Fast and Accurate Multiprocessor Architecture Exploration with Symbolic Programs

Vladimir D. Živković
Ed Deprettere
LIACS - Leiden University
The Netherlands
{lale,edd}@liacs.nl

Erwin de Kock
Pieter van der Wolf
Philips Research Laboratories
The Netherlands
{erwin.de.kock,pieter.van.der.wolf}@philips.com

Abstract

In system-level platform-based embedded systems design, the mapping model is a crucial link between the application model and the architecture model. All three models must match when design-space exploration has to be fast and accurate, and when exploration methods and design methods have to be closely related. For the media processing application domain we present an architecture model and corresponding mapping model that meet these requirements better than previously proposed models. A case study illustrates this improvement.

1. Introduction

The increasing interest in embedded systems has heightened the need for methodologies and tools suitable for modelling, simulation, and design of embedded systems. We focus on heterogeneous embedded systems, i.e., those that mix programmable, configurable, and dedicated components. These systems are of particular interest since they are used as underlying platforms in multimedia and communications products. Such embedded systems tend to become increasingly complex and they are not easily designed. More and more they have to meet non-functional constraints that designers have to incorporate in their models. Models become more accurate when more details are added. However, this also increases system model development and simulation effort and time. To overcome this problem, the evaluation of design choices should be moved to the early phases of the design process. The cost of model construction and model evaluation is lower at the higher levels of abstraction, such that there are more opportunities to explore alternative realizations at these levels. Therefore, methodologies that deal with the exploration of embedded systems at the system level are of interest.

We assume in this paper that the Y-chart approach [1] is followed. Thus, the application, architecture, and mapping models are separated in three layers, each being specified appropriately. Given that the applications of interest are streaming based, we assume that they are specified as Kahn Process Networks (KPN) [2]. A KPN is one of many so-called Models of Computation (MoC). A KPN consists of executable *processes* that communicate point-to-point over unbounded FIFO *channels* and synchronise by means of *blocking reads*. The mapping layer receives a representation derived from the KPN application model, and transforms this representation to a representation that matches more closely the architecture model onto which the application is to be mapped.

In [3] we investigated Design-Space Exploration (DSE) approaches for streaming applications. We observed that the exploration approaches mainly differ in terms of the mapping layer representations. We identified two extremes, one being the *symbolic instruction trace* representation, the other one being the *executable control data-flow graph* representation. The former representation has led to the *Trace Driven* (TD) co-simulation exploration approach, whilst the latter representation is the designer's approach we refer to as the *Control Data-Flow Graph* (CDFG) approach. For example, in the SPADE framework [4] the mapping layer transforms application traces to architecture traces. Similarly, but not strictly according to the Y-chart, tools presented in [5] transform CDFGs.

These two extreme approaches each have their own pros and cons. To begin with, the two approaches aim at different objectives (exploration vs. design) that are hard to reconcile. Next, the CDFG representation is more powerful than the TD representation because in CDFGs control constructs are present and the representation covers all data sets. In the TD representation only the data dependent results of control constructs are represented. However, the CDFG representation has two drawbacks: (1) CDFGs are usually more complex than symbolic instruction traces, and (2) the

1530-1591/03 $17.00

CDFG representation is an executable representation: mapping this representation into an architecture implies that the architecture model has to capture both functional behaviour and timing behaviour. Therefore, the level of reusability of architectures for different CDFGs is restricted. Simply, different applications assume different CDFG representations, and, hence, different architecture CDFGs. Step by step, the designer moves to the detailed design stages, and he overlooks other design choices. This results in a longer time needed for DSE. The limitation of the TD representation is apparent in e.g., the SPADE exploration framework, where the communication aspects of the non-functional architecture models are very close to communication structure of the application models. Since limited information is captured in the traces, only simple transformations can be performed. Of course, there is nothing wrong with this tight coupling between application and architecture models when they do match as required. However, a typical "generic" *model of architecture* (MoA) is not the Kahn MoC. Hence, MoC and MoA do not match. Therefore, mapping requires more information than only traces of symbolic instructions.

In [3] we proposed a hybrid representation of the application model as an input to the mapping layer that unifies the best of the TD and the CDFG representations. In short, the application model represents itself to the mapping layer in terms of a symbolic CDFG called *symbolic program* (SP) - as opposed to symbolic instructions - plus a *control trace* in which the outcomes of conditional constructs are conveyed to the mapping layer. In addition, the symbolic program allows the annotation of possibly parallel symbolic instructions, i.e., offers a coarse model of *instruction level parallelism*. Because this representation is more powerful than the linear symbolic instruction trace representation, the architecture model, in particular the processing unit model is more powerful.

At a first glance, a symbolic program may look similar to *DF** and *grey box* models [5]. However, the *DF** models are for synthesis mainly, while with the symbolic program we aim at DSE too. Moreover, it seems that designers must posses a solid "know-how" in order to be able to derive *DF** models. On the contrary, symbolic programs can be easily generated automatically from an already available process network. Thus, *DF** and *grey box* models are closer to control data-flow graphs than to symbolic programs.

The rest of the paper is organised as follows. First, we give some general remarks about symbolic programs in Section 2. The architecture model suitable for the corresponding DSE approach is described in Section 3. In Section 4 we present the performance/cost numbers obtained when using these models in a simple, yet representative case study. Finally, we draw some conclusions in Section 5.

2. The Symbolic Program Approach

On the left-hand side of Figure 1, the typical sequence of activities using instruction traces is depicted. After the source code of an application model is annotated, the model is executed on a single data-set. Traces of the execution of an application model are collected and transformed to architecture-level traces, which are then consumed by an architecture simulator.

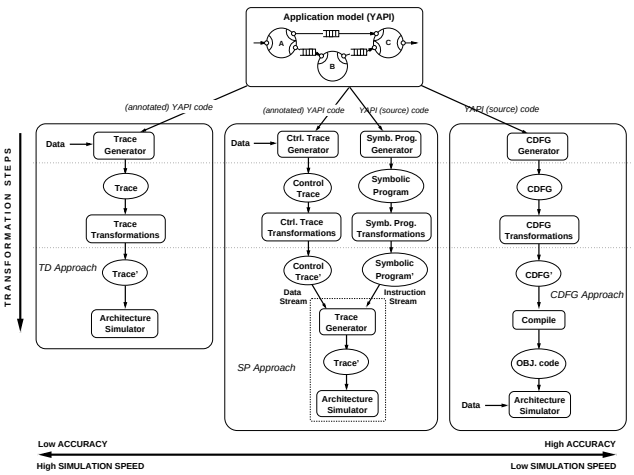
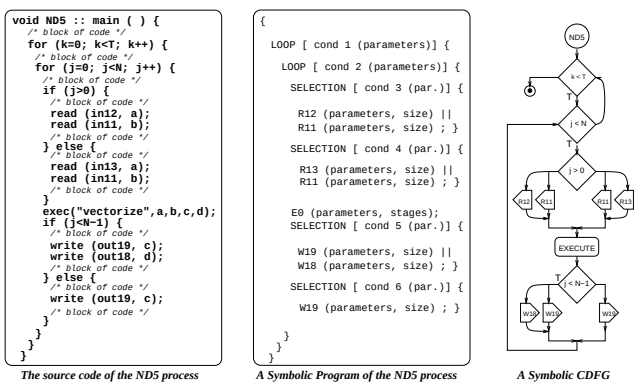


Figure 1. The SP Hybrid mapping approach

Similarly, on the right-hand side of Figure 1, the typical flow of activities using control data-flow graphs is shown. The source code of an application model is parsed and the structure of application processes is preserved in a form of CDFGs. The CDFGs are then transformed during mapping onto the specified architecture. The resulting CDFGs are compiled/synthesised into the implementation model (e.g., an object code or a hardware model), which is then executed an architecture simulator. During this execution a data-set is processed, as opposed to the instruction trace approach described formerly.

Finally, in the middle of Figure 1, the symbolic program flow is shown. This approach has emerged in an effort to obtain a fast and accurate exploration approach that is close to the design trajectory [3]. As expected, symbolic programs are located somewhere in-between the two extremes mentioned above. On one hand, symbolic programs are obtained in a similar way as control data-flow graphs - by parsing the application sources. On the other hand, control information is obtained by gathering the control traces that come out of the execution of an annotated application model. The control-trace is, of course, valid for a single data-set, since the data is processed prior to the architecture simulation. As a consequence, the architecture model does not need to capture the functional behaviour, while still being able to handle data dependent behaviour correctly [4].

Figure 1 also shows the positioning of the three approaches with respect to accuracy and simulation speed. Symbolic programs allow designers (1) to perform design-steps as in the case of detailed design (indicated with dashed lines in Figure 1), (2) to run fast simulations of architectures being explored, and (3) to have more accurate numbers than in the case of trace-driven simulations [6]. An example with the pseudo code of a symbolic program and its correspondence with real source code is shown in Figure 2. The listing on the left side gives a piece of the C++ code which specifies the functionality of one KPN process. The control flow and the coarse communication and computation are indicated with bold letters, while the other code is abstracted. Such abstraction leads us to the listing in the middle, which is the symbolic program of the original specification. Finally, the symbolic CDFG on the right side in Figure 1 shows the CDFG-like equivalent of the derived symbolic program.



The source code of the ND5 process *A Symbolic Program of the ND5 process* *A Symbolic CDFG*

Figure 2. An example symbolic program

3. The Symbolic Program Architecture Models

In the previous section we have motivated the symbolic program approach. Here we elaborate on architecture models that support the evaluation of the timing behaviour of systems being designed. There are a few key issues concerning architecture models for system-level exploration: (1) ease of modelling, (2) accuracy, and (3) simulation speed. The accuracy is determined by modelling capabilities. Modelling restrictions may hamper correct modelling and lead to inaccurate results.

Particularly, we focus on systems based on execution units connected via a point-to-point communication network. In the model, the units communicate tokens, so data is not really processed at the architecture level. We also assume that execution units do not support multitasking. However, we want to underline that this is just a first iteration of our work, and that in the near future we will also cover other types of communication networks as well as an explicit model of shared memory. We will also support multitasking on execution units.

Starting from an available process network, e.g., the KPN network[1] shown in Figure 3, a designer can describe and examine an architecture instance using a set of symbolic program modules we currently provide. Such architecture instance is shown in Figure 4. There are three basic generic architecture module types:

1. Symbolic Program Units (SPU),
2. Read-or-Write Interfaces, and
3. First-In-First-Out (FIFO) buffers.

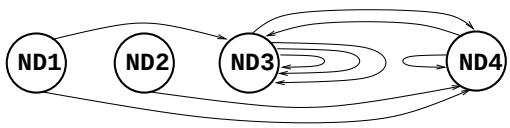


Figure 3. An application PN (QR unfold 1)

While the purpose of symbolic program units is to model the instruction level parallelism[2], the purpose of the other two modules is to allow modelling of task level parallelism that is used by executing different tasks on different SPUs in parallel. In the rest of this section we describe the properties of each of these modules.

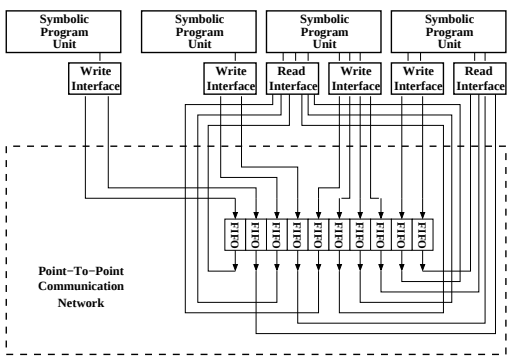


Figure 4. A symbolic program architecture

3.1. The Symbolic Program Unit - SPU

The aim of the symbolic program unit is to allow high-level modelling of both the instruction level parallelism and the reuse of available processing resources. The results of the case study performed in [6] indicate that the high-level trace driven execution unit in [4] has to be modified in order to be able to handle both the *horizontal* (VLIW-like) and *vertical* (superscalar-like) types of parallelism. We need the modelling support for both types of parallelism due to the fact that embedded platforms are heterogeneous. However, descending down to the lower levels of abstraction would

[1] Circles indicate processes, edges indicate infinite FIFOs.

[2] Actually, SPUs model high-level manifestations of ILP.

make this unit too dedicated, inflexible and, hence, not reusable. We provide a solution to this problem by combining the concept of symbolic programs [7] and the simulation engine of SystemC [8]. Note that symbolic program architectures are non-functional as mention above. The symbolic program unit contains parallel processing resources that are parametrised. On one hand, this internal parallelism of the architecture module allows a flexible deployment of the potentially parallel load originating from the application process, and thus, it allows designers to model both static and dynamic instruction scheduling [9] at a very high-level of abstraction. On the other hand, the potential parallelism available at compile time is easily expressed using symbolic programs. Jointly, these two properties result in the symbolic program unit that is illustrated in Figure 5.

The unit in Figure 5 can be divided in two parts: (1) the front-end part that schedules the program on the available resources - a number of Read, Write, and Execute Units - and (2) the back-end part that dispatches the instructions to these resources. Front-end and back-end parts communicate via the finite FIFO buffer[3], called *instruction stream*. The width of each buffer cell is defined by the number of resources and, from the high level point of view, determines whether a VLIW-like or a superscalar-like module is to be instantiated. The front-end part resembles a conditional execution, while the back-end part resembles an unconditional execution of instructions.

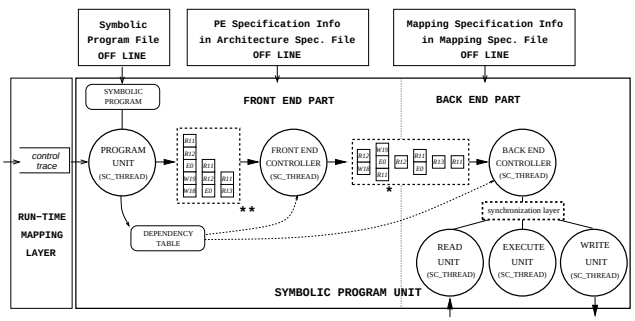


Figure 5. The symbolic program unit model

The front-end is subdivided in two units, each of which have an independent thread of execution. These are (1) the Program Unit, that parses the symbolic program, reads the control trace, and generates the stream of potentially parallel symbolic instructions, and (2) the Front-End Controller, which restricts this "partial order" of symbolic instructions into the one that can be handled by the available resources. The program unit and front-end controller communicate via a FIFO buffer[4]. The size of the buffer restricts the number of symbolic instructions that can be fetched in advance. Since multiple symbolic instructions are now allowed to be processed in parallel it is easier to exploit at run-time the compile-time transformations that expose such parallelism [10] on symbolic programs than on symbolic instruction traces. Additionally, in order to facilitate the overlapping of execution of symbolic instructions in hardware, the same unit builds a randomly accessible table with dependencies among symbolic instructions that are already fetched by the front-end controller but are not completed by the back-end controller.

The back-end part consists of the Back-End Controller, that dispatches the workload and synchronises the operation of the resources with the input flow of symbolic instructions, and a number of read, write, and execute units. Again, all units have their own thread of execution. The back-end controller can use information about dependencies among the pending symbolic instructions which is available in the table. Therefore, it can perform run-time reordering, which is equivalent to superscalar execution [9]. As a result, both the read and the write units can be reused by different read or write symbolic instructions (respectively).

3.2. Communication Interfaces

Communication interfaces connect symbolic program units and FIFOs. We illustrate this with a typical "producer-consumer" connection in Figure 6.

Figure 6. Communication refinement

An interface refines a task-level communication into explicit buffer synchronisation (dashed arrows in Figure 6) and data-transfer (white arrows in Figure 6). The idea is to split coarse read and write instructions into finer grained primitives for explicit synchronisation and data-transfer, and consequently, to allow overlapping of different synchronisation instructions [11]. Additionally, interfaces support connections between reusable read and write units at one side and communication channels at the other side. Since symbolic program units may reorder instructions, it is necessary to match sources and destinations of the produced tokens. To this end, a small cross-bar switch interconnect network is needed. This is illustrated in Figure 7 for the case of the Read Interface (the similar holds for the Write Interface).

There are two types of modules that can be recognised in the interface model introduced in Figure 7: (1) a FIFO Input Controller, which receives the requests from a particular read unit, and configures the connection between the read unit and the corresponding FIFO channel, and (2) a FIFO Input Unit, which interfaces the communication buffer from one side and the read unit master on the other side. The type of communication between the read unit and the FIFO input unit is a *pull handshake* (i.e., the receiver is a master

[3]The instruction stream FIFO buffer is marked with $*$ in Figure 5.

[4]This FIFO buffer is marked with $**$ in Figure 5.

and the sender is a slave), while the type of the communication between a write unit and a FIFO output unit is a *push handshake* (i.e., the sender is a master and the receiver is a slave) [12]. On the other hand, the FIFO input unit communicates with its counterpart in a peer interface (i.e., the FIFO output unit) using buffer synchronisation primitives [11]. For example, when the output unit writes a token to a FIFO, it will *signal data* to its input counterpart, which may be in a blocked state because it performed *check data* previously. A vice-versa example is when the input unit reads a token from a FIFO, it will *signal room* to its output counterpart, which may be in a blocked state because it executed *check room* previously.

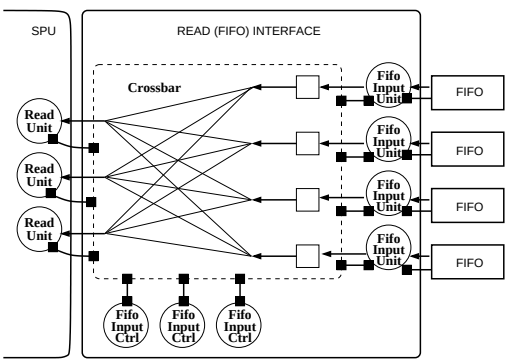


Figure 7. The Read Interface

Finally, all interface units have an independent thread of execution. The interface description (the number of units the interface contains) together with the mapping file (mapping of application ports onto the architecture/interface ports) constitute the input parameters for this module.

3.3. FIFO Communication Buffers

The FIFO Communication Buffer is characterised by two parameters: (1) a delay and (2) a buffer-size. The buffer size represents the capacity of the buffer.

We split the buffer delay in two parts: (1) the part that is incorporated in the buffer controllers (see Section 3.2) and (2) the part that represents the delay that has to pass before the notification of an event is recognised by blocked buffer controllers. Thus, we take into account the total communication delay instead of just the delay caused by blocking (synchronisation).

Additionally, instead of using the coarse read/write communication primitives, our buffer model uses the synchronisation and data-transfer primitives introduced in [11]. Hence, our FIFO Communication Buffer model is more refined and contains more details than the SPADE buffer model [4].

4. An Example Case Study

In order to verify the accuracy of our symbolic program approach, we reused the case study reported in [6] where the parallelised version of the adaptive QR algorithm was mapped on an FPGA board. The adaptive QR algorithm is used in the domain of signal processing, e.g., for the *adaptive beam-forming* with multiple antennas [6]. For more detailed information about the QR algorithm, see [13]. We briefly describe the benchmark in the next subsection. After that, we describe the cases we examined.

4.1. The Benchmark

The QR algorithm was given in an abstract representation and then partitioned into a process network as in Figure 3 using a tool called COMPAAN [14]. The QR process network was mapped on the FPGA board afterwards. The system architecture was specified similarly to the one shown in Figure 4. Each process was mapped on a single processing unit, while processing units communicated through finite FIFO buffers. Additionally, each processing unit was configured to perform several operations in parallel, and FIFO accesses coming from different units were pipelined. Finally, the size of the FIFO buffers was set to the value that guarantees deadlock-free execution.

Table 1. Previous results

$\frac{Network \rightarrow}{Case \downarrow}$	*unfold1*	*unfold3*	*unfold5*	*t*
FPGA	29281	9771	6111	≈10h
SPADE	6049	2021	1285	≈10m
SPADE+R/W	107395	35951	22474	≈30m

The system was instantiated for three different process networks described in [15] and [6]: the QR network with the unfold factors equal to 1, 3, and 5, respectively. The *unfold factor* assumes that the nested-loop programs of the original network are unfolded by this factor. Such transformation is known as a *loop unrolling* [16] and is used in compilers in order to increase the instruction-level parallelism. However, in [15] this transformation was used to increase task-level parallelism inside application networks. The simulation results in cycles and approximative simulation times are given in Table 1. The FPGA case is a reference to both mapping and very detailed VHDL cycle accurate simulation [15] [6]. As one can see, although in the SPADE case simulations took significantly less time than in the FPGA case, the accuracy of the SPADE simulations was very low.

4.2. Experiments and Results

We also applied our symbolic program approach to the case studies reported in [15] and [6]. We examined cases which illustrate effects of limited models and also in which models are flexible enough to capture characteristics of the real architecture. Concretely, in Table 2, Case 1 illustrates symbolic program units that schedule their programs purely

sequential and that communicate via non-pipelined FIFO buffers. Case 2 illustrates symbolic program units that behave like VLIWs and that communicate via non-pipelined FIFO buffers. Case 3 illustrates a more accurate model of the real architecture in which the symbolic program units behave like VLIWs and communicate via pipelined FIFO buffers.

Simulation was performed using the SystemC simulation engine [8]. Simulation results and the corresponding simulation times are shown in Table 2.

Table 2. Our results

$\frac{Network\rightarrow}{Case\downarrow}$	unfold1	unfold3	unfold5	t
case 1	107951	36111	22590	$\leq$10s
case 2	41073	13787	8653	$\leq$10s
case 3	29458	9884	6202	$\leq$10s

The results shown in Table 2 lead to the following observations: (1) the symbolic program approach very accurately model the reality, and (2) the models can more easily represent both application and architecture properties compared to instruction traces used in [4]. We assume that the reality is represented by the FPGA numbers given in Table 1. By comparison of the FPGA numbers with the numbers for Case 3 given in Table 2, we support the former observation. The clarification of the later observation requires a bit more explanation. In [3] we conducted an exploration on a conceptual level about the possible causes for the inaccurate results shown in Table 1. The preliminary results showed that inaccuracies are mainly due to the lack of the modelling capabilities in [4]. On the contrary, our approach exploits multithreading in symbolic program units and the expressiveness of our symbolic programs. Furthermore, the symbolic programs are fully scalable and parametrised, and thus, fully generic. Since each unit has its own thread of execution, one can easily check the time that each unit spends in each state, and therefore, can easily detect whether units are under-utilised. Such units are un-necessarily allocated units, i.e. over-designed. Finally, the simplicity of symbolic programs allows easy parsing and modifications from a fully-sequential to a VLIW-like schedule.

5. Conclusion

In this paper we have presented architecture models for design-space exploration of embedded systems. These models are built in a generic way, so they can be used to model various architecture characteristics. Further, the models can capture most of the parallelism that designers can express at compile time. This is due to the fact that they use symbolic programs as application workload instead of traces of symbolic instructions. We verified our model by regenerating the almost exact results of an earlier case study [6]. The model has been implemented using SystemC [8], which makes it reusable for a wider system-design community.

6. Acknowledgements

This work was performed in part in the Archer project, funded by Philips Semiconductors. We want to thank Bernhard Niemann (Fraunhofer Institute for Integrated Circuits, Germany) for his advises concerning SystemC.

References

[1] B. Kienhuis, et al., "An Approach for Quantitative Analysis of Application-specific Dataflow Architectures," in *Proc. of ASAP'97*, July, 1997.

[2] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing 74* - North-Holland Publishing Company, 1974.

[3] V. Živković et al., "Design Space Exploration of Streaming Multiprocessor Architectures," at *SiPS'02*, USA,Oct.,2002.

[4] P. Lieverse et al., "A methodology for architecture exploration of heterogeneous signal processing systems," in *Proc. SiPS'99*, Taiwan, Oct. 1999.

[5] N. Cossement et al., "DF*: An extension of synchronous dataflow with data dependency and non-determinism," in *FDL'00*, Germany, Sep. 2000.

[6] T. Harriss et al., "Compilation from Matlab to Process Networks Realized in FPGA," in *the 35th Asilomar Conf. on Signals, Systems, and Computers*, US, Nov. 2001.

[7] J. Larus, "Abstract execution: A technique for efficiently tracing programs," *Soft. Practice & Experience*, Dec.,1990.

[8] "SystemC Version 2.0 User's Guide," *Synopsys, Inc., CoWare Inc., Frontier Design, Inc.*, http://www.systemc.org/

[9] J. Hennessy et al., *Computer Architecture - A Quantitive Approach*, Morgan Kaufmann Publishers, 1996.

[10] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proc. of SIGPLAN'88*, USA, June, 1988.

[11] P. Lieverse et al., "A Trace Transformation Technique for Communication Refinement," in *Proc. CODES'01*, Denmark, Apr. 2001.

[12] A. Peeters, "Single-Rail Handshake Circuits," *PhD thesis, Technische Universiteit Eindhoven*, the Netherlands, 1996.

[13] J. Proakis et al., "Algorithms for Statistical Signal Processing," *Prentice Hall, Inc.*, 2002.

[14] A. Turjan et al., "The Compaan Tool Chain: Converting Matlab into Process Networks," in *DATE'02*, France, 2002.

[15] T. Stefanov et al., "Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances," in *Proc. CODES'02*, USA, May, 2002.

[16] S. Muchnick, "Advanced Compiler Design and Implementation," *Morgan Kaufmann Publishers, Inc.*, 1997.