An extensible framework for fast prototyping of multiprocessor dataflow applications

Jonathan Piat, Mickaël Raulet, Maxime Pelcat, Pengcheng Mu, Olivier Déforges

▶ **To cite this version:**

Jonathan Piat, Mickaël Raulet, Maxime Pelcat, Pengcheng Mu, Olivier Déforges. An extensible framework for fast prototyping of multiprocessor dataflow applications. Design and Test Workshop, 2008. IDT 2008. 3rd International, 2008, Tunisia. pp.215–220, 10.1109/IDT.2008.4802500 . hal-00398830

HAL Id: hal-00398830
https://hal.science/hal-00398830v1

Submitted on 25 Jun 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AN EXTENSIBLE FRAMEWORK FOR FAST PROTOTYPING OF MULTIPROCESSOR DATAFLOW APPLICATIONS

Jonathan Piat, Mickaël Raulet, Maxime Pelcat, Pengcheng Mu, Olivier Déforges
IETR/INSA, UMR CNRS 6164
Image and Remote Sensing laboratory
F-35043 Rennes, France
firstname.lastname@insa-rennes.fr

***Abstract*—As the number of cores continues to grow in both digital signal and general purpose processors, tools which perform automatic scheduling from model-based designs are of increasing interest. CAL is a new actor/dataflow oriented language that aims at helping the programmer to express the concurrency and parallelism that are very important aspects of embedded system design as we enter in the multicore era. The design framework is composed by the OpenDF simulation platform, by Cal2C and CAL2HDL code generators and by a multiprocessor scheduling tool called PREESM. Yet in this paper, a subset of CAL is used to describe the application such that the application is SDF. This SDF graph is one starting point of the workflow of PREESM (composed of several plug-ins) to be prototyped/distributed/scheduled over an IP-XACT multiprocessor platform description. The PREESM automatic scheduling consists in statically distributing the tasks that constitute an application between available cores in a multi-core architecture in order to minimize the final latency. This problem has been proven to be NP-complete. An IDCT 2D example will be used as test case of the full framework.**

I. INTRODUCTION

As applications such as video coding/decoding or digital communications with advanced features (MIMO, Beamforming, Equalization, etc...) are becoming more complex, the need for computational power is rapidly increasing. In order to satisfy software requirements, the use of parallel architecture is a common answer. To reduce the software development effort for such architectures, it is necessary to provide the programmer with efficient tools capable of automatically solving communications and software partitioning/scheduling concerns.

A methodology called Algorithm-Architecture Matching (AAM) guides the software designer through a design and development process, from application specification to code generation. This methodology uses a graph based representation of the application, following the Synchronous Data Flow graph (SDF) semantic, which is well-suited for signal-oriented applications and is supported by several graph transformations and scheduling methods. This graph is transformed using typical SDF transformation methods and partitioned on a given multiprocessor architecture. A schedule of the partitioned graph for each processor of the architecture is generated.

In order to be able to generate a multi-processor implementation from a CAL network we have chosen to restrict the CAL expressiveness in order to get a statically schedulable CAL that can be transformed and processed using the AAM methodology in PREESM tool.

Section II presents the dataflow networks concepts followed by the CAL actor language in section III. Section IV will explain the CAL supporting tools. Then we will explain the synchronous dataflow graph in section V that our proposed framework takes as an input (section VI). Section VII describes the different plug-in component used by the framework. In section VIII IDCT study case is taken as an example for the proposed method followed by the conclusion IX.

II. DATAFLOW NETWORKS

A dataflow program is defined as a directed graph, where the nodes represent computational units and the arcs represent the flow of data. The lucidness of dataflow graphs can be deceptive. To be able to reason about the effect of the computations performed, the dataflow graph has to be put in the context of a computation model, which defines the semantics of the communication between the nodes. There exists a variety of such models, which makes different tradeoffs between expressiveness and analyzability. Of particular interest are Kahn process networks [1], and synchronous dataflow networks [2]. The latter is more constrained and allows for more compile-time analysis for calculation of static schedules with bounded memory. It has been shown that dataflow models offer a representation that can effectively support the tasks of parallelization [2] -thus providing a practical means of supporting multiprocessor systems. A restricted form of dataflow, known as synchronous dataflow, can be synthesized particularly efficiently, since the computations can be scheduled statically. More general forms of dataflow programs are usually scheduled dynamically, which induces a run-time overhead.

The fundamental entity of this model is an actor [3], also called dataflow actor with firing. Dataflow graphs, called networks, are created by means of connecting the input and output

ports of the actors. Ports are also provided by networks, which means that networks can nested in a hierarchical fashion. Data is produced and consumed as tokens, which could correspond to samples or have a more complex structure. This model has the following properties:

- **Strong encapsulation**:
  Every actor completely encapsulates its own state together with the code that operates on it. No two actors ever share state, which means that an actor cannot directly read or modify another actor's state variables. The only way actors can interact is through streams, directed connections they use to communicate data tokens.
- **Explicit concurrency**:
  A system of actors connected by streams is explicitly concurrent, since every single actor operates independently from other actors in the system, subject to dependencies established by the streams mediating their interactions.
- **Asynchrony untimedness**:
  The description of the actors as well as their interaction does not contain specific real-time constraints (although, of course, implementations may).

III. THE CAL ACTOR LANGUAGE

CAL [4] is a domain-specific language that provides useful abstractions for dataflow programming with actors. CAL has been used in a wide variety of applications and has been compiled to hardware and software implementations [5], [6], and work on mixed HW/SW implementations is under way.

CAL is particularly well suited for describing signal processing systems which are intrinsically data-driven. It is not by chance that CAL language has been chosen by the ISO/IEC standardization organization in the new MPEG standard called "Reconfigurable Video Coding (RVC)" (ISO/IEC 23001-4 and 23002-4). RVC is a framework allowing users to define a multitude of different codecs, by combining together actors (called coding tools in RVC) from the MPEG standard library written in CAL, that contains video technology from all existing MPEG video past standards (i.e. MPEG-2, MPEG-4, etc.) . The reader can refer to [7] for more information about RVC. CAL is used to provide the reference software for all coding tools of the entire library. The essential elements of the RVC framework, besides the tool library, include a Decoder Description expressed in an XML dialect, which describes the architecture of the decoder by specifying the connections between the different actors, a Bitstream Schema which describes the structure, the organization of the data in the bitstream and implicitly defines the parser needed for the specific decoder reconfiguration.

IV. CAL SUPPORT TOOLS

CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses project. Moses features a graphical network editor, and allows the user to monitor actors execution (actor state and token values). The project being no longer maintained, it has been superseded by the Open Dataflow environment (OpenDF for short). Contrarily to Moses, this project does not yet provide a network graphical editor.

OpenDF is also a compilation framework. Today there exists a backend for generation of HDL(VHDL/Verilog) [5], and another backend for that generates C for integration with the SystemC tool chain [6]. A third backend targeting ARM11 and embedded C is under development [8] as part of the EU project ACTORS . It is also possible to simulate CAL models in the Ptolemy II environment.

V. SYNCHRONOUS DATA FLOW GRAPH

A. Principles

A Synchronous Data Flow graph is a finite directed, weighted graph $G =< V, E, d, t, p, c >$ where :

- V is the vertex set of nodes which computes an input data stream and outputs its results.
- $E \subseteq V \times V$ is the edge set, representing channels which carry data streams.
- $d : E \rightarrow N \cup \{0\}$ is a function with $d(e)$ the number of initial tokens on an edge e.
- $t : V \rightarrow N$ is a function with $t(v)$ representing the execution time of node v.
- $p : E \rightarrow N$ is a function with $p(e)$ representing the number of data tokens produced at e's source to be carried by e.
- $c : E \rightarrow N$ is a function with $c(e)$ representing the number of data tokens consumed from e by e's sink node.

The topology matrix is the matrix of size $\lceil E \rceil \times \lceil V \rceil$, in which each row corresponds to an edge e in the graph and each column corresponds to a node v. Each coefficient (i, j) of the matrix is positive and equal to N if N tokens are produced by the j^{th} node on the i^{th} edge. (i, j) coefficients are negative and equal to N if N tokens are consumed by the j^{th} node on the i^{th} edge.

It was proved in [9] that a statical schedule for Graph G can be computed only if its topology matrix's rank is one less than the number of nodes in G. This necessary condition means that there is a Basic Repetition Vector (BRV) q of size $\lceil V \rceil$ in which each coefficient is the repetition factor for the j^{th} vertex of the graph. In the example described in the Figure 1 the given SDF graph is schedulable as the matrix's rank is 3 and the number of vertices 4. SDF graph representation allows use of hierarchy, meaning that for $v = G$, a vertex may be described as a graph. A vertex with no hierarchy is called an actor.

B. Statically Schedulable CAL

In order to have a CAL description which is statically schedulable, user must respect restrictive programming rules. Those rules ensure that every action of single actor consume and produce the same amount of tokens on all the actor interfaces. This rule ensures that producing and consuming

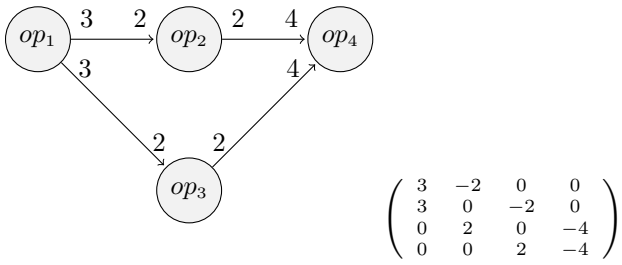

Fig. 1. SDF and topology matrix

rates of actors are known at analysis time. This allows performing balance equation. Thus, actor firing pattern can be extracted and SDF graph can easily be extracted from the network and CAL actors. The structured of the SDF graph is extracted from the XML description of the network (XDF) and the patterns from CAL actors give the token production and consumption rates necessary for the SDF graph. The SDF graph is the skeleton that will give us the multi-processor schedule automatically computed by PREESM. An actor is composed of actions that fulfill the skeleton and are compiled by the code generators (Cal2C, Cal2HDL and Cal2ARM) dependent of the targeted processor.

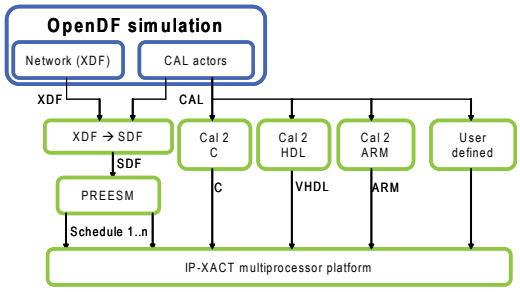

Fig. 2. Workflow from static schedulable CAL

C. PREESM WORKFLOW

VI. PROPOSED FRAMEWORK

A. Framework overview

The proposed framework goes from graph specification using a graph Editor (Graphiti) to code generation using PREESM and a library of proposed plug-in. The Graphiti graph editor allows the user to design application graph using a Synchronous Data Flow (SDF) semantic and/or the CAL network semantic (XDF) (Figure 2). The design can be hierarchical when a vertex in the graph has a graph representation for its behaviour, or atomic when the vertex behaviour is described using programming language as C or CAL. Graphiti can also edit architectures described using IP-XACT language, an IEEE standard from the SPIRIT consortium.

Synchronous Data Flow graphs can then be processed using the PREESM Eclipse plug-in which computes different transformation on the graph and then can match the graph with a given hardware architecture using different mapping algorithms. At the end of the process, the so mapped application is translated into C code, which can be compiled and executed on the hardware target.

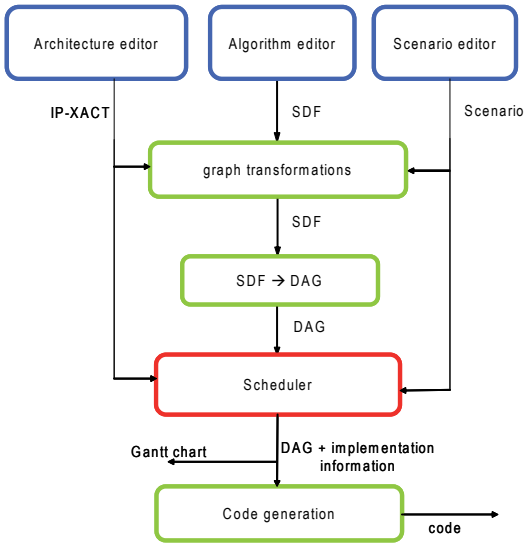

Fig. 3. From SDF and IP-XACT descriptions to code

All the transformation, mapping, code generation algorithms are designed as PREESM plug-ins, which allow user to extend the transformation, mapping, code generation algorithms library. The process, which goes from graph transformation to code generation, is driven by a Workflow graph (also edited in Graphiti) that specifies the order in which the PREESM plug-ins are to be executed and all the data exchange between the plug-ins (Figure 3).

A PREESM workflow is designed as a directed graph with input nodes that are the application graph, the architecture graph, and a scenario. The application graph is the SDF graph designed in Graphiti that describes the application behavior. This graph can be transformed using the transformation plug-in that are designed to take a SDF graph as an input and output a transformed SDF graph. SDF graph can also be mapped using the mapping plug-in that takes as input the architecture, the SDF graph and the scenario.

The architecture is an undirected graph in which each node represents an operator or a communication media. Operators can be connected together through communication medium using their available interfaces. The scenario specifies for SDF graph nodes, the execution time on different kinds of operators that could be instantiated in the architecture graph. The mapping plug-in outputs a Directed Acyclic Graph (DAG) in which nodes have a property that gives the operator in which its computation must be implemented.

This DAG also has send and receive nodes that represent

a transfer between two different operators and for which the communication medium to use is specified.

In order to view the DAG/SDF graph at different steps of the workflow user can place exporter plug-in instance that output the graph in a file in a given format. DAG can then be used to generate code using the code generation plug-in that takes a DAG an input and outputs a set of files (one for each operator) to be compiled for running on a target.

VII. EXISTING PLUG-INS

A. Graph Transformation

Graph transformation plug-in provides algorithms for parallelism extraction in the graph and complexity minimizing.

- **Hierarchy Flattening**:
  A representation of the application using different grain levels (hierarchical representation) may simplify the user's view and allow analysis of the application at different levels. Certain analyses and transformations need to have a fine grain representation of the algorithm in order to extract parallelism or optimize memory. The extraction of the finest grain representation from a hierarchical graph is performed by flattening the hierarchy. After the flattening operation, the graph is composed of actors performing the same algorithm as the hierarchical SDF graph.
- **Homogeneous SDF graph**:
  Another common operation computed on SDF graphs is repetition flattening. Repetition flattening is used to extract the HSDF (Homogeneous SDF) representation of the graph in which producing and consuming rates are the same on each edge (Figure 4). This operation creates copies of each vertex to satisfy the repetition pattern of vertices at the top hierarchy level and then links dependencies between vertices. If there is no delay on the edge, the first consumer vertex consumes the first token produced. If there are N delays on the edge, the N delays are consumed by the N first copies of the vertex after which the vertices then consume the produced tokens.
- **Internalization**:
  Internalization defined in [10] is a process that tends to minimize the application complexity by clustering vertices of the graph. The clustering is done by analyzing the critical path of the SDF graph to determine which vertices can be computed in another cluster considering a cost function.
- **Loop Pipelining**: The loop pipelining process as described in [11] tries to extract a pipeline from the potential loops of the graph. It then constructs a cluster around the pipelines stages. The algorithm tends to reduce the complexity of the applications graph and to maximize the loops iteration domain.

B. Mapper Scheduler

Scheduling algorithms have been the subject of intense study for the past few years. In his PhD thesis [12], Y. K. Kwok presents high-performance algorithms focusing on their complexity and on the analysis capacity to achieve the smallest latency. The model he used to represent the architecture behavior is a basic one: the architecture is homogeneous and communications have a given fixed cost for the transfer between cores. In [13], O. Sinnen analyses the models that could be used to accurately simulate implementation during scheduling. He introduces the idea of routes and edge scheduling to model realistic systems.

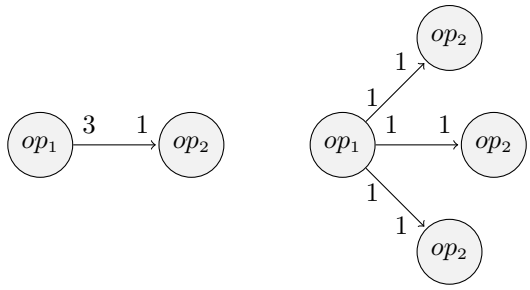

Fig. 4. SDF to HSDF

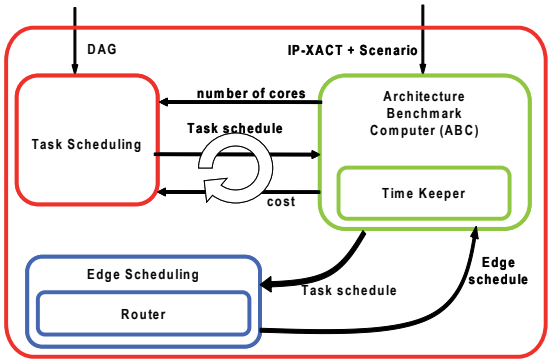

Fig. 5. Scheduler module structure

The PREESM scheduler is divided in three sub-modules which share minimal interfaces: the task scheduling, the edge scheduling and the Architecture Benchmark Computer (ABC) sub-modules (Figure 5). The task scheduling sub-module determines a scheduling solution of the application tasks mapped onto the architecture cores and then questions the ABC sub-module to evaluate the cost of the proposed solution. The advantage of this approach is that any task scheduling heuristic may be combined with any ABC model, leading to many different scheduling possibilities.

The interface offered by the ABC to the task scheduling sub-module is minimal: it gives this sub-module the number of available cores, receives an implementation description and returns costs (infinite if the implementation is impossible). The time keeper calculates and stores timings (such as ASAP, ALAP and so on [12]) for the tasks when necessary for the ABC. This interface contains four methods which return

costs: **getTimeCost(task), getTimeCost(transfer), getTimeCost(core), and getCost(implementation)**.

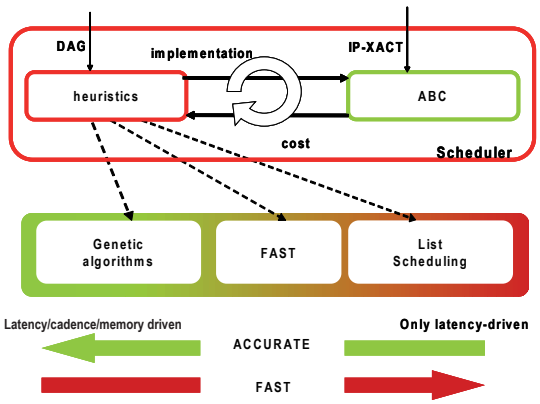

Fig. 6. Switchable scheduling algorithms

The ABC needs to schedule the edges in order to calculate the implementation cost. However, it is not designed to make any implementation choice, thus delegates this task to the edge scheduling sub-module. The router in the edge scheduling sub-module finds potential routes between the available cores. The choice of module structure (Figure 6) was motivated by the behavioral commonality of the majority of scheduling algorithms.

Three algorithms are currently coded; all of which are defined in [12]:

- A list-scheduling algorithm schedules the tasks in the order of a list constructed by calculating a critical path. Once a mapping choice has been made, it will never be modified. This algorithm is quite fast but has limitations due to this last property. List scheduling is used as a starting point for other refinement algorithms.
- The FAST algorithm is a refinement of the list scheduling solution using probabilistic hops. It can run until stopped by the user and keeps the best latency found. The algorithm is multi-threaded to exploit the multi-core parallelism of a computer.
- A genetic algorithm is coded as a refinement of the FAST algorithm. The n best solutions of FAST are used as the base population for the genetic algorithm. Again, the user is free to stop the processing at any time. This algorithm is also multi-threaded. The scheduler generates a graph in DAG format in which vertices are mapped to cores in the architecture and edges correspond to routes mapped on the available media.

C. Code generator

The code generation uses the Algorithm Architecture Matching methodology (AAM, former AAA [14]). It scans the scheduler output DAG graph and generates a source file for each core in the architecture. In this code, a computation thread calls the functions corresponding to the vertices in the DAG while a communication thread handles inter-core communication. The two threads are synchronized with semaphores. The current code generator generates C code for the tri-core TMS320TCI6487 [15] from Texas Instruments. More details on the code generation for this platform and the thread synchronization are given in [16].

VIII. CASE STUDY

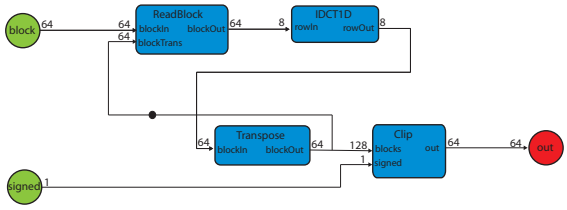

Fig. 7. SDF representation extracted from the CAL network

In order to show the interest of the framework we are going to study an application example starting from a CAL description to automatically generate a bi-processor implementation. The chosen application is the IDCT2D, which is largely used in image/video decoding. The implementation is based on a recursive representation using only four operations (Figure 7):

- ReadBlock : read a block for the first iteration of the graph, transmit the block for the second iteration. thia actor acts as a multiplexer.
- IDCT1D : performs an IDCT on a row of 8 pixels
- Transpose : transposes the block matrix
- Clip: saturate the values to be either stored as 9 bit signed integer or as a 8 bit unsigned integer.

The IDCT2D is then place in a testbed (Figure 8) which generates two blocks on the IDCT2D input and groups the two outputs.

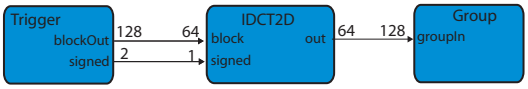

Fig. 8. IDCT2D testbed

This testbed is processed following a PREESM workflow (Figure 9) which performs graph transformation (HSDF, Hierarchy Flattening), and then maps the graph on an architecture composed of two Texas Instrument C64x communicating using EDMA channels. The outputs of the mapping are then used to generate C64x C code for each of the processor, and the Gantt chart of the mapping is displayed.

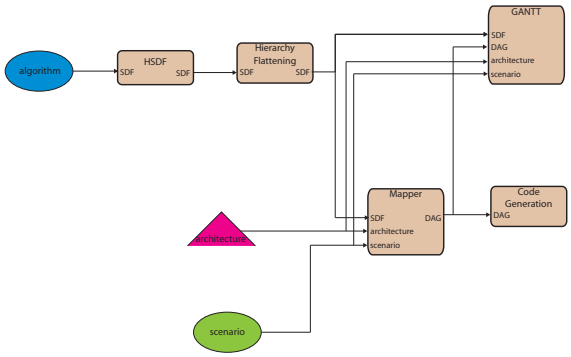

Fig. 9. PREESM Workflow

From the mapping result, each IDCT2D is performed in parallel on a different processor and then results are grouped on a single processor through an EDMA transfer. Benchmarking the result leads to the following performance (Figure 10).

Architecture	Clock cycles
2 C64x + EDMA	5840
C64x	9904

Fig. 10. Performance evaluation for the IDCT2D

Considering those results, the performance gain is 42 %. This benchmark does not take into account the EDMA setup time, which decreases the performance, for small amount of data. To minimize the overhead we consider using clustering algorithm that maximize the throughput by grouping buffers.

IX. CONCLUSION

Using CAL as an input for the framework leads to a decent implementation with C code generation that can target embedded systems. Using CAL2HDL instead of CAL2C to generate some regions of an application (some part of the network) leads to the same kind of result with the ability to target programmable logic [5]. While the PREESM software will evolve, we will be able to use other optimizations in the graph transformations and scheduling that might lead to better implementation either for C generation or HDL generation. The next step in our development is to fully support CAL by extracting SDF region from the network [17] to be processed by the framework. Other improvements could be programming other code generation plug-ins to target other processors such as ARM11, MicroBlaze and PowerPC.

REFERENCES

[1] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proceedings of the IFIP Congress*, J. L. Rosenfeld, Ed., pp. 471–475. North-Holland, New York, NY, 1974.

[2] Edward Ashford Lee and David G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[3] Carl Hewitt, "Viewing control structures as patterns of passing messages.," *Artif. Intell.*, vol. 8, no. 3, pp. 323–364, 1977.

[4] J. Eker and J. W. Janneck, "Cal language report specification of the cal actor language," Tech. Rep. UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003.

[5] Jorn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickael Raulet, "Synthesizing hardware from dataflow programs: An mpeg-4 simple profile decoder case study," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, 2008, pp. 287–292.

[6] Ghislain Roquier, Matthieu Wipliez, Mickael Raulet, Jorn W. Janneck, Ian D. Miller, and David B. Parlour, "Automatic software synthesis of dataflow program: An mpeg-4 simple profile decoder case study," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, 2008, pp. 281–286.

[7] Christophe Lucarz, Marco Mattavelli, Joseph Thomas-Kerr, and Jorn Janneck, "Reconfigurable media coding: A new specification model for multimedia coders," in *Signal Processing Systems, 2007 IEEE Workshop on*, 2007, pp. 481–486.

[8] Carl von Platen and Johan Eker, "Efficient realization of a cal video decoder on a mobile terminal (position paper)," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, 2008, pp. 176–181.

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

[10] V. Sarkar, *Partitioning and scheduling parallel programs for execution on multiprocessors*, Ph.D. thesis, Stanford, CA, USA, 1987.

[11] Liang fang Chao, Andrea Lapaugh, and Edwin Hsing mean Sha, "Rotation scheduling: A loop pipelining algorithm," in *Proc. 30th ACM/IEEE Design Automation Conference*, 1993, pp. 566–572.

[12] Yu-Kwong Kwok, *High-performance algorithms of compile-time scheduling of parallel processors*, Ph.D. thesis, 1997, Adviser-Ishfaq Ahmad.

[13] O. Sinnen, *Task Scheduling for Parallel Systems*, May 2007.

[14] T. Grandpierre and Y. Sorel, "From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[15] Texas Instrument, "Tms320tci6487 dsp platform (sprt405)," Tech. Rep.

[16] "Optimization of auto-matically generated multi-core code for the lte rach-pd algorithm," November 2008.

[17] Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *SCOPES '05: Proceedings of the 2005 workshop on Software and compilers for embedded systems*, New York, NY, USA, 2005, pp. 37–49, ACM.