SCHEDULING OF DATAFLOW MODELS WITHIN THE RECONFIGURABLE VIDEO CODING FRAMEWORK

Jani Boutellier[1], Veeranjaneyulu Sadhanala[2], Christophe Lucarz[3], Philip Brisk[4], Marco Mattavelli[3]

[1] Machine Vision Group, University of Oulu, Finland
[2] Department of Computer Science and Engineering, IIT Bombay, India
[3] Microelectronic Systems Laboratory (GR-LSM), [4] Processor Architecture Laboratory,
École Polytechnique Fédérale de Lausanne, Switzerland

ABSTRACT

The upcoming Reconfigurable Video Coding (RVC) standard from MPEG (ISO/IEC SC29WG11) defines a library of coding tools to specify existing or new compressed video formats and decoders. The coding tool library has been written in a dataflow/actor-oriented language named CAL. Each coding tool can be represented with an extended finite state machine and the dependencies between the tools are described as dataflow graphs. This paper proposes an approach to derive a multiprocessor execution schedule for RVC systems that are comprised of CAL actors. In addition to proposing a scheduling approach for RVC, an extension to the well-known permutation flow shop scheduling problem that enables rapid run-time scheduling of RVC tasks is introduced.

Index Terms— Scheduling, parallel processing, digital signal processors

1. INTRODUCTION

The effort of designing the Reconfigurable Video Coding (RVC) standard [1] is motivated by the intent to describe existing video coding standards with a set of common atomic building blocks (e.g., IDCT). Under RVC, existing video coding standards are described as specific configurations of these atomic blocks. This greatly simplifies the task of designing future multi-standard video decoding applications and devices by allowing software and hardware reuse across video standards.

The RVC coding tools are specified in a dataflow/actor object-oriented language named CAL [2] that describes the atomic blocks in a modular way. Abstract, high-level models require a systematic implementation methodology and tools to realize into practical systems. Design flows are generally composed of several phases: specification, design space exploration (DSE) and implementation.

The implementation phase needs two components: the implementation code (generally C and VHDL) and a schedule. Code generators are under development in the RVC framework [5, 6]. A fundamental step to efficiently complete the implementation phase supported by the code generators is the schedule, i.e. the sequence in which CAL actors fire.

This paper introduces the model of computation used in RVC, which is strongly based on the CAL language model. The RVC scheduling shown in this paper consists of *static* (offline) and *dynamic* (runtime) components. The static schedules are computed at compile-time and are collected in a repository for use by the runtime system, which selects entries from the repository and appends them to the ongoing schedule. The result is similar to a permutation flow shop schedule (PFSS) [3].

2. RELATED WORK

2.1. The Reconfigurable Video Coding framework

The Open Dataflow environment [4] supports the specification, simulation and debugging of CAL models, such as RVC. From CAL models, hardware and software code generators convert CAL actors into implementation languages such as VHDL, Verilog and C [5, 6]. The hardware code generator [5] has the capability to compile networks of actors; the C code generator produces CAL code within atomic blocks, but lacks capability to compile a network of actors. In particular, the C code generator lacks a scheduler that determines the order in which actors fire. A schedule is required to complete the design flow from specification to implementation. At present, there is no multiprocessor scheduling mechanism that can meet the needs of the CAL implementation of the RVC standard.

The Syndex tool [7] maps synchronous data flow (SDF) graphs onto multiprocessor architectures. CAL, unfortunately, is an asynchronous model. This paper shows that the RVC model can be transformed from its asynchronous representation in CAL to a synchronous model accepted by Syndex, enabling multiprocessor scheduling.

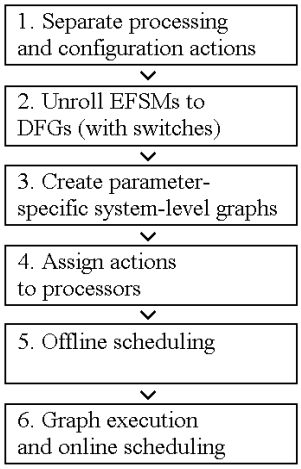

Fig. 1. Overview of our approach.

2.2. Scheduling of similar systems

CAL models are too general to be scheduled efficiently; even straightforward scheduling is intractable. Each CAL actor is represented as an extended finite state machine (EFSM) that contains variables and guard conditions that enable or disable possible state transitions. The relationships between actors are expressed with dataflow graphs (DFGs). Actors communicate by firing tokens along the edges of the dataflow graph. The state transition and token(s) fired by each actor is a function of the current state and the tokens that arrive at its input.

As a partial solution to the scheduling problem, literature suggests that EFSMs can be transformed into regular FSMs, with the cost of a possible state-space explosion [8]. It is then possible to model the system as a combination of dataflow circuits that have actors containing state machines. There are several modeling methods [9, 10, 11] for such problems.

3. THE RVC MODEL

Figure 2 shows a high-level view of the RVC implementation of the MPEG-4 Simple Profile decoder. Figure 1 provides an overview of our approach. The first three steps of our approach deal with the RVC model, and are described in Sections 3.1, 3.2, and 3.3; the other steps, described in Section 4, comprise the scheduling mechanism.

3.1. Processing and configuration actions

In the RVC model, the actions of EFSMs are separated into two classes: *processing (P-actions)* and *configuration (C-actions)*.

A P-action: (1) does not change the state of the EFSM (although, it may modify variables), or (2) can not be reached from the initial state without passing through an action that does not change the EFSM state; all other actions are C-actions. The RVC model imposes one further restriction at the actor interface level: if an actor has an input or an output that has a variable token rate, then all of its internal actions are C-actions.

The actor "add" is used to illustrate the difference between these two classes of actions. "add" combines the predicted image data with the coded prediction error. The inputs and outputs of the actions contained in "add" are listed in Table 1, and Figure 3 shows its EFSM. The *guard* column in the table describes the condition, which must be satisfied so that the state transition can take place; the body column describes the consequences of the state transition affecting only the actor.

The self-loop actions in the EFSM in Figure 3 are *tex*, *mot*, and *comb*; they are processing actions. The three *done* actions are also processing actions, since they cannot be reached from the *cmd* state without passing through *tex*, *mot*, or *comb*. The remaining actions are of the configuration type.

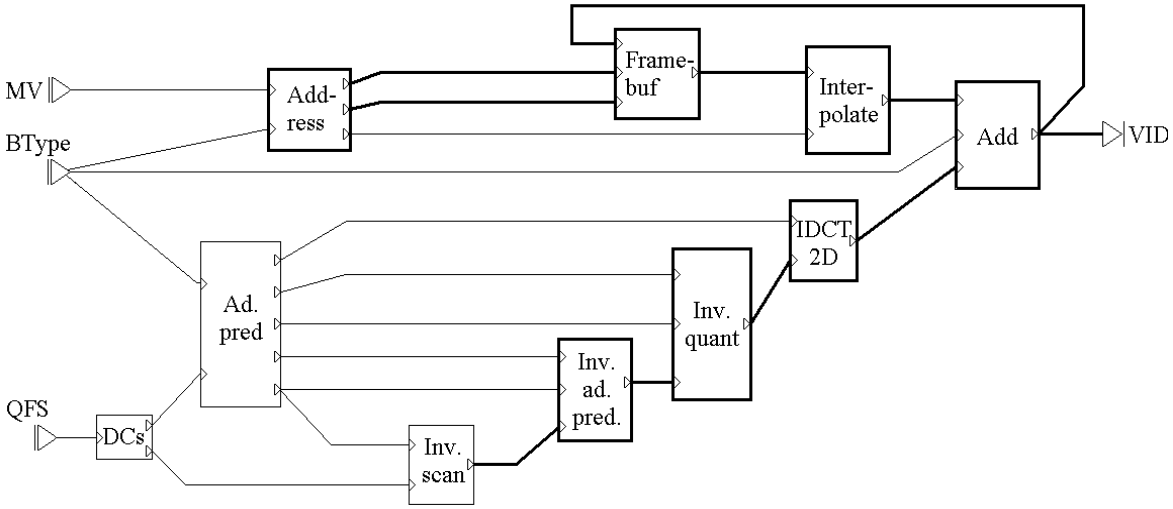

Fig. 2. High level view of the RVC MPEG-4 Simple Profile decoder in the RVC framework.

Also Figure 2 depicts the result of our action classifications, but on a higher level. The actors that have thin outlines only contain configuration actions, whereas the rest of the actors contain both configuration and processing actors. Respectively, signals that are marked in bold are data signals that have no configuration properties and are only used by processing actors.

Table 1. Actions of the "add" actor.

Action: input =>output	Guard	Body
newVop: btype ==> NULL	btype==NEWVOP	NULL
texture: btype ==> NULL	btype == INTRA	NULL
motion: btype ==> NULL	btype!= ACCODED	NULL
other: btype ==> NULL	NULL	NULL
done: NULL ==> NULL	count==64	count=0
tex: TEX ==> VID	NULL	count++
mot: MOT ==> VID	NULL	count++
comb: MOT, TEX ==> VID	NULL	count++

3.2. EFSM unrolling

In the RVC model, the EFSMs can be unrolled into a collection of SDFs that are connected by virtual switching actions, as shown in Figure 4. The unrolling is done so that each iteration variable value has its own action instance. In the RVC model of MPEG-4 Simple Profile, this does not lead to unreasonably large graphs, since the iterations of one action are at limited to at most 81.

In Figure 4, the leftmost element is a virtual *fork* action that shows the different paths of execution that can originate from the initial state. Each transition from *cmd* starts with a configuration action; afterwards, each path of execution, with the exception of the *newVop* path, continues with data processing.

All of the actors in the MPEG-4 Simple Profile do not map trivially into our model. One exception can be found within the hierarchical "IDCT2D" functional unit that contains several actors inside. One of them is named "Clip" and it does clipping of integer values. The problem with the implementation of this actor is that depending on the value of the integer that has to be clipped, a different action is chosen. It was trivial to modify the "Clip" –actor to an implementation which has the same functionality, but within one single action. Generally, such data-dependent actors can not be handled well by static scheduling methods. The "Inverse Scan" actor had to be left outside the static schedule, because of its data-dependent nature.

There are also actors that do not contain a state machine in the CAL code. In this case, an artificial state machine structure is created to make these actors suitable for further processing.

3.3. Parameter-specific system-level graphs

The unrolled RVC model actor-specific graphs contain virtual switches that interconnect multiple SDF fragments, as can be seen in Figure 4. The data processing actions in Figure 4 have the same interfaces as the actions in the CAL code and Figure 2. These interfaces must be connected to the respective interfaces of other actors to create a large SDF processing graph that encompass the whole system.

At the system level, there will be one and only one configuration graph whose topology changes during execution. For each combination of system parameter (switch) values, there will be a unique data processing graph on the system level, called a *subgraph*. For the sake of brevity, we shall call the parameter value combination a *setting* from here on. To model and schedule the complete system, the total number of settings must remain small: otherwise, the number of subgraphs in the system will explode. The MPEG-4 Simple Profile system that is handled

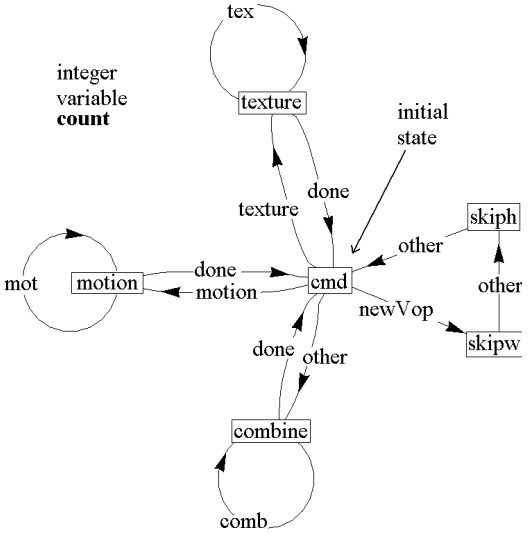

Fig. 3. The EFSM of the "add" actor.

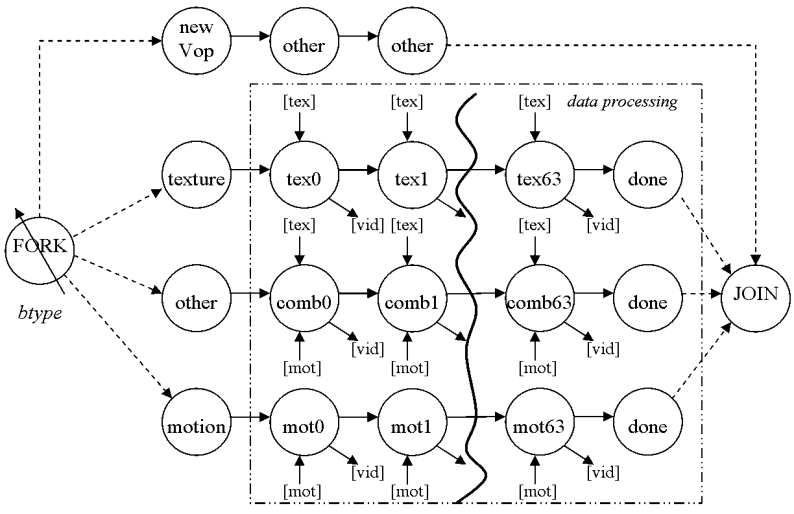

Fig. 4. Combination of SDF graphs derived from "add".

here as an example, produces 4 different subgraphs.

To construct the subgraphs, the processing actions of all actors are grouped according to their corresponding settings; e.g. if the system has 4 settings, each actor must contribute 4 (possibly empty) SDFs. It is possible for two or more *isomorphic* (identical) SDFs to correspond to different settings.

The SDFs corresponding to each setting are connected through their interfaces. For example, in Figure 4, the control path starting with the *texture* action has 64 interfaces of type *[tex]*: the corresponding interface can be found inside the actor "IDCT" depicted in Figure 2. Actions of Figure 4 have also an output named *[vid]* that is not connected to any other graph, but serves as the output of the whole system.

Figure 5 shows a system-wide subgraph that contains all the actions and their dependencies.

4. SCHEDULING

Scheduling for the RVC model is divided into three parts. First, actions are assigned to processors. Second, *offline* schedules are computed at compile-time. An offline schedule is a data processing schedule that exists for each setting. At runtime, an online scheduling and dispatching mechanism executes the configuration actors and selects a schedule for data processing.

The runtime system consists of a set of homogeneous or heterogeneous processors. For now, we assume that the set of processors in the system is fixed (i.e. we do not account for the possibility of a processor failing or the addition of extra processors during runtime).

4.1. Assignment of processors to actions

The first step is to assign each action from each actor to one of the processors in the system. Each action is mapped to exactly one processor; each processor may be responsible for any number of actions. Although it is not mandatory, it is generally advisable to map all instances of one processing action to the same processor. This is not problematic because all the instances of the same processing action are dependent on their preceding instance, i.e. it is impossible to execute multiple instances of the same action in parallel.

4.2. Offline scheduling

Offline scheduling must consider the execution time of each action, which is either assumed to be deterministic, or is the worst-case execution time. This ensures that inter-processor communication, which is determined offline, meets its deadlines.

Configuration actions are executed without explicit scheduling since their number is small and their control flow varies at runtime; the processing graphs, in contrast, are scheduled offline using self-timed scheduling [12], which easily meets our requirements. Self-timed scheduling is convenient because it allows the user to impose inter-processor communication costs and resource sharing, which may be required for some systems.

Self-timed scheduling always produces a static schedule, which can then be invoked at runtime. This is particularly beneficial for MPSoCs, since communication between processors is hard-coded, and thereby eliminates the need for synchronization between processors during data processing.

4.3. Online scheduling

The first step of online scheduling is to execute the configuration actions that are assumed to represent a small proportion of the total action count. Thus, the speed at which their schedules are computed online is more important than the quality of the schedules produced (details

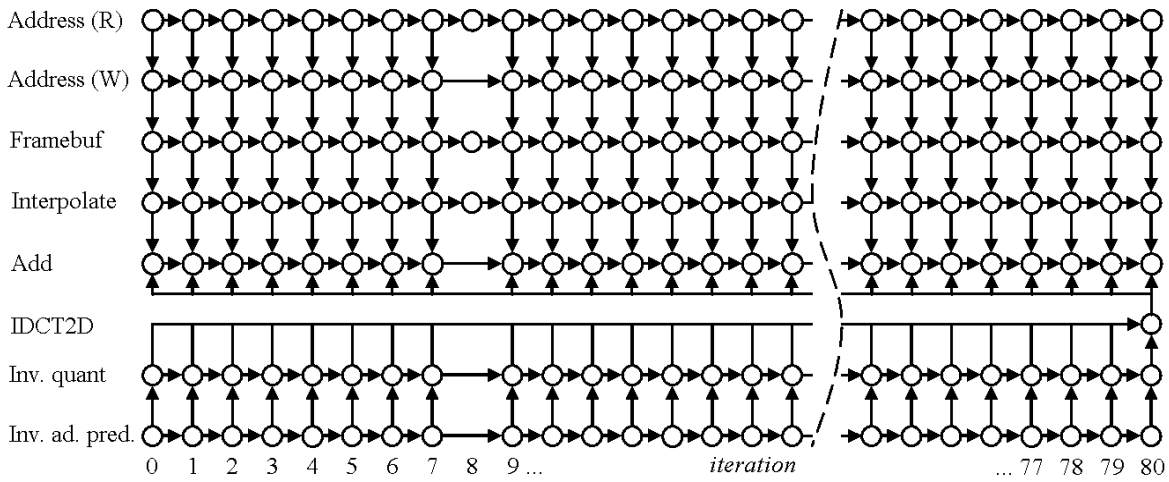

Fig 5. A subgraph corresponding to one specific setting.

are omitted).

The execution of the configuration actions resolves the system setting; based on setting, the corresponding offline processing schedule is selected and dispatched. The new schedule is appended to any previously-dispatched processing schedules that are currently executing; their schedules must not overlap because preemption is not allowed. This model is similar to permutation flow shop scheduling (PFSS) [3], which can be implemented efficiently on a CPU at runtime [13].

4.4. Extended permutation flow shop scheduling

Flow shop scheduling is a specific type of multiprocessor scheduling that has very elegant theoretical properties that make it practical for applications such as RVC. We are given N jobs to schedule on M machines. Each job consists of M tasks, and the j^{th} task in the job *must* be scheduled on machine j. A job can issue its j^{th} task to machine j if the $(j-1)^{st}$ task is complete and machine j is free. Each task is assumed to have a predetermined constant processing time. By definition, each job must have M tasks, one for each machine. However, by setting the execution time of a task to zero, the effect is the same as if that task would not exist. This is called *machine skipping* in literature.

Permutation flow shop scheduling (PFSS) is a more restricted version of flow shop scheduling. Here, task j must be performed for job $n-1$, before it can be performed for job n. Usually the goal in solving the PFSS problem is to find a permutation of jobs which minimizes the makespan (total schedule length in time units). We also assume that this is the objective.

Figure 6 shows the Gantt-chart of a PFSS problem instance consisting of 3 jobs, 3 machines, and 3 tasks per job. Each task within a job executes on a separate machine and *no-wait timetabling* [3] ensures that the next task within the same job starts immediately upon completion of the previous task in the same job. The *inter-job distance* is the overlap in time between two sequential jobs, and is shown for jobs B and C in Figure 6.

Previous research [13] suggests that a particularly efficient implementation of online no-wait PFSS can be made possible by pre-computing the inter-job distances at compile-time and storing them into a look-up table D; this pre-computed information helps the online system efficiently compute a PFSS and dispatch jobs.

The scheduling method applied here is called extended permutation flow shop scheduling (EPFSS). It originates from a computationally efficient implementation of PFSS and has the ability to represent a larger set of scheduling problems. EPFSS extends PFSS in two respects.

The first extension to PFSS is to *enable* dependencies between tasks by modifying the look-up table D. For example, the inter-job distance could be increased so that C1 is forced to start only after B2 finishes. No-wait PFSS cannot represent these types of dependencies.

The second extension *disables* dependencies between tasks. In addition to the inter-job distance look-up table D this requires a *job data* look-up table J, which is used to record the characteristics of all job types. We assume that this table J also contains a value of inter-task distance, which tells the distance between tasks within the same job. In the no-wait timetabling definition this is fixed to zero. In EPFSS we extend this to allow nonnegative inter-task distances. This violates the definition of PFSS, since it is assumed that consecutive tasks are dependent on one another. EPFSS can remove this limitation, if desired.

EPFSS models the scheduling problem for the RVC model because the pre-computed subgraph schedules are precisely of the EPFSS type. The dependencies between subgraph actions executed on different processors are dependent on each other in a fine-grained manner that can not be expressed as a traditional flow shop problem. Figure 7 shows an example of a schedule that can be generated by EPFSS when the second PFSS extension of disabling dependencies is applied. The tasks within one job start with some fixed delay after each other. This delay can be specified arbitrarily in the job data look-up table J. Note that EPFFS scheduling is not a way to improve the makespan of the results: it is a PFSS variation that can model situations which are beyond the expressive power of PFSS. By looking at Figures 6 and 7, one can see that the depicted EPFSS schedule is shorter than the PFSS schedule. This is true, but the figures depict different problems, so the comparison of the makespans is meaningless.

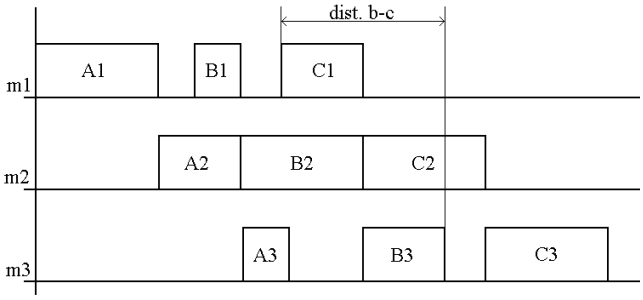

Fig. 6. A conventional PFSS schedule.

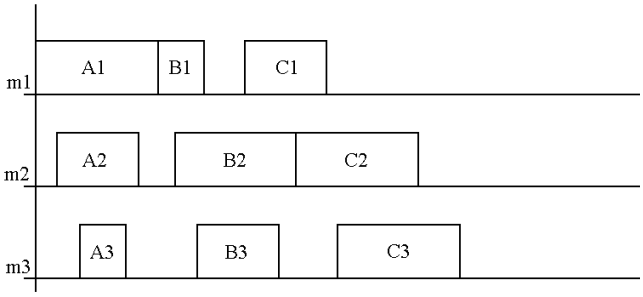

Fig. 7. An EPFSS schedule.

5. EXPERIMENTS

The transformation and offline scheduling steps described in Sections 3 and 4 have been implemented to a great extent in Java and formulated as an OpenDF [4] plugin. In the earlier phases of the transformations, the EFSMs are represented with classes provided by the JGraphT package [15] and during the later stages the SDF graphs are represented with classes from the SDF4J package [16]. All of these steps are performed in the same OpenDF environment as the code generators [5,6] use, which enables smooth interoperability. However, the practical work to enable the use of these schedules for code generation has not yet been started.

The online scheduling method has been implemented in the C language and there is also a yet unpublished hardware implementation of it. However, the online scheduling approach has not yet been incorporated to the OpenDF environment.

6. DISCUSSION

The RVC model and its scheduling mechanism, described in Sections 3 and 4, are not applicable to certain types of systems, for example, those that support control actions *between* data processing actions. The conditions outlined in Section 3.1 imply an algorithm to determine whether a CAL model application meets the same standards as the RVC model; if so, they are compatible with the transformation and scheduling mechanisms described in Sections 3.2, 3.3, and 4.

Future work may generalize the existing RVC model, i.e. extend or otherwise change the classification rules of actions as more complicated video coding algorithms implemented using the RVC standard emerge. Another potential extension of this work is the study of the RVC model as an extension of a more formally established model than CAL, such as parameterized SDF (PSDF) [14].

7. CONCLUSION

This paper describes a sequence of steps to schedule Reconfigurable Video Coding models that are specified as networks of CAL actors. The procedure is based on local and global graph transformations followed by piecewise static multiprocessor scheduling. At runtime, the piecewise static schedules are selected based on the system parameters, and appended to the ongoing processor schedule by means of extended flow shop scheduling. A further step of this work would be to formalize the steps of the procedure and include them into an evolution of the CAL2SW code generation tool.

8. REFERENCES

[1] C. Lucarz et al., "Reconfigurable Media Coding: A New Specification Model for Multimedia Coders," IEEE Workshop on Signal Processing Systems, Shanghai, China: 2007, pp. 481-486.

[2] J. Eker and J.W. Janneck, "CAL Language Report," Tech. Memo UCB/ERL M03/48, UC Berkeley, 2003.

[3] S. French, "Sequencing and Scheduling: an Introduction to the Mathematics of the Job-Shop," Ellis Horwood Ltd, Chichester, 1982.

[4] Open DataFlow Sourceforge Project, *http://opendf.sourceforge.net/*

[5] J. W. Janneck, I. D. Miller, D. B. Parlour, M. Mattavelli, C. Lucarz, M. Wipliez, M. Raulet, and G. Roquier, "Translating Dataflow Programs to Efficient Hardware: an MPEG-4 Simple Profile Decoder Case Study," in Design, Automation and Test in Europe (DATE), Munich, Germany, 2008.

[6] M. Wipliez, G. Roquier, M. Raulet, J.-F. Nezan, and O. Déforges, "Code generation for the MPEG reconfigurable video coding framework: from CAL actions to C functions," in IEEE International Conference on Multimedia & Expo (ICME), Hannover, Germany, 2008.

[7] M. Raulet, M. Babel, O. Déforges, J. Nezan, and Y. Sorel, "Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures," in IEEE Workshop on Signal Processing Systems, 2003, Pages 316 – 321.

[8] O. Henniger and P. Neumann, "Test case generation based on formal specifications in Estelle," in Proceedings of the IEEE International Workshop on Factory Communication Systems, 1995. Pages: 135 - 141.

[9] J. T. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in El Greco," in Proceedings of the Eighth International Workshop Hardware/Software Codesign, 2000. Pages: 142 - 146.

[10] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "FunState-An internal representation for codesign," in Proceedings International Conference on Computer-Aided Design, 1999. Pages: 558 - 565.

[11] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 18(6), Pages 742 - 760, 1999.

[12] S. Sriram and S. S. Bhattacharyya. "Embedded Multiprocessors: Scheduling and Synchronization." Marcel Dekker, Inc., 2000.

[13] J. Boutellier, S. S. Bhattacharyya and O. Silven, "Low-Overhead Run-Time Scheduling for Fine-Grained Acceleration of Signal Processing Systems," Proceedings of the IEEE Workshop on Signal Processing Systems, 2007. Pages: 457 - 462.

[14] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," IEEE Transactions on Signal Processing, 49(10), Pages: 2408 - 2421, October 2001.

[15] JGraph Sourceforge Project, *http://sourceforge.net/projects/jgraph/*

[16] SDF4J Sourceforge Project, *http://sourceforge.net/projects/sdf4j*