

DESIGN METHODOLOGY FOR EMBEDDED COMPUTER VISION SYSTEMS

Sankalita Saha and Shuvra S. Bhattacharyya

Abstract Computer vision has emerged as one of the most popular domains of embedded applications. The applications in this domain are characterized by complex, intensive computations along with very large memory requirements. Parallelization and multiprocessor implementations have become increasingly important for this domain, and various powerful new embedded platforms to support these applications have emerged in recent years. However, the problem of efficient design methodology for optimized implementation of such systems remains vastly unexplored. In this chapter, we look into the main research problems faced in this area and how they vary from other embedded design methodologies in light of key application characteristics in the embedded computer vision domain. We also provide discussion on emerging solutions to these various problems.

1 Introduction

Embedded systems that deploy computer vision applications are becoming common in our day-to-day consumer lives with the advent of cell-phones, PDAs, cameras, portable game systems, smart cameras and so on. The complexity of such embedded systems is expected to rise even further as consumers demand more functionality and performance out of such devices. To support such complex systems, new heterogeneous multiprocessor System-on-Chip (SoC) platforms have already emerged in the market. These platforms demonstrate the wide range of architectures available to designers today for such applications, varying from dedicated and programmable to configurable processors, such as programmable DSP, ASIC, FPGA subsystems,

Sankalita Saha
RIACS/NASA Ames Research Center, Moffett Field, CA e-mail: ssaha@riacs.edu

Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, University of Maryland, College Park, MD
e-mail: ssb@umd.edu

and their combinations. They not only consist of hardware components, but also integrate embedded software modules. Such heterogeneous systems pose new and difficult challenges in the design process, since now the designer not only has to take care of the effectiveness of hardware but also has to ensure the correctness and efficiency of software along multiple dimensions, such as response time, memory footprint, and power consumption.

In addition, newer, more sophisticated algorithms and product features emerge continually to keep up with the demands of consumers, and to help differentiate products in highly competitive markets. Balancing these specifications and their large computational and communication demands with the stringent size, power, and memory resource constraints of embedded platforms have created formidable new research challenges in design methodology for embedded systems — i.e., in the step-by-step process of starting from a given application specification, and deriving from it a streamlined hardware/software implementation. In this chapter, we present an overview of these various challenges, along with existing techniques and ongoing research directions to address the challenges.

To cope with the tight constraints on performance and cost that are typical of embedded systems, most designers use low-level programming languages such as C or assembly language for embedded software, and hardware description languages such as Verilog or VHDL for hardware. Although there are a number of tools emerging for creating and debugging such designs from higher levels of abstraction, they are generally not sophisticated enough to handle such complex systems and often designers have no choice but to manually design, implement and verify the systems. These are very time-consuming tasks since they not only involve embedded software and/or hardware design, but also interfacing of the various heterogeneous components. Aggravating this problem is the lack of standards for such interfaces. For example in the case of embedded software, because of performance and memory requirements, typically designers use application-dependent, proprietary operating systems, which vary from platform to platform.

Many design groups have enhanced their design methodologies to increase productivity and product quality by adopting object-oriented approaches, and other syntactically-driven methods. Although such methods aid in clarifying system structure and improving documentation, they are not sufficient to handle the details of diverse implementation platforms while ensuring quality and time to market. In some applications, the need to capture specifications at high abstraction levels has led to the use of modeling tools such as The MathWorks' MATLAB and Simulink tools. These tools let designers quickly assemble algorithms and simulate behavior. However, these tools do not cover the full embedded-system design spectrum, and hence do not generally lead to highly optimized final implementations.

Before we look into the details of the design process for embedded computer vision systems, it is important to have an understanding of the unique characteristics of this application domain, and the associated implementation constraints. Computer vision applications involve very high levels of computational complexity. Typically, these applications require complex math operations, including intensive floating point operations, as well as high volumes of memory transactions, since

large amounts of data need to be processed. These operations must be carried out and the outputs transmitted in a continuous manner while satisfying stringent timing constraints to ensure that the results are meaningful to the end-user. Therefore, computation time, and in particular, the processing throughput is of significant importance.

Two other important considerations are reduction of energy consumption to maximize battery life, and reduction of area requirements to minimize cost and size. These considerations limit the computational capabilities of the underlying processing platforms. Besides these important constraints, other performance metrics such as latency, jitter (unwanted variation of one or more characteristics of a periodic signal such as the interval between successive pulses, or the amplitude, frequency, or phase of successive cycles), and overall cost are used to evaluate a design as well. Although all of these metrics collectively may come across as common to many domains of embedded system design, what distinguishes computer vision systems is the relative importance of each of them. For example, due to the large volumes of data that need to be processed, computer vision applications require consistently high throughput, but can tolerate reasonable levels of jitter and packet errors. In contrast, consider audio applications, which typically manipulate much smaller volumes of data and hence do not require such a high bandwidth, but place tighter constraints on jitter and error rates. Motivated by the needs of embedded computer vision systems, the discussions in the remainder of this chapter focus mainly on implementation considerations and constraints associated with computational performance, area requirements, and energy consumption.

The lack of effective high level design methodologies and tools for embedded computer vision systems is a significant impediment to high-productivity product development, and to exploiting the full potential of embedded processing platforms for such systems. However, other aspects of the design and implementation process, such as algorithm selection/development and architecture design are also important problems. Thus, in this chapter, we categorize design and implementation for embedded computer systems into the following different subtasks:

- **Algorithms:** Due to the special characteristics of the targeted embedded platforms, various efforts have been spent on devising computer vision algorithms that are especially streamlined for this domain. Most such algorithms attempt to provide solutions in general to the high computational and memory requirements of the applications, while some also attempt to provide energy-efficient alternatives.
- **Architectures:** Innovative architectures for hardware subsystems already exist and continue to emerge to facilitate optimized implementation of embedded computer vision systems. These approaches range from hardware solutions — involving both system- and circuit-level optimizations — to efficient software methods.
- **Interfaces:** Interfaces can be viewed as “glue subsystems” that hold together a complete system implementation, and ensure the proper interoperability of its distinct components. Interfaces can consist of software as well as hardware components. The diverse, heterogeneous nature of state-of-the-art embedded plat-

forms makes the job of designing interfaces complex and necessitates new approaches.

- **Design methodology:** Design methodology deals with the actual job of developing a complete implementation given an algorithm (or a collection of algorithms that needs to be supported) and the targeted implementation platform. The task of design methodology is the main focus of this chapter. As we shall see, design methodology comprises of various important subproblems, each of which is complex and multi-faceted in itself. As a result, the subproblems associated with design methodology are often considered independent research problems, and a main aspect of design methodology is therefore how to relate, integrate, and develop better synergies across different solutions and methods that are geared towards these subproblems.

2 Algorithms

Because of resource constraints for the target platforms, algorithms for embedded computer vision and for embedded signal processing in general require special design efforts. Thus, optimized versions of various often-used sub-systems or low-level functions have been designed over the years, and packaged in ways to promote reuse in many implementations. Examples of such optimized signal processing library modules involve Gaussian noise generators, trigonometric functions such as *sin* or *cos* and computationally expensive functions such as fast Fourier transform computations. In general, certain characteristics make some computer vision algorithms better suited for embedded implementation. Such algorithm characteristics include sequential data access (as opposed to random access); multiple, independent or mostly-independent streams of data; and fixed or tightly-bounded sizes for data packets. However, all these features are not necessarily present in a given algorithm and hence various trade-offs needs to be considered. In [78], requirements for embedded vision systems and issues involved in software optimization to meet these requirements are analyzed. The authors proceed by first replacing optimized algorithms for various functions whenever they exist, followed by analyzing the bottleneck portions in the code, which are then appropriately rewritten after careful selection of data structures. Such an approach is a viable option for a large and complex system though it does not necessarily ensure a globally optimized design system.

Until now, most computer vision algorithms have been developed without considering in depth the target platform, and hence, aspects related to parallelization and distribution across hardware resources have conventionally been applied as a separate, later stage of design. However, in recent years, researchers have started exploring design of algorithms while considering the final implementation platforms, for example distributed algorithms for networks of smart cameras. Such algorithms take into account the distributed nature of image capture and processing that is enabled by environments where multiple cameras observe a scene from different viewpoints.

In [13], the authors present an approach for motion detection and analysis for gesture recognition for a two-camera system. The authors use MPI (Message Passing Interface) for communication between the cameras. To ensure efficient communication, it is imperative to minimize message length. This is done in several ways; one important approach being replacement of irregular shapes (to represent hands and face for gesture recognition) by regular geometric models such as ellipses so that only parameters for the model can be communicated instead of a large set of pixels. For distributed camera systems, one needs to develop vision algorithms using a different premise that considers the fact that a lot of redundant information may be present. Such a design space is considered in [55] that presents a multi-camera tracking system that uses several omnidirectional sensors and [24] where scene reconstruction using several uncalibrated cameras is presented. The trend shown by these works is encouraging. However, more effort in this direction involving more platform considerations are required. For example, memory size and requirements often pose major design bottlenecks for computer vision systems. Thus, memory architecture of the target platform need to be taken into account while designing the algorithms.

3 Architectures

Architectural exploration for embedded computer vision systems ranges from high-level system architecture to analog and circuit-level design. The architectural design space should not only include architectures for the main functional components, but should also encompass network architecture, since parallelization and spatial distribution are used increasingly for such systems. Software architectures are important as well, since in current hardware/software platforms, optimized embedded software architecture is essential for efficient implementation and high-productivity code development. Since new approaches in this area are numerous, the following discussion is by no means exhaustive, and is limited to a small, representative subset of approaches to help illustrate the variety of techniques that have been explored in this area.

New architectures initially were proposed for low-level functions such as edge-detection and smoothing. However, in recent years, new designs for complete embedded computer vision systems — made possible largely by the development of powerful new SoC platforms — have emerged. In general, the computational engines for these architectures can be classified as fast, customized single processors; networks of parallel processing units; and more recently, heterogeneous multiprocessor-on-chip (MPSoCs) devices that employ special accelerators.

Trimedia and Texas Instruments' Da Vinci VLIW are well-known commercial DSPs used in video processing. In the non-commercial domain, representative examples include *Imagine* a programmable stream processor by Kapasi et al. [33], the MOLEN reconfigurable microcoded processor developed at Delft University [42] and the HiBRID-SoC architecture for video and image processing by Berkovic et

al. [5]. *Imagine* was designed and prototyped at Stanford University and is the first programmable streaming-media processor that implements a stream instruction set. It can operate at 288 MHz at controlled voltage and temperature at which the peak performance is 11.8 billion 32-bit floating-point operations per second, or 23.0 billion 16-bit fixed point operations per second. The *Imagine* processor architecture has been commercialized resulting in STORM-1 from Stream Processors Inc. The *Imagine* architecture inspired other designs such as the *Cell* processor (jointly developed by SONY, IBM and TOSHIBA). However, unlike *Imagine* that can be programmed in C, the *Cell* processor cannot be programmed efficiently using standard sequential languages. The MOLEN reconfigurable processor utilizes microcode and custom-configured hardware to improve performance and caters to the application market that requires fast reconfigurability. It allows dynamic and static adaptation of the microarchitectures to fit application design requirements. The HiBRID-SoC integrates three fully programmable processor cores and various interfaces on a single chip. It operates at 145 MHz, and consumes 3.5 Watts. The processor cores are individually optimized to the particular computational characteristics of different application fields.

With the advent of powerful new FPGA platforms comprising of both hardware and software components, embedded computer vision architectures for FPGAs are becoming increasingly popular. In [67], a hardware/software implementation on a Xilinx FPGA platform is presented for a 3D facial pose tracking application; the most-computation intensive part was implemented in hardware while the remaining were implemented on the soft-core processors. A similar implementation of an optical-flow based object tracking algorithm is explored in [71] where matrix multiplication and matrix inversion operations were parallelized. Various architectures employ programmable DSPs with additional resources, such as special graphics controllers and reconfigurable logic devices, as shown in [39] and [51].

Since most computer vision systems employ intensive memory operations, an efficient memory architecture is required to prevent the memory system from becoming a major bottleneck in an implementation. A novel technique to reduce the on-chip memory size required for stream processing on MPSoC architectures is presented in [64]. This technique involves redistributing playout delay associated with the display device in a multimedia embedded system to processing elements on-chip connected in pipeline to the output device. Playout delay in this case is the artificial initial delay introduced before playing of received packet to ensure continuous output. The delay is introduced to make sure that all packets for a certain length of time (corresponding to the length of the playout buffer from which the output device reads) are received before starting their playout. In [16], the authors present a methodology to evaluate different memory architectures for a video signal processor. They show how variations in circuit sizes and configurations can help in determining the variations in the delay of both the memory system and the network; the associated delay curves can be used to design, compare, and choose from different memory system architectures. In [58], a comprehensive survey of memory optimizations for embedded systems is presented, where starting from architecture-independent optimizations such as transformations, direct optimization techniques

ranging from register files to on-chip memory, data caches, and dynamic memory (DRAM) are covered. Such a list of possible optimizations are important to consider in the design of vision systems, especially because of the extraordinary memory requirements involved.

Architectures for low-level computer vision algorithms mainly consist of arrangements of linear 1D arrays, 2D meshes of processing elements [23], systolic arrays of processing elements (PEs) (e.g., [12], [41]) and networks of transputers [73]. New analog as well as mixed-signal circuit designs have also been explored. In [73] and [75] analog implementation of particle-filter based tracking is explored, where non-linear functions such as exponential and arctangent computations are implemented using multiple-input, translinear element (MITE) networks. The use of mixed-signal circuits present an interesting option for computer vision systems since they can provide significant optimizations not achievable using digital circuits. However, they need to be explored judiciously for complex systems comprising of multiple sub-systems since they add further challenges to the already complex design process.

4 Interfaces

In our context, interfaces in general refer to the “glue” subsystems that connect the various components of an embedded computer system. Such interfaces include drivers, communication interfaces, I/O components, and middleware. An interface can be a software-based component or a hardware component. Middleware refers to the software layer that lies between the operating system and the applications at each “terminal” of a networked computing system. Many designers refer to any software in an embedded system as embedded software. However, in our discussions in this chapter, embedded software refers only to the application software and the associated APIs used to access various functions from within the application software. Thus, we exclude middleware from the notion of “embedded software” and instead, we consider middleware as part of the platform-based, interface infrastructure.

In a hardware/software platform, the role of the interface on the software side is to hide the CPU from the application developer under a low-level software layer ranging from basic drivers and I/O functionality to sophisticated operating systems and middleware. On the hardware side, the interface hides CPU bus details through a hardware adaptation layer besides making applications more portable among different hardware platforms. This layer can range from simple registers to sophisticated I/O peripherals, including direct memory access queues and complex data conversion and buffering systems. On a heterogeneous platform, interfaces also hide the varying characteristics of the computing elements, such as differences in operating frequencies (hidden through appropriate buffering), data widths, and instruction widths. The need to comprehensively handle such mismatches further complicates the design of the interfaces and makes the design process time-consuming because it requires knowledge of all the hardware and software components and their in-

teractions. In [31] the authors provide detailed insight into the interface between hardware and software components in MPSoC (Multi-processor System-on-Chip) architectures.

For computer vision systems, memory interfaces are of great importance because of the large memory requirements. Also, since almost all system architectures use data-parallelization, communication between the different parallel components — mostly involving streams of data — has to be carefully designed to enable maximum use of parallelization. In [81], the authors report a case study of multiprocessor SoC (MPSoC) design for a complex video encoder. The initial specification was provided in sequential C code that was parallelized to execute on four different processors. MPI was used for inter-task communication; but it required the design of an additional hardware-dependent software layer to refine the abstract programming model. The design was compiled by three types of designers — application software, hardware-dependent software and hardware platform designers — signifying the complexity of the interface design problem.

Various innovative interface designs have been explored in the multimedia domain. These interface designs generally extend to vision applications as well. For example, in [4], the problem of data storage and access optimizations for dynamic data types is addressed by using a component-based abstract data type library that can handle efficiently the dynamic data access and storage requirements of complex multimedia applications. For advanced DSP chips with multiple co-processors, Networks-on-chips (NoCs) are emerging as a scalable interconnect. Integration of co-processors with NoCs requires load/store packetizing wrappers on the network interfaces. Communication in such NoCs using a task transaction level high-level hardware interface is presented in [25].

Considering the popularity as well as the importance of parallelization in embedded computer vision systems to meet throughput requirements, efficient inter-processor communication is extremely important. A flexible and efficient queue-based communication library for MPSoCs called MP-queue is presented in [74]. Although, there are many powerful parallel hardware platforms available in the market, such as the *Cell* processor [62], Intel's quad-core processors, Stream Processor Inc.'s Storm-1 family [84] etc., there is a distinct lack of a standard communication interface that takes care of the associated heterogeneity while catering to the special needs of signal processing applications. In [67] and [70], an effort is described to create such a standard interface by merging two important existing paradigms — synchronous dataflow (see Section 5.1) and MPI — to formulate a new optimized communication interface for signal processing systems called the Signal Passing Interface (SPI). Software- as well as FPGA-based hardware communication libraries are created for SPI and tested on image processing applications as well as on other signal processing applications. Although pure synchronous dataflow semantics can model applications with *static* inter-module communication behaviors only, capability in SPI is provided to handle significant amounts of dynamic behavior through structured use of variable token sizes (called "virtual token sizes or VTS" in SPI terminology) [70]. Such interfaces are of significant importance, since they can be easily integrated to a existing design environments — in this case dataflow based

design flow — seamlessly. However, standards for most of the interfaces utilized are still lacking and more focussed attention on interface development and their optimization is required.

5 Design Methodology

As mentioned earlier, design methodology for embedded computer vision system implementation is a critical and challenging problem due to the increasing complexity in both the applications and targeted platforms. The problem can be decomposed into several, inter-related sub-problems: (1) modeling, specification and transformation; (2) partitioning and mapping; (3) scheduling; (4) design space exploration; and (5) code generation and verification. This is neither a rigid nor standard decomposition for the design and implementation process, but it highlights considerations that are of key importance in most design flows. Due to their strong inter-relationships, in many cases there is significant overlap between these sub-tasks. Similarly, various alternative categorizations into different sub-tasks exist. However, for the remainder of this chapter, we restrict ourselves to the specific decomposition above for concreteness and clarity. Note also that the overall design and implementation problem is typically addressed out through an iterative process — i.e., if at a given sub-task level, performance constraints are not met or the design is deemed to be otherwise unfeasible or undesirable, then re-design and subsequent re-assessment is carried out based on the findings of the previous iterations.

5.1 Modeling and Specification

A suitable model to specify an application is an extremely important first step towards an efficient implementation. The most popular approach for modeling and specification of embedded systems continues to be in terms of procedural programming languages, especially C. However, various formal models and formally-rooted specification languages exist for this purpose and such approaches are finding increasing use in certain domains, such as signal processing and control systems. Design using a well-suited, high-level formal model aids in a better understanding of the system behavior, as well as of the interaction between the various subsystems. Hence, formal models can be extremely useful in detecting problems early in the design stage. Also, when aided by an automatic code generation framework, such a design process can eliminate human errors, especially for complex systems such as computer vision systems.

A design (at all levels of the abstraction hierarchy) is generally represented as a set of components, which can be considered as isolated monolithic blocks, interacting with each other and with an environment that is not part of the design. A formal *model* defines the behavior and interaction of these blocks. Various formal models

in use for embedded system design include finite state machines, dataflow, Petri nets, and statecharts. Amongst these, dataflow is of significant importance since it is widely considered — due for example, to its correspondence with signal flow graph formulations — as one of the most natural and intuitive modeling paradigms for DSP applications. A formal *language*, on the other hand, allows the designer to specify inter-component interactions as well as sets of system constraints through a formal set of symbols and language grammar. To ensure a robust design, a language should have strong formal properties. Examples of such languages include ML [49], dataflow languages (e.g., Lucid [77], Haskell [14], DIF [27], CAL [18]) and synchronous languages (e.g., Luster, Signal, Esterel [22]).

Dataflow graphs provide one of the most natural and intuitive modeling paradigms for DSP systems. In the dataflow modeling paradigm, the computational behavior of a system is represented as a directed graph. A vertex or node in this graph represents a computational module or a hierarchically nested subgraph and is called an *actor*. A directed *edge* represents a FIFO buffer from a source actor to its sink actor. An edge can have a non-negative integer *delay* associated with it, which specifies the number of initial data values (*tokens*) on the edge before execution of the graph.

Dataflow graphs use a data-driven execution model. Thus, an actor can execute (*fire*) whenever it has sufficient numbers of data values (tokens) on all of its input edges to perform a meaningful computation. On firing, an actor consumes certain numbers of tokens from its input edges and executes based on its functionality to produce certain numbers of tokens on its output edges. Of all the dataflow models, synchronous dataflow (SDF), proposed by Lee and Messerschmitt [46], has emerged as the most popular model for DSP system design, mainly due to its compile-time predictability, and intuitive simplicity from a modeling viewpoint. However, SDF lacks significantly in terms of expressive power and is often not sufficient for modeling computer vision systems. Alternative DSP-oriented dataflow models, such as cyclo-static dataflow (CSDF) [11], parameterized dataflow [6], blocked dataflow (BLDF) [36], multi-dimensional dataflow [52] and windowed SDF [35] are considered more suitable for modeling computer-vision applications. These models try to extend the expressive power of SDF while maintaining as much compile-time predictability as possible.

Associated with the modeling step are transformations, which can be extremely beneficial for deriving optimized implementations. High-level transformations provide an effective technique for steering lower level steps in the design flow towards solutions that are streamlined in terms of given implementation constraints and objectives. These techniques involve transforming a given description of the system to another description that is more desirable in terms of the relevant implementation criteria. Although traditional focus has been on optimizing code-generation techniques and hence relevant compiler technology, high-level transformations, such as those operating at the formal dataflow graph level, have been gaining importance because of their inherent portability and resultant boost in performance when applied appropriately (e.g., [19], [48]).

Dataflow graph transformations can be of various kinds, such as algorithmic, architectural [59], and source-to-source [20]. These methods comprise of optimiza-

tions such as loop transformations [65], clustering [63], block processing optimization [66], [38] and so on. All of these techniques are important techniques to consider based on their relevance to the system under design.

However, most of these existing techniques are applicable to applications with static data rates. Transformation techniques that are more streamlined towards dynamically-structured (in a dataflow sense) computer vision systems have also come up in recent years, such as dynamic stream processing by Geilen and Basten in [21]. In [17], the authors present a new approach to express and analyze implementation-specific aspects in CSDF graphs for computer vision applications with concentration only on the channel/edge implementation. A new transformation technique for CSDF graphs is demonstrated in [69] where the approach was based on transforming a given CSDF model to an intermediate SDF model using clustering, thereby allowing SDF-based optimizations while retaining a significant amount of the expressive power and useful modeling details of CSDF. CSDF is gradually gaining importance as a powerful model for computer vision applications and thus optimization techniques for this model are of significant value.

5.2 *Partitioning and Mapping*

After an initial model of the system and specification of the implementation platform are obtained, the next step involves partitioning the computational tasks and mapping them onto the various processing units of the platform. Most partitioning algorithms involve computing the system's critical performance paths and hence require information about the performance constraints of the system. Partitioning and mapping can be applied at a macro as well as micro level. High-level coarse partitioning of the tasks can be identified early on and suitably mapped and scheduled, while pipelining within a macro task can be performed with detailed considerations of the system architecture. However, the initial macro partitioning may be changed later on in order to achieve a more optimized solution. The partitioning step is of course trivial for a single-processor system. However, for a system comprising multiple integrated circuits or heterogeneous processing units (CPUs, ASICs etc), this is generally a complex, multi-variable and multi-objective optimization problem.

Most computer vision algorithms involve significant amounts of data-parallelism and hence parallelization is frequently used to improve the throughput performance. However, parallelizing tasks across different processing resources does not in general guarantee optimal throughput performance for the whole system, nor does it ensure benefit towards other performance criteria such as area and power. This is because of the overheads associated with parallelization such as interprocessor communication, synchronization, optimal scheduling of tasks and memory management associated with parallelization. Since, intensive memory operations are another major concern, optimized memory architecture and associated data partitioning is of great importance as well. In video processing, it is often required to partition the image into blocks/tiles and then process or transmit these blocks — for example in

convolution or motion estimation. Such a partitioning problem has been investigated in [1]; the work is based on the concept that if the blocks used in images are close to squares then there is less data overhead. In [30], the authors look into dynamic data partitioning methods where processing of the basic video frames is delegated to multiple microcontrollers in a coordinated fashion; three regular ways to partition a full video frame which allows an entire frame can be divided into several *regions* (or slices), each region being mapped to one available processor of the platform for real-time processing. This allows higher frame rate with low energy consumption since different regions of a frame can be processed in parallel. Also, the frame partitioning scheme is decided adaptively to meet the changing characteristics of the incoming scenes. In [45], the authors address automatic partitioning and scheduling methods for distributed memory systems by using a compile-time processor assignment and data partitioning scheme. This approach aims to optimize the average run-time by partitioning of task chains with nested loops in a way that carefully considers data redistribution overheads and possible run-time parameter variations.

In terms of task-based partitioning, the partitioning algorithms depend on the underlying model being used for the system. For the case of dataflow graphs, various partitioning algorithms have been developed over the years in particular for synchronous dataflow graphs [32, 72]. However, as mentioned in section 5.1, other dataflow graphs allowing dynamic data interaction are of more significance. In [2], the authors investigate the system partitioning problem based on a constructive design space exploration heuristic for applications described by a control-data-flow specification.

5.3 Scheduling

Scheduling refers to the task of determining the execution order of the various functions on sub-systems in a design such that the required performance constraints are met. For a distributed or multiprocessor system, scheduling involves not only scheduling the execution order of the various processing units but also tasks on individual units. A schedule can be static, dynamic or a combination of both. In general, a statically determined schedule is the most preferred for the case of embedded systems since it avoids the run-time overhead associated with dynamic scheduling, and it also evolves in a more predictable way. However, for many systems it may not be possible to generate a static schedule because certain scheduling decisions may have to be dependent on the input or on some intermediate result of the system that cannot be predicted ahead of time. Thus, often a combination of static and dynamic schedules is used, where part of the schedule structure is fixed before execution of the system, and the rest is determined at run-time. The term *quasi-static scheduling* is used to describe scenarios in which a combination of static and dynamic scheduling is used, and a relatively large portion of the overall schedule structure is subsumed by the static component.

Scheduling for embedded system implementation have been studied in great details. However, the focus in this section is mainly on representative developments in the embedded computer vision domain. As mentioned earlier in section 5.1, dataflow graphs, in particular, new variants of SDF graphs have showed immense potential for modeling computer vision systems. Therefore, in this section we focus considerably on scheduling algorithms for these graphs. We start by first defining the problem of scheduling of dataflow graphs.

In the area of DSP-oriented dataflow-graph models, especially SDF graphs, a graph is said to have a *valid schedule* if it is free from deadlock and is sample rate consistent — i.e., it has a periodic schedule that fires each actor at least once and produces no net change in the number of tokens on each edge [46]. To provide for more memory-efficient storage of schedules, actor firing sequences can be represented through looping constructs [9]. For this purpose, a schedule loop, $L = (mT_1T_2\dots T_n)$, is defined as the successive repetition m times of the invocation sequence $T_1T_2\dots T_n$, where each T_i is either an actor firing or a (nested) schedule loop. A looped schedule $S = (T_1T_2\dots T_n)$, is an SDF schedule that is expressed in terms of the schedule loop notation define above. If every actor appears only once in S , then S is called a *single appearance schedule*, otherwise, is called a *multiple appearance schedule* [9].

The first scheduling strategy for CSDF graphs — a uniprocessor scheduling approach — was proposed by Bilsen et al. [10]. The same authors formulated computation of the minimum repetition count for each actor in a CSDF graph. Their scheduling strategy is based on a greedy heuristic that proceeds by adding one node at a time to the existing schedule; the node selected adds the minimum cost to the existing cost of the schedule. Another possible method is by decomposing a CSDF graph into an SDF graph [60]. However, it is not always possible to transform a CSDF graph into a deadlock-free SDF graph, and such an approach cannot in general exploit the versatility of CSDF to produce more efficient schedules. In [79], the authors provide an algorithm based on a min-cost network flow formulation that obtains close to minimal buffer capacities for CSDF graphs. These capacities satisfy both the time constraints of the system as well as any buffer capacity constraints that are for instance caused by finite memory sizes. An efficient scheduling approach for parameterized dataflow graphs is the quasi-static scheduling method presented in [7]. As described earlier, in a quasi-static schedule some actor firing decisions are made at run-time, but only where absolutely necessary.

Task graphs have also been used extensively in general embedded systems modeling and hence are of considerable importance for computer vision system. Scheduling strategies for task-graph models is explored by Lee et al. in [44] by decomposing the task graphs into simpler subchains, each of which is a linear sequence of tasks without loops. An energy-aware method to schedule multiple real-time tasks in multiprocessor systems that support dynamic voltage scaling (DVS) is explored in [80]. The authors used probabilistic distributions of the tasks' execution time to partition the workload for better energy reduction while using applications typical in a computer vision system for experiments. In [37], a novel data structure called the pipeline decomposition tree (PDT), and an associated scheduling framework, PDT scheduling, is presented that exploits both heterogeneous data parallelism and

task-level parallelism for scheduling image processing applications. PDT scheduling considers various scheduling constraints, such as number of available processors, and the amounts of on-chip and off-chip memory, as well as performance-related constraints (i.e., constraints involving latency and throughput) and generates schedules with different latency/throughput trade-offs.

5.4 Design Space Exploration

Design space exploration involves evaluation of the current system design and examination of alternative designs in relation to performance requirements and other relevant implementation criteria. In most cases, the process involves examining multiple designs and choosing the one that is considered to provide the best overall combination of trade-offs. In some situations, especially when one or more of the constraints is particularly stringent, none of the designs may meet all of the relevant constraints. In such a case, the designer may need to iterate over major segments of the design process to steer the solution space in a different direction. The number of platforms, along with their multi-faceted functionalities, together with a multi-dimensional design evaluation space result in an immense and complex design space. Within such a design space, one is typically able to evaluate only a small subset of solutions, and therefore it is important to employ methods that form this subset strategically. An efficient design space exploration tool can dramatically impact the area, performance, and power consumption of the resulting systems by focusing the designer's attention on promising regions of the overall design space. Such tools may also be used in conjunction with the individual design tasks themselves.

Although most of the existing techniques for design space exploration are based on simulations, some recent studies have started using formal models of computation (e.g., [34, 83]). Formal model based methods may be preferable in many design cases, in particular in the design of safety-critical systems, since they can provide frameworks for verification of system properties as well. For other applications, methods that can save on time — leading to better time-to-market — may be of more importance and hence simulation-based methods can be used. A methodology for system level design space exploration is presented in [3], where the focus is on partitioning and deriving system specifications from functional descriptions of the application. Peixoto et al. give a comprehensive framework for algorithmic and design space exploration along with definitions for several system-level metrics [61]. A design exploration framework that make estimations about performance and cost based on instruction set simulation of architectures is presented in [43]. A simple, yet intuitive approach to an architectural level design exploration is proposed in [68], which provides models for performance estimation along with means for comprehensive design space exploration. It exploits the concept of synchronization between processors, a function that is essential when mapping to parallel hardware. Such an exploration tool is quite useful, since it eliminates the task building a sep-

arate formal method and instead used a core form of functionality that is inevitable in such design and implementation is reused for exploration purposes.

In [82], Stochastic Automata Networks (SANs) have been used as an effective application-architecture formal modeling tool in system-level average-case analysis for a family of heterogeneous architectures that satisfy a set of architectural constraints imposed to allow re-use of hardware and software components. They demonstrate that SANs can be used early in the design cycle to identify the best performance/power trade-offs among several application-architecture combinations. This helps in avoiding lengthy simulations for predicting power and performance figures, as well as in promoting efficient mapping of different applications onto a chosen platform. A new technique based on probabilistically estimating the performance of concurrently executing applications that share resources is presented in [40]. The applications are modeled using SDF graphs while system throughput is estimated by modeling delay as the probability of a resource being blocked by actors. The use of such stochastic and probability based methods show an interesting and promising direction for design space exploration.

5.5 Code Generation and Verification

After all design steps involving formulation of application tasks and their mapping onto hardware resources, the remaining step of code generation for hardware and software implementation can proceed separately to a certain extent. Code generation for hardware typically goes through several steps: a description of behavior; a register-transfer level design, which provides combinational logic functions among registers, but not the details of logic design; the logic design itself; and the physical design of an integrated circuit, along with placement and routing. Development of embedded software often starts with a set of communicating processes, since embedded systems are effectively expressed as concurrent systems based on decomposition of the overall functionality into modules. For many modular design processes, such as those based on dataflow and other formal models of computation, this step can be performed from early on in the design flow, as described in Section 5. As the functional modules in the system decomposition are determined, they are coded in some combination of assembly languages and platform-oriented, high-level languages (e.g., C), or their associated code is obtained from a library of pre-existing intellectual property.

Various researchers have developed code generation tools for automatically translating high-level dataflow representations of DSP applications into monolithic software, and to a lesser extent, hardware implementations. Given the intuitive match between such dataflow representations and computer vision applications, these kinds of code generation methods are promising for integration into design methodologies for embedded computer vision systems. For this form of code generation, the higher level application is described as a dataflow graph, in terms of a formal, DSP-oriented model of computation, such as SDF or CSDF. Code for the

individual dataflow blocks (written by the designer or obtained from a library) is written in a platform-oriented language, such as C, assembly language, or a hardware description language. The code generation tool then processes the high level dataflow graph along with the intra-block code to generate a standalone implementation in terms of the targeted platform-oriented language. This generated implementation can then be mapped into the given processing resources using the associated platform-specific tools for compilation or synthesis.

An early effort on code generation from DSP-oriented dataflow graphs is presented in [26]. A survey on this form of code generation as well as C compiler technology for programmable DSPs is presented in [8]. Code generation techniques to automatically specialize generic descriptions of dataflow actors are developed in [53]. These methods provide for a high degree of automation and simulation-implementation consistency as dataflow blocks are refined from simulation-oriented form into implementation-oriented form. In [57], an approach to dataflow graph code generation geared especially for multimedia applications is presented. In this work, a novel fractional rate dataflow (FRDF) model [56] and buffer sharing based on strategic local and global buffer separation are used to streamline memory management. A code generation framework for exploring trade-offs among dataflow-based scheduling and buffer management techniques is presented in [28].

The final step before release of a product is extensive testing, verification and validation to ensure that the product meets all the design specifications. Verification and validation in particular are very important steps for safety-critical systems. There are many different verification techniques but they all basically fall into two major categories — dynamic testing and static testing. Dynamic testing involves execution of a system or component using numerous test cases. Dynamic testing can be further divided into three categories — functional testing, structural testing, and random testing. Functional testing involves identifying and testing all the functions of the system defined by the system requirements. Structural testing uses the information from the internal structure of a system to devise tests to check the operation of individual components. Both functional and structural testing both choose test cases that investigate a particular characteristic of the system. Random testing randomly chooses test cases among the set of all possible test cases in order to detect faults that go undetected by other systematic testing techniques. Exhaustive testing, where the input test cases consists of every possible set of input values, is a form of random testing. Although exhaustive testing performed at every stage in the life cycle results in a complete verification of the system, it is realistically impossible to accomplish. Static testing does not involve the operation of the system or component. Some of these techniques are performed manually while others are automated.

Validation techniques include formal methods, fault injection and dependability analysis. Formal methods involve use of mathematical and logical techniques to express, investigate and analyze the specification, design, documentation and behavior of both hardware and software. Formal methods mainly comprise of two approaches — model checking [85] which consists of a systematically exhaustive exploration of the mathematical model of the system and theorem proving [86] which comprises of logical inference using a formal version of mathematical reasoning about

the system. Fault injection uses intentional activation of faults by either hardware or software to observe the system operation under fault conditions. Dependability analysis involves identifying hazards and then proposing methods that reduces the risk of the hazard occurring.

6 Conclusions

In this chapter, we have explored challenges in the design and implementation of embedded computer vision systems in light of the distinguishing characteristics of these systems. We have also reviewed various existing and emerging solutions to address these challenges. We have studied these solutions by following a standard design flow that takes into account the characteristics of the targeted processing platforms along with application characteristics and performance constraints. Although new and innovative solutions for many key problems have been proposed by various researchers, numerous unsolved problems still remain, and at the same time, the complexity of the relevant platforms and applications continues to increase. With rising consumer demand for more sophisticated embedded computer vision (ECV) systems, the importance of ECV design methodology, and the challenging nature of this area are expected to continue and escalate, providing ongoing opportunities for an exciting research area.

References

1. Altılar D, Paker Y(2001) Minimum overhead data partitioning algorithms for parallel video processing. In: Proc. of 12th Intl. Conf. on Domain Decomposition Methods.
2. Auguin M, Bianco L, Capella L, Gresset E (2000) Partitioning conditional data flow graphs for embedded system design. In: IEEE Intl. Conf. on Application-Specific Systems, Architectures, and Processors, 2000, pp.339 - 348.
3. Auguin M, Capella L, Cuesta F, Gresset E(2001) CODEF: a system level design space exploration tool. In: Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing, 7-11 May 2001, Vol. 2, pp. 1145 - 1148.
4. Baloukas C, Papadopoulos L, Mamagkakis S, Soudris D (2007) Component based library implementation of abstract data types for resource management customization of embedded systems. In: Proc. of ESTIMEDIA 2007.
5. Berekovic M, Flugel S, Stolberg H.-J, Friebe L, Moch S, Kulaczewski M.B, Pirsch P (2003) HiBRID-SoC: a multi-core architecture for image and video applications. In: Proc. of 2003 Intl. Conf. on Image Processing, 14-17 Sept. 2003.
6. Bhattacharya B, Bhattacharyya S. S (2000) Parameterized dataflow modeling of DSP systems. In Proc. of the International Conference on Acoustics, Speech, and Signal Processing, Istanbul, Turkey, Jun. 2000, pp. 1948-1951.
7. Bhattacharya B, Bhattacharyya S. S(2000) Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In: Proc. of the International Workshop on Rapid System Prototyping, Paris, France, Jun. 2000, pp. 84-89.

8. Bhattacharyya S.S, Leupers R, Marwedel P (2000) Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Sep 2000, Vol.47, Issue.9, pp.849-875.
9. Bhattacharyya S. S, Murthy P. K, Lee E. A(1996) *Software Synthesis from Dataflow Graphs*. Boston, MA: Kluwer.
10. Bilsen G, Engels M, Lauwereins R, Peperstraete J(1994) Static scheduling of multi-rate and cyclostatic DSP applications. In: *Workshop on VLSI Signal Processing*, 1994, pp. 137-146.
11. Bilsen G, Engels M, Lauwereins R, Peperstraete J (1996) Cyclo-Static dataflow. *IEEE Transactions on Signal Processing*, Febr. 1996, Vol. 44, No.2, pp. 397-408.
12. Crisman J.D, Webb J.A (1991) The Warp Machine on Navlab. *IEEE Trans. Pattern Analysis and Machine Intelligence*, May 1991, vol. 13, no. 5, pp. 451465.
13. Daniels M, Muldawert K, Schlessman J, Ozert B, Wolf W (2007) Real-time human motion detection with distributed smart cameras. In: *First ACM/IEEE Intl. Conf. on Distributed Smart Cameras*, 25-28 Sept. 2007.
14. Davie A (1992) *An introduction to functional programming systems using Haskell*, Cambridge University Press.
15. Dutta S, Connor K.J, Wolf W, Wolfe A (1998) A design study of a 0.25- μ m video signal processor. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol.8, Aug. 1998, Issue.4, pp.501 - 519.
16. Dutta S, Wolf W, Wolfe A (1998) A methodology to evaluate memory architecture design tradeoffs for video signal processors. *IEEE Transactions on Circuits and Systems for Video Technology*, feb. 1998, Vol.8, Issue.1, pp. 36-53.
17. Denolf K, Bekooji M, Cockx J, Verkest D, Corporaal H(2007) Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *EURASIP Journal on Advances in Signal Processing*, doi:10.1155/2007/84078.
18. Eker J, Janneck J. W (2003), CAL Language Report: Specification of the CAL actor language. Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA, 94720, USA, Dec. 1, 2003.
19. Franke B, Boyle M. O(2001) An empirical evaluation of high level transformations for embedded processors. In: *Proc. of Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, Nov. 2001.
20. Franke B, Boyle M. O(2003) Array recovery and high-level transformations for DSP applications. *ACM TECS*, vol. 2, May 2003, pp. 132162.
21. Geilen M, Basten T(2004) Reactive process networks. In: *Proc. of the Intl. Workshop on Embedded Software*, Sept. 2004, pp. 137-146.
22. Halbwachs N (1993) *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers.
23. Hammerstrom D.W, Lulich D.P (1996) Image Processing Using One-Dimensional Processor Arrays. *Proc. IEEE*, July 1996, vol. 84, no. 7, pp. 1,0051,018.
24. Han M, Kanade T (2001) Multiple Motion Scene Reconstruction from Uncalibrated Views. In: *Proc. 8th IEEE Intl Conf. on Computer Vision*, vol. 1, 2001, pp. 163-170.
25. Henriksson T, Wolf P. V. D (2006) TTL Hardware Interface: A High-Level Interface for Streaming Multiprocessor Architectures. In: *Proc. of IEEE/ACM/IFIP Wkshp. on Embedded Systems for Real Time Multimedia*, Oct. 2006, pp. 107 - 112.
26. Ho W. H, Lee E. A, Messerschmitt D. G (1988), High Level Data Flow Programming for Digital Signal Processing. In: *Proc. of the International Workshop on VLSI Signal Processing*, 1988.
27. Hsu C, Bhattacharyya S. S (2005) Porting DSP applications across design tools using the dataflow interchange format. In: *Proc. of the Intl. Wkshp on Rapid System Prototyping*, Montreal, Canada, Jun. 2005, pp. 40-46.
28. Hsu D, Ko M, Bhattacharyya S. S (2005), Software Synthesis from the Dataflow Interchange Format. In: *Proc. of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, Sept. 2005, pp. 37-49.
29. Hu X, Greenwood G. W, Ravichandran S, Quan G (1999) A framework for user assisted design space exploration. In *36th Design Automation Conference*, New Orleans, Jun. 21-25, 1999.

30. Hu X, Marculescu R(2004) Adaptive data partitioning for ambient multimedia. In: Proc. of Design Automation Conferenc , June 711, 2004, San Diego, California, USA.
31. Jerraya A.A, Wolf W (2005) Hardware/Software Interface Codesign for Embedded Systems. Computer, Feb. 2005, Vol.38, Issue 2, pp. 63 - 69.
32. Kalavade A, Lee E (1995) The extended partitioning problem: hardware/software mapping and implementation-bin selection. In: Intl. Workshop on Rapid System Prototyping, Jun. 7-9, Chapel Hill, NC, 1995.
33. Kapasi U.J, Rixner S, Dally W.J, Khailany B, Ahn J. H, Mattson, P, Owens J.D (2003) Programmable stream processors. Computer, vol. 35, no. 8, Aug. 2003, pp. 54-62.
34. Karkowski I, Corporaal H(1998) Design space exploration algorithm for heterogeneous multiprocessor embedded system design. In: 35th Design Automation Conference, San Francisco, Jun. 15-18, 1998.
35. Keinert J, Haubelt C, Teich J (2006) Modeling and Analysis of Windowed Synchronous Algorithms. In: Proc. of the Intl. Conf. on Acoustics, Speech, and Signal Processing, May 2006.
36. Ko D, Bhattacharyya S. S (2005) Modeling of block-based DSP systems. Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, Jul. 2005, Vol. 40(3), pp:289-299.
37. Ko D, Bhattacharyya S. S (2006). The pipeline decomposition tree: An analysis tool for multiprocessor implementation of image processing applications. In: Proc. of the Intl. Conf. on Hardware/Software Codesign and System Synthesis, Seoul, Korea, Oct. 2006, pp. 52-57.
38. Ko M-Y, Shen C-C, Bhattacharyya S. S (2006). Memory-constrained Block Processing for DSP Software Optimization. In: Proc. of Embedded Computer Systems: Architectures, Modeling and Simulation, Jul. 2006, pp. 137-143.
39. Kshirsagar S. P, Harvey D. M, Hartley D. A, Hobson C. A (1994) Design and application of parallel TMS320C40-based image processing system. In: Proc. of IEE Colloquium on Parallel Architectures for Image Processing, 1994.
40. Kumar A, Mesman B, Corporaal H, Theelen B, Ha Y(2007) A Probabilistic Approach to Model Resource Contention for Performance Estimation of Multifeatured Media Devices. In: Proc. of Design Automation Conference, Jun. 4-8, San Diego, USA.
41. Kung S. Y (1988) VLSI Array processors. Prentice Hall.
42. Kuzmanov G.K, Gaydadjiev G. N, Vassiliadis S (2005) The Molen media processor: design and evaluation. In: Proc. of the International Workshop on Application Specific Processors, WASP 2005, New York Metropolitan Area, USA, September 2005, pp. 26-33.
43. Kwon S, Lee C, Kim S, Yi Y, Ha S(2004) Fast design space exploration framework with an efficient performance estimation technique. In: Proc. of 2nd Workshop on Embedded Systems for Real-Time Multimedia, 2004, pp. 27 - 32.
44. Lee C, Wang Y, Yang T(1994) Static global scheduling for optimal computer vision and image processing operations on distributed-memory multiprocessors. Tech. Report: TRCS94-23, University of California at Santa Barbara Santa Barbara, CA, USA.
45. Lee C, Yang T, Wang Y(1995) Partitioning and scheduling for parallel image processing operations. In: Proceedings of the 7th IEEE Symp. on Parallel and Distributed Processing, 1995.
46. Lee E. A, Messerschmitt D. G (1987) Static scheduling of synchronous dataflow programs for digital signal processing. IEEE Transactions on Computers, Vol. C-36, No. 2, Febr. 1987.
47. Lee H. G, Ogras U. Y, Marculescu R, Chang N (2006) Design space exploration and prototyping for On-chip multimedia applications. In: Proc. of Design Automation Conference, July 24-28, 2006, San Francisco, USA,
48. Marwedel P(2002) Embedded software: how to make it efficient. In: Proc. of the Euromicro Symp. on Digital System Design, Sept. 2002, pp. 201 207.
49. Milner R, Tofte M, Harper R (1990) The definition of Standard ML, MIT Press.
50. Miramond B, Delosme J(2005) Design space exploration for dynamically reconfigurable architectures. In: Proc. of Design Automation and Test in Europe, 2005, pp. 366-371.
51. Murphy C. W, Harvey D. M, Nicholson L. J (1999) Low cost TMS320C40/XC6200 based reconfigurable parallel image processing architecture. In: Proc. of IEEE Colloquium on Reconfigurable Systems, 10 Mar. 1999

52. Murthy P. K, Lee E. A (2002) Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, Aug. 2002, vol. 50, no. 8, pp. 2064-2079.
53. Neuendorffer S (2002), Automatic Specialization of Actor-Oriented Models in Ptolemy II. Master's Thesis, Dec. 2002, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.
54. Niemann R, Marwedel P(1997) An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, Vol. 2, N. 2, Kluwer, Mar. 1997.
55. Ng K. et al. (2004) An integrated surveillance system – human tracking and view synthesis using multiple omni-directional vision sensors. *Image and Vision Computing Journal*, Jul. 2004, vol. 22, no. 7, pp. 551-561.
56. Oh H, Ha S (2004), Fractional rate dataflow model for efficient code synthesis. *Jnl. of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, May 2004, Vol. 37, pp. 41-51.
57. Oh H, Ha S (2002) Efficient code synthesis from extended dataflow graphs for multimedia applications. In: *Proc. of 39th Design Automation Conference*, 2002, pp.275- 280.
58. Panda P. R, Catthoor F, Dutt N. D, Danckaert K, Brockmeyer E, Kulkarni C, Vandercappelle A, Kjeldsberg P. G (2001) Data and memory optimization techniques for embedded systems. *ACM Trans. on Design Automation of Electronic Systems*, , Apr. 2001, vol. 6, no. 2, pp. 149206.
59. Parhi K. K (1995) High-level algorithm and architecture transformations for DSP synthesis. *Journal of VLSI Signal Processing*, Jan. 1995.
60. Parks T. M, Pino J. L, Lee E. A (1995) A comparison of synchronous and cyclo-static dataflow. In *Proc. of IEEE Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, Oct. 29 - Nov. 1, 1995.
61. Peixoto H. P, Jacome M. F(1997) Algorithm and architecture-level design space exploration using hierarchical data flows. In: *Proc. of IEEE Intl. Conference on Application-Specific Systems, Architectures and Processors*, 14-16 Jul. 1997, pp. 272 - 282.
62. Pham D. et al (2006) Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Jnl. of solid-state circuits*, Jan. 2006, Vol. 41, Issue. 1, pp. 179-196.
63. Pino J. L, Bhattacharyya S. S, Lee E. A(1995) A hierarchical multiprocessor scheduling system for DSP applications. In: *Proc. of the IEEE Asilomar Conf. on Signals, Systems, and Computers*, Nov. 1995, vol.1, pp. 122126.
64. Raman B, Chakraborty S, Ooi W. T, Dutta S (2007) Reducing data-memory footprint of multimedia applications by delay redistribution. In: *Proc. of 44th ACM/IEEE Design Automation Conference*, 4-8 June 2007, San Diego, Ca, USA, pp:738-743.
65. Rim M, Jain R(1996) Valid transformations: a new class of loop transformations for high-level synthesis and pipelined scheduling applications. *IEEE Trans. on Parallel and Distributed Systems*, Apr. 1996, vol. 7, pp. 399-410.
66. Ritz S, Pankert M, Zivojnovic V, Meyr H(1993) Optimum vectorization of scalable synchronous dataflow graphs. In: *Proc. of Intl. Conf. on Application-Specific Array Processors*, 1993, pp. 285-296.
67. Saha S(2007) Design methodology for embedded computer vision systems. PhD Thesis, University of Maryland, College Park.
68. Saha S, Kianzad V, Schessman J, Aggarwal G, Bhattacharyya S. S, Wolf W, Chellappa R. An architectural level design methodology for smart camera applications. *Intl. Journal of Embedded Systems*, Special Issue on Optimizations for DSP and Embedded Systems (To appear).
69. Saha S, Puthenpurayil S, Bhattacharyya S. S(2006) Dataflow transformations in high-level DSP system design. In: *Proc. of the Intl. Symp. on System-on-Chip*, Tampere, Finland, Nov. 2006, pp. 131-136.
70. Saha S, Puthenpurayil S, Schlessman J, Bhattacharyya S. S, Wolf W (2007) An optimized message passing framework for parallel implementation of signal processing applications.

- In: Proc. of the Design, Automation and Test in Europe Conference and Exhibition, Munich, Germany, Mar. 2008.
71. Schlessman J, Chen C-Y, Wolf W, Ozer B, Fujino K, Itoh K (2006) Hardware/Software Co-Design of an FPGA-based Embedded Tracking System. In: Proc. of 2006 Conf. on Computer Vision and Pattern Recognition Workshop, 17-22 Jun., 2006.
 72. Sriram S, Bhattacharyya S. S (2000) Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker.
 73. Teoh E. K, Mital D. P Real-time image processing using transputers. In: Proc. of Intl. Conf. on Systems, Man and Cybernetics, 17-20 Oct. 1993, pp.505 - 510.
 74. Torre A. D, Ruggiero M, Benini L, Acquaviva A (2007) MP-Queue: an efficient communication library for embedded streaming multimedia platforms. In: Proc. of IEEE/ACM/IFIP Wkshp. on Embedded Systems for Real-Time Multimedia, 4-5 Oct. 2007, pp.105-110.
 75. Velmurugan R, Subramanian S, et.al. (2006) On low-power analog implementation of particle filters for target tracking. In: Proc. 14th European Signal Processing Conf. EUSIPCO, Sep. 2006.
 76. Velmurugan R, Subramanian S, et.al. (2007) Mixed-mode Implementation of Particle Filters. In: Proc. of IEEE PACRIM Conf., Aug. 2007.
 77. Wadge W, Ashcroft E. A (1985) Lucid, the dataflow programming language, Academic Press.
 78. Wardhani A. W, Pham B. L, (2002) Programming Optimisation for Embedded Vision. In: Proc. of DICTA2002: Digital Image Computing Techniques and Applications, Melbourne, Australia, 2122 Jan. 2002.
 79. Wiggers M. H, Bekooji M. J. G, Smit G. J. M (2007) Efficient computation of buffer capacities for cyclo-static dataflow graphs. In: Proc. of Design Automation Conference, Jun. 4-8, San Diego, USA.
 80. Xian C, Lu Y, Li Z(2007) Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In: Proc. of Design Automation Conference, Jun. 4-8, San Diego, USA.
 81. Youssef M, Sungjoo Y, Sasongko A, Paviot Y, Jerraya A.A (2004) Debugging HW/SW interface for MPSoC: video encoder system design case study. In: Proc. of 41st Design Automation Conference, 2004, pp.908- 913.
 82. Zamora N. H, Hu X, Marculescu R(2007) System-level performance/power analysis for platform-based design of multimedia applications. ACM Transactions on Design Automation of Electronic Systems, Jan. 2007, Vol. 12, No. 1, Article 2,
 83. Ziegenbein D, Ernest R, Richter K, Teich J, Thiele L(1998) Combining multiple models of computation for scheduling and allocation. In: Proc. of Codes/CASHE 1998, pp. 9-13.
 84. Wong W (2007) Architecture Maps DSP Flow To Parallel Processing Platform. In: Electronic Design, May 10, 2007.
 85. Clarke E. M, Grumberg O, Peled D (1999) Model Checking. MIT Press.
 86. Duffy D. A (1991) Principles of Automated Theorem Proving. John Wiley and Sons.