesign, Washington DC, 2006.)
3.6 LANGUAGES AND TOOLS FOR MODEL-BASED DESIGN

In this section, we survey research on tools for model-based design of embedded systems, with an emphasis on tool capabilities that are relevant to video processing systems. We discuss several representative tools that employ established and experimental models of computation and provide features for simulation, rapid prototyping, synthesis, and optimization. For more extensive coverage of model-based design tools for video processing systems and related application areas, we refer the reader to Reference 19.

3.6.1 CAL

CAL is a language for dataflow programming that has been applied to hardware and software synthesis and a wide variety of applications, with a particular emphasis on applications in video processing [20]. One of the most important applications of CAL to date is the incorporation of a subset of CAL, called RVC-CAL, as part of the MPEG reconfigurable video coding (RVC) standard [49]. In CAL, dataflow actors are specified in terms of entities that include actions, guards, port patterns, and priorities. An actor can contain any number of actions, where each action describes a specific computation that is to be performed by the actor, including the associated consumption and production of tokens at the actor ports, when the action is executed. Whether or not an action can be executed at any given time depends in general on the number of available input tokens, the token values, and the actor state. These fireability conditions are specified by input patterns and guards of the action definition. The relatively high flexibility allowed for constructing firing conditions makes CAL a very general model, where fundamental scheduling-related problems become undecidable, as with Boolean dataflow and other highly expressive, “dynamic dataflow” models.

Input patterns also declare variables that correspond to input tokens that are consumed when the action executes and that can be referenced in specifying the computation to be performed by the action. Such deep integration of dataflow-based, actor interface specification with specification of the detailed internal computations performed by an actor is one of the novel aspects of CAL.

Priorities in CAL actor specifications provide a way to select subsets of actions to execute when there are multiple actions that match the fireability conditions defined by the input patterns and guards. For more details on the CAL language, we refer the reader to the CAL language report [50].

A wide variety of tools has been developed to support design of hardware and software systems using CAL. For example, OpenDF was introduced in Reference 22 as an open-source simulation and compilation framework for CAL; the open RVC-CAL compiler (Orcc) is an open-source compiler infrastructure that enables code generation for a variety of target languages and platforms [51]; Boutellier et al. present a plug-in to OpenDF for multiprocessor scheduling of RVC systems that are constructed using CAL actors [52]; and Gu et al. present a tool that automatically extracts and exploits statically schedulable regions from CAL specifications [53].

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Descriptive Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD</td>
<td>Frame detector</td>
<td>D</td>
</tr>
<tr>
<td>IDCT</td>
<td>Inverse discrete cosine transformation</td>
<td>K</td>
</tr>
<tr>
<td>MC</td>
<td>Motion compensation</td>
<td>K</td>
</tr>
<tr>
<td>RC</td>
<td>Reconstruction</td>
<td>K</td>
</tr>
<tr>
<td>VLD</td>
<td>Variable-length decoding</td>
<td>K</td>
</tr>
</tbody>
</table>

© 2016 by Taylor & Francis Group, LLC
3.6.2 COMPAAN

MATLAB® is one of the most popular programming languages for algorithm development, and high-level functional simulation for DSP applications. In the Compaan project, developed at Leiden University, systematic techniques have been developed for synthesizing embedded software and FPGA-based hardware implementations from a restricted class of MATLAB programs known as parameterized, static nested loop programs [54]. In Compaan, an input MATLAB specification is first translated into an intermediate representation based on the Kahn process network model of computation [26]. The Kahn process network model is a general model of data-driven computation that subsumes as a special case the dataflow process networks mentioned earlier in this chapter. Like dataflow process networks, Kahn process networks consist of concurrent functional modules that are connected by FIFO buffers with nonblocking writes and blocking reads; however, unlike the dataflow process network model, modules in Kahn process networks do not necessarily have their execution decomposed a priori into well-defined, discrete units of execution [27].

Through its aggressive dependence analysis capabilities, Compaan combines the widespread appeal of MATLAB at the algorithm development level with the guaranteed determinacy, compact representation, simple synchronization, and distributed control features of Kahn process networks for efficient hardware/software implementation.

Technically, the Kahn process networks derived in Compaan can be described as equivalent cyclo-static dataflow graphs [55,56] and therefore fall under the category of dataflow process networks. However, these equivalent cyclo-static dataflow graphs can be very large and unwieldy to work with, and therefore, analysis in terms of the Kahn process network model is often more efficient and intuitive.

The development of the capability for translation from MATLAB to Kahn process networks was originally developed by Kienhuis, Rijpkema, and Deprettere [57], and this capability has since evolved into an elaborate suite of tools for mapping Kahn process networks into optimized implementations on heterogeneous hardware/software platforms consisting of embedded processors and FPGAs [54]. Among the most interesting optimizations in the Compaan tool suite are dependence analysis mechanisms that determine the most specialized form of buffer implementation, with respect to reordering and multiplicity of buffered values, for implementing interprocess communication in Kahn process networks [58].

Commercialization of the Compaan Technology is presently being explored as part of a Big Data effort in the field of astronomy. The Compaan tool set is used to program multi-FPGA boards from C-code for real-time analysis of astronomy data.

At Leiden University, Compaan has been succeeded by the Daedalus project, which provides a design flow for mapping embedded multimedia applications onto multiprocessor SoC devices [59].

3.6.3 PREESM

Parallel and Real-time Embedded Executives Scheduling Method (PREESM) is an extensible, Eclipse-based framework for rapid programming of signal processing systems [21,60]. Special emphasis is placed in PREESM for integrating and experimenting with different kinds of multiprocessor scheduling techniques and associated target architecture models. Such modeling and experimentation is useful in the design and implementation of real-time video processing systems, which must often satisfy stringent constraints on latency, throughput, and buffer memory requirements.

Various types of tools for compilation, analysis, scheduling, and architecture modeling can be integrated into PREESM as Eclipse [61] plug-ins. Existing capabilities of PREESM emphasize the use of architecture models and scheduling techniques that are targeted to mapping applications onto Texas Instruments programmable digital signal processors, including the TMS320C64x+ series of processors. Applications are modeled in PREESM using SDF graphs,
while target architectures are modeled as interconnections of abstracted processor cores, hardware coprocessors, and communication media. Both homogeneous and heterogeneous architectures can be modeled, and emphasis also is placed on careful modeling of DMA-based operation associated with the communication media.

Multiprocessor scheduling of actors in PREESM is performed using a form of list scheduling. A randomized version of the list scheduling algorithm is provided based on probabilistic generation of refinements to the schedule derived by the basic list scheduling technique. This randomized version can be executed for an arbitrary amount of time, as determined by the designer, after which the best solution observed during the entire execution is returned. Additionally, the randomized scheduling algorithm can be used to initialize the population of a genetic algorithm, which provides a third alternative for multiprocessor scheduling in PREESM. A plug-in for edge scheduling is provided within the scheduling framework of PREESM to enable the application of alternative methods for mapping interprocessor communication operations across the targeted interconnection of communication media.

Pelcat et al. present a study involving the application of PREESM to rapid prototyping of a stereo vision system [62].

3.6.4 PTOLEMY

The Ptolemy project at U.C. Berkeley has had considerable influence on the general trend toward viewing embedded systems design in terms of models of computation [30,41]. The design of Ptolemy emphasizes efficient modeling and simulation of embedded systems based on the interaction of heterogeneous models of computation. A key motivation is to allow designers to represent each subsystem of a design in the most natural model of computation associated with that subsystem, and allow subsystems expressed in different models of computation to be integrated seamlessly into an overall system design.

A key constraint imposed by the Ptolemy approach to heterogeneous modeling is the concept of hierarchical heterogeneity [63]. It is widely understood that in hierarchical modeling, a system specification is decomposed into a set $C$ of subsystems in which each subsystem can contain one or more hierarchical components, each of which represents another subsystem in $C$. Under hierarchical heterogeneity, each subsystem in $C$ must be described using a uniform model of computation, but the nested subsystem associated with a hierarchical component $H$ can be expressed (refined) in a model of computation that is different from the model of computation that expresses the subsystem containing $H$.

Thus, under hierarchical heterogeneity, the integration of different models of computation must be achieved entirely through the hierarchical embedding of heterogeneous models. A key consequence is that whenever a subsystem $S_1$ is embedded in a subsystem $S_2$ that is expressed in a different model of computation, the subsystem $S_1$ must be abstracted by a hierarchical component in $S_2$ that conforms to the model of computation associated with $S_2$. This provides precise constraints for interfacing different models of computation. Although these constraints may not always be easy to conform to, they provide a general and unambiguous convention for heterogeneous integration, and perhaps even more importantly, the associated interfacing methodology allows each subsystem to be analyzed using the techniques and tools available for the associated model of computation.

Ptolemy has been developed through a highly flexible, extensible, and robust software design, and this has facilitated experimentation with the underlying modeling capabilities by many research groups in various aspects of embedded systems design. Major areas of contribution associated with the development of Ptolemy that are especially relevant for video processing systems include hardware/software codesign, as well as contributions in dataflow-based modeling, analysis, and synthesis (e.g., see References 35, 64–66).

The current incarnation of the Ptolemy project, called Ptolemy II, is a Java-based tool that furthers the application of model-based design and hierarchical heterogeneity [30] and provides an even more malleable software infrastructure for experimentation with new techniques involving models of computation.
An important theme in Ptolemy II is the reuse of actors across multiple computational models. Through an emphasis in Ptolemy II on support for domain polymorphism, the same actor definition can in general be applicable across a variety of models of computation. In practice, domain polymorphism greatly increases the reuse of actor code. Techniques based on interface automata [67] have been developed to systematically characterize the interactions between actors and models of computation and reason about their compatibility (i.e., whether or not it makes sense to instantiate an actor in specifications that are based on a given model) [68].

3.6.5 SysteMoc

SysteMoc is a SystemC-based library for dataflow-based, hardware/software codesign and synthesis of signal processing systems. (See Section 3.7 for more details about SystemC, the simulation language on which SysteMoc was developed.) SysteMoc is based on a form of dataflow in which the model for each actor $A$ includes a set $F$ of functions and a FSM called the firing FSM of $A$. Each function $f \in F$ is classified as either an action function or a guard function. The action functions provide the core data processing capability of the actor, while the guard functions determine the activation of transitions in the firing FSM. Guard functions can access values of tokens present at the input edges of an actor (without consuming them), thereby enabling data-dependent sequencing of actions through the firing FSM. Furthermore, each transition $t$ of the firing FSM has an associated action function $x(t) \in F$, which is executed when the transition $t$ is activated. Thus, SysteMoc provides an integrated method for specifying, analyzing, and synthesizing actors in terms of FSMs that control sets of alternative dataflow behaviors.

SysteMoc has been demonstrated using a design space exploration case study for FPGA-based implementation of a 2D inverse DCT, as part of a larger case study involving an MPEG-4 decoder [69]. This case study considered a 5D design evaluation space encompassing throughput, latency, number of lookup tables (LUTs), number of flip-flops, and a composite resource utilization metric involving the sum of block RAM and multiplier resources. Among the most impressive results of the case study was the accuracy with which the design space exploration framework developed for SysteMoc was able to estimate FPGA hardware resources. For more details on SysteMoc and the MPEG-4 case study using SysteMoc, we refer the reader to Reference 69.

3.7 Simulation

Simulation is very important in SoC design. Simulation is not limited to functional verification, as with logic design. SoC designers use simulation to measure the performance and power consumption of their SoC designs. This is due in part to the fact that much of the functionality is implemented in software, which must be measured relative to the processors on which it runs. It is also due to the fact that the complex input patterns inherent in many SoC applications do not lend themselves to closed-form analysis.

SystemC is a simulation language that is widely used to model SoCs [70]. SystemC leverages the C++ programming language to build a simulation environment. SystemC classes allow designers to describe a digital system using a combination of structural and functional techniques. SystemC supports simulation at several levels of abstraction. Register-transfer-level simulations, for example, can be performed with the appropriate SystemC model. SystemC is most often used for more abstract models. A common type of model built in SystemC is a transaction-level model. This style of modeling describes the SoC as a network of communicating machines, with explicit connections between the models and functional descriptions for each model. The transaction-level model describes how data are moved between the models.

Hardware/software cosimulators are multimode simulators that simultaneously simulate different parts of the system at different levels of detail. For example, some modules may be simulated in register-transfer mode, while software running on a CPU is simulated functionally. Cosimulation is particularly useful for debugging the hardware/software interface, such as debugging driver software.
Functional validation, performance analysis, and power analysis of SoCs require simulating large numbers of vectors. Video and other SoC applications allow complex input sequences. Even relatively compact tests can take up tens of millions of bytes. These long input sequences are necessary to run the SoC through a reasonable number of the states implemented in the system. The large amounts of memory that can be integrated into today’s systems, whether they be on-chip or off-chip, allow the creation of SoCs with huge numbers of states that require long simulation runs.

Simulators for software running on processors have been developed over the past several decades. The Synopsys Virtualizer, for example, provides a transaction-level modeling interface for software prototyping. Both computer architects and SoC designers need fast simulators to run the large benchmarks required to evaluate architectures. As a result, a number of simulation techniques covering a broad range of accuracy and performance have been developed.

A simple method of analyzing a program’s execution behavior is to sample the program counter (PC) during program execution. The Unix prof command is an example of a PC-sampling analysis tool. PC sampling is subject to the same limitations on sampling rate as any other sampling process, but sampling rate is usually not a major concern in this case. A more serious limitation is that PC sampling gives us relative performance but not absolute performance. A sampled trace of the PC tells us where the program spent its time during execution, which gives us valuable information about the relative execution time of program modules that can be used to optimize the program. But it does not give us the execution time on a particular platform—especially if the target platform is different than the platform on which the trace is taken—and so we must use other methods to determine the real-time performance of programs.

Some simulators concentrate on the behavior of the cache, given the major role of the cache in determining overall system performance. The Dinero simulator (http://pages.cs.wisc.edu/~markhill/DineroIV/) is a well-known example of a cache simulator. These simulators generally work from a trace generated from the execution of a program. The program to be analyzed is augmented with additional code that records the execution behavior of the program. The Dinero simulator then reconstructs the cache behavior from the program trace. The architect can view the cache in various states or calculate cache statistics.

Some simulation systems model the behavior of the processor itself. A functional CPU simulator models instruction execution and maintains the state of the programming model, that is, the set of registers visible to the programmer. The functional simulator does not, however, model the performance or energy consumption of the program’s execution.

A cycle-accurate simulator of a CPU is designed to accurately predict the number of clock cycles required to execute every instruction, taking into account pipeline and memory system effects. The CPU model must therefore represent the internal structure of the CPU accurately enough to show how resources in the processor are used. The SimpleScalar simulation tool [71] is a well-known toolkit for building cycle-accurate simulators. SimpleScalar allows a variety of processor models to be built by a combination of parameterization of existing models and linking new simulation modules into the framework.

Power simulators are related to cycle-accurate simulators. Accurate power estimation requires models of the CPU microarchitecture at least as detailed as those used for performance evaluation. A power simulator must model all the important wires in the architecture since capacitance is a major source of power consumption. Wattch [72] and SimplePower [73] are the two best-known CPU power simulators.

### 3.8 HARDWARE/SOFTWARE COSYNTHESIS

Hardware/software cosynthesis tools allow system designers to explore architectural trade-offs. These tools take a description of a desired behavior that is relatively undifferentiated between hardware and software. They produce a heterogeneous hardware architecture and the architecture for the software to run on that platform. The software architecture includes the allocation of software tasks to the processing elements of the platform and the scheduling of computation and communication.
3.8 Hardware/Software Cosynthesis

The functional description of an application may take several forms. The most basic is a task graph, as shown in Figure 3.7. The graph describes data dependencies between a set of processes. Each component of the graph (i.e., each set of connected nodes) forms a task. Each task runs periodically and every task can run at a different rate. The task graph model generally does not concern itself with the details of operations within a process. The process is characterized by its execution time. Several variations of task graphs that include control information have been developed. In these models, the output of a process may enable one of several different processes.

Task graph models are closely related to the dataflow graph models introduced in Section 3.5.1. The difference often lies in how the models are used. A key difference is that in dataflow models, emphasis is placed on precisely characterizing and analyzing how data are produced and consumed by computational components, while in task graph models, emphasis is placed on efficiently abstracting the execution time or resource utilization of the components or on analyzing real-time properties. In many cases, dataflow graph techniques can be applied to the analysis or optimization of task graphs and vice versa. Thus, the terms “task graph” and “dataflow graph” are sometimes used interchangeably.

An alternative representation for behavior is a programming language. Several different code-sign languages have been developed and languages like SystemC have been used for cosynthesis as well. These languages may make use of constructs to describe parallelism that were originally developed for parallel programming languages. Such constructs are often used to capture operator-level concurrency. The subroutine structure of the program can be used to describe task-level parallelism.

The most basic form of hardware/software cosynthesis is hardware/software partitioning. As shown in Figure 3.8, this method maps the design into an architectural template. The basic system architecture is bus based, with a CPU and one or more custom hardware processing elements attached to the bus. The type of CPU is determined in advance, which allows the tool to accurately estimate software performance. The tool must decide what functions go into the custom processing elements; it must also schedule all the operations, whether implemented in hardware or software. This approach is known as hardware/software partitioning because the bus divides the architecture into two partitions and partitioning algorithms can be used to explore the design space.

Two important approaches to searching the design space during partitioning were introduced by early tools. The Vulcan system [74] starts with all processes in custom processing elements and iteratively moves selected processes to the CPU to reduce the system cost. The COSYMA system [75] starts with all operations running on the CPU and moves selected operations from loop nests into the custom processing element to increase performance.

Hardware/software partitioning is ideally suited to platform FPGAs, which implement the bus-partitioned structure and use FPGA fabrics for the custom processing elements. However,
the cost metric is somewhat different than in custom designs. Because the FPGA fabric is of a
fixed size, using more or less of the fabric may not be important, so long as the design fits into the
amount of logic available.

Other cosynthesis algorithms have been developed that do not rely on an architectural
template. Kalavade and Lee [64] alternately optimize for performance and cost to generate
a heterogeneous architecture. Wolf [76] alternated cost reduction and load balancing while
maintaining a performance-feasible design. Dick and Jha [77] used genetic algorithms to search
the design space.

Scheduling is an important task during cosynthesis. A complete system schedule must ulti-
mately be constructed; an important aspect of scheduling is the scheduling of multiple processes
on a single CPU. The study of real-time scheduling for uniprocessors was initiated by Liu and
Layland [78], who developed rate-monotonic scheduling (RMS) and earliest-deadline-first (EDF)
scheduling. RMS and EDF are priority-based schedulers, which use priorities to determine which
process to run next. Many cosynthesis systems use custom, state-based schedulers that deter-
mine the process to execute based upon the state of the system.

Design estimation is an important aspect of cosynthesis. While some software characteristics
may be determined by simulation, hardware characteristics are often estimated using high-level
synthesis. Henkel and Ernst [79] used forms of high-level synthesis algorithms to quickly synthe-
size a hardware accelerator unit and estimate its performance and size. Fornaciari et al. [80] used
high-level synthesis and libraries to estimate power consumption.

Software properties may be estimated in a variety of ways, depending on the level of abstrac-
tion. For instance, Li and Wolf [81] built a process-level model of multiple processes interacting
in the cache to provide an estimate of the performance penalty due to caches in a multitask-
ing system. Tiwari et al. [82] used measurements to build models of the power consumption of
instructions executing on processors.

3.9 SUMMARY

System-level design is challenging because it is heterogeneous. The applications that we want
to implement are heterogeneous in their computational models. The architectures on which we
implement these applications are also heterogeneous combinations of custom hardware, proces-
sors, and memory. As a result, system-level tools help designers manage and understand complex,
heterogeneous systems. Models of computation help designers cast their problem in a way that
can be clearly understood by both humans and tools. Simulation helps designers gather impor-
tant design characteristics. Hardware/software cosynthesis helps explore design spaces. As applica-
tions become more complex, expect to see tools continue to reach into the application space to
aid with the transition from algorithm to architecture.
REFERENCES


© 2016 by Taylor & Francis Group, LLC
58 References


