HIGH LEVEL DESIGN SPACE EXPLORATION OF RVC CODEC SPECIFICATIONS FOR MULTI-CORE HETEROGENEOUS PLATFORMS

Christophe Lucarz[1], *Ghislain Roquier*[1], *Marco Mattavelli*[1]

[1] Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland
emails:{firstname.lastname}@epfl.ch

ABSTRACT

Nowadays, the design flow of complex signal processing embedded systems starts with a specification of the application by means of a large and sequential program (usually in C/C++). As we are entering in the multi-core era, sequential programs are no longer the most appropriate way to specify algorithms targeted to run on several processing units. The new ISO/MPEG Reconfigurable Video Coding (RVC) standard is proposing a new paradigm for specifying and designing complex signal processing systems. The RVC standard enables specifying new codecs by assembling blocks, or so called Functional Units (FUs) from a standard Video Tool Library (VTL). Flexibility, reusability, and modularity are the key features of RVC. This new way of specifying algorithms clearly simplifies the task of designing future video coding applications by allowing software and hardware reuse across multiple video coding standards. Specifications are provided in the form of an actor and dataflow-based language called CAL. Although the RVC standard does not imply any specific implementation design flow, it is an appropriate starting point for targeting multiple processing units platforms. This paper describes a new model-driven design flow which considers both algorithm and architecture to map RVC codec specifications onto heterogeneous and multi-core systems.

1. INTRODUCTION

Designing and implementing complex digital systems as video decoders on heterogeneous multi-core platforms is a very difficult task. It is even more difficult when such process has to result into implementation with strict requirements on performances and/or resources. Nowadays, the starting point of any traditional design flow is a specification of the algorithm, generally expressed in imperative programming (like C/C++, as used in MPEG). This way of specifying algorithms is essentially sequential, specifying unnecessarily orders of the operations to perform. It also hides the inherent parallelism of the algorithm that is extremely useful while we are now entering the multi-core era. Even if many designs are already taking care to expose parallelism by using threads, it is more and more difficult to guess the behavior of the final application. It becomes clear that any language coming from the sequential paradigm is not appropriate anymore to specify complex signal processing algorithms. A shift towards another paradigm more apt to face the parallelism challenges is clearly necessary.

The CAL Language [1] is a recently specified dataflow and actor-based language capable of concisely expressing complex signal processing algorithms. Such language has very interesting features for describing parallel applications [2]. A subset of the original CAL language has been standardized in the new MPEG Reconfigurable Video Coding framework (RVC) [3, 4] and is used to specify the standard Video Tool Library (VTL).

This paper presents a complete design flow aiming at easing the implementation of complex signal processing systems starting from the dataflow specification used by MPEG RVC. Being such specification characterized by a higher level of abstraction than the one provided by common imperative sequential languages, it is possible to use it when targeting both software and hardware implementation worlds, thus achieving a true unified representation at the level of the specification. When designers develop at low levels of abstractions, any modification of the design can be very resource-consuming since it has to take into account unnecessary low-level details of both software and hardware implementations. Being able to design systems at a high level of abstraction by taking into account implementation constraints (with a correspondence to the low levels of abstractions) and by detecting bottlenecks at the early stages of the design process is a clear advantage of time and resources in the design of any (complex) processing system.

Section 2 explains the implications of the use of the CAL language in terms of implementation. Section 3 describes the steps of the design flow. Section 4 describes

This work is part of the ACTORS European Project (Adaptivity and Control of Resources in Embedded Systems), funded in part by the European Unions Seventh Framework Programme. Grant agreement no 216586

the supporting tools. Section 6 presents a case study, the MPEG-4 Simple Profile decoder as defined within the RVC standard library. Section 7 discusses the advantages and drawbacks of the described methodology and outlines perspectives of future work. Section 8 concludes the paper.

2. WHAT THE IMPLEMENTATION OF CAL PROGRAMS IMPLIES?

The raise of the abstraction level, possible by the use of CAL presents several advantages in the design of signal processing systems in terms of design productivity, low-level implementation details are not taken into account and only the architecture of the dataflow processing is considered at CAL level. However, the fact that the design is done at high level is not a guarantee that it results into an efficient implementation, but requires an appropriate approach to the CAL dataflow design.

Inter-actors communication channels are implemented as finite size FIFO, potentially introducing performance limitations in case of concurrent memory accesses. An actor is in general constituted by a set of actions that may fire according to the current state of the actor, to the availability of tokens and to the satisfaction of firing conditions (guards). If several actors are mapped on the same processing unit, several actions may be fireable at the same time. Thus, a scheduling policy must be defined to select the action that may be fired next.

An appropriate design willing to achieve efficient implementations starting from CAL dataflow programs need to minimize the unnecessary overhead introduced by such control mechanisms such as scheduling of actions and FIFO accesses, implied by the dataflow paradigm on which the CAL language is built. This is the objective of the design flow and the tools that implement it.

3. DESIGN FLOW

The most interesting feature of a dataflow based design flow is that there is a close correspondence between the high level specification and its actual implementation. Token flows corresponds to bandwidths, partitions of a CAL networks correspond to mappings on architectural components and so on. Relying on such correspondence, a design flow alternating design phases and evaluation phases might be able to detect and correct bottlenecks while remaining at high levels of abstractions. Figure 1 illustrates the main steps of the design flow:

- The preliminary step (*Sequential to Parallel transformation*) aims at **building a first CAL program** from any specification. It results in an architecture-agnostic CAL program which exposes the inherent parallelism and the data flow structure of the algorithm.
- The design loop (composed of the *characterization*, *profiling*, *partitioning and scheduling* steps) is responsible for **building an appropriate CAL program** according to a specific target platform (thick red arrows).
- The evaluation loop is responsible for **evaluating the design**, given a target platform (thin blue arrows).
- After each evaluation, the designer has the choice to refactor the CAL program (*CAL Transformations*) or stop the refactoring iterations for the final implementation. Automatic hardware and software synthesis tools are used to generate the platform specific code including the computed partitioning and scheduling.

Many possible optimization criteria can be used to drive the refactoring. However performance maximization and resource minimization combined in different trade-offs are the most common.

3.1. Sequential to Parallel program transformation

The first step is to write a specification in the form of a CAL program. If the starting point is sequential imperative program, extracting metrics such as the computational load of functions, the critical path of functions, the amount of data transfers between functions might be valuable information for building a dataflow program. During this process, the designer could be supported by various profiling tools that outline where the complexity of the algorithm is in order to design an efficient first version of the CAL program.

3.2. Characterization

Once the first version of the CAL dataflow program is built, the designer can characterize it by running static and dynamic (trace-based and simulation-based) analyzes to extract measures of: the nature of the actors (SDF [5], CSDF [6] or DDF [7]), computational load of each actor/action, dependencies between actors, data exchanges between actions, number of executions of actions, the critical path of the execution of the program.

3.3. Profiling

Such profiling step consists in determining the execution times of actions according to the considered underlying

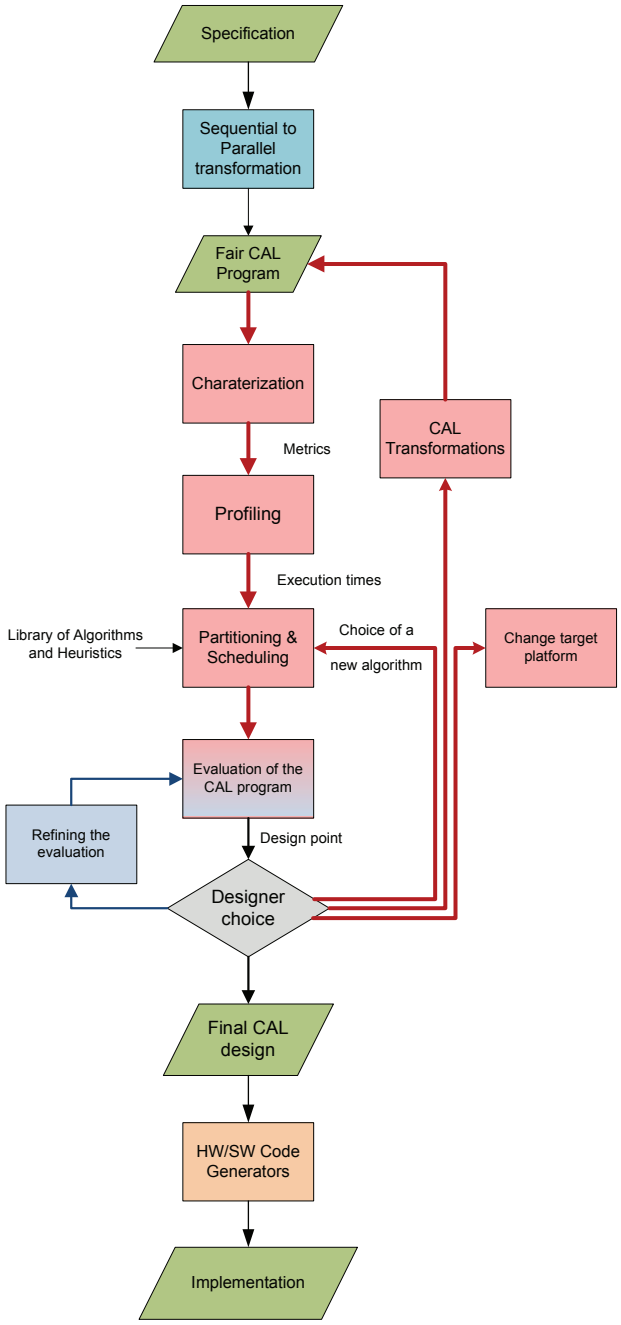


Fig. 1: Design Flow: interlacing of a design and an evaluation loops.

architecture. The specificities of the target platform can taken into account in order to obtain a good approximation of the execution time of the actions. For example, in the ARM 7500FE processor, a General Purpose Processor (GPP), the multiply instructions take one instruction fetch and m internal cycles, m being the number of cycles required by the multiply algorithm, which is determined by the contents of the registers. In Digital Signal Processors (DSP), a Multiply-Accumulate operation costs only one clock cycle. This step is very important because the partitioning and scheduling steps are based on the values of the computational load of the actions.

3.4. Partitioning and scheduling

The partitioning consists of mapping a network of actors onto processing units. Scheduling consists in determining the policy for execution of actions partitioned in the same processing units. The problem of optimally partitioning and the scheduling CAL programs is obviously a NP-hard combinatorial problems and efficient heuristics are currently being studied by several research groups [8] [9].

3.5. Evaluation

Detecting bottlenecks and modifying the architecture of a system at high abstraction level (i.e. CAL level) is advantageous in terms of productivity. It is more problematic and resource consuming when done at low levels of abstractions (e.g. in C for software and in VHDL for hardware implementations). This is the most attractive feature of CAL dataflow design flow. However, in order to make it practically possible, it is necessary that the evaluation step validating the behavior of the refactored CAL program onto the target platform is performed without low level manual rewriting and results should be sufficiently reliable for driving the appropriate dataflow architecture changes. Since there is a correspondence between specific architecture-dependent implementation components and elements of the CAL program, the designer can evaluate the results of successive refactoring and consequent architecture changes in several iterations, thus isolating and evaluating the effect of each single change on the implementation.

3.6. CAL refactoring

The designer has essentially two refactoring possibilities at the level of actors: splitting or merging, resulting in increasing or decreasing the explicit level of parallelism. Another level of refactoring is at level of actions that may consume/generate tokens at coarser or finer levels.

3.7. Code generation

The most attractive feature of CAL dataflow based design is the existence of synthesis that can generate C/C++/LLVM/Java or VHDL/Verilog implementations from CAL programs. Software (C language) and Hardware (VHDL/Verilog) code generators are described respectively in [10] [11] and [12].

4. TOOLS

This section describes in more details the tools that support the steps of the design flow described in the previous section. The tools are classified into two categories : the *characterization tools* which aim at extracting all the properties of the CAL program and the *exploration tools* which aim at exploring the design space. All these tools are integrated in an environment called CAL Design Suite [13] which is open-sourced on Sourceforge and has been developed in Java.

4.1. Characterization Tools

StatiCAL performs static analyzes of the CAL code. It extracts information such as the structure of the program, the size of the ports. **ProfiCAL** determines the execution times of actions by means of a dynamic analysis of the CAL program given an input stimulus. It records the computational load of the actions composing the program and measures executions times. **TraciGen** records an execution of the CAL program in a graph representation (causation trace) in which each node of the graph represents a firing of an action and arcs represent dependencies between action executions. **TraciCAL** analyzes the causation trace and extracts statistics on the execution. **CrossCAL** generates new metrics (e.g. data transfers between actions, code coverage rate) by crossing metrics with each others.

4.2. Exploration Tools

SchedulCAL computes partitioning and scheduling configurations for the CAL program according to different optimization criteria. For example, load balancing, aiming at sharing equally the computations on the different processors, has been implemented in the environment. Algorithms aiming at maximizing the throughput are currently under development. **EvalCAL** evaluates the resulting refactored CAL program, which has been scheduled and partitioned. **AnalytiCAL** displays all the results obtained with the characterization and exploration tools.

5. DESIGN SPACE EXPLORATION

Designing highly complex digital streaming system does not result into a single solution. The design space representation provides a mean to visualize graphically and to compare different designs according to given criteria (e.g. performance, resources) in order to evaluate the efficiency of the current design.

Axes System design is guided by constraints : performance, resources, size, cost, etc. Designers try to optimize the design according to the chosen criteria. Thus, the design space can be represented according to these optimization criteria. The number of criteria defines the dimension of the design space, e.g. respectively 2D or 3D if two or three criteria are considered and so on. Each criterion represents an axis of the design space representation. Usually, throughput and resources are the criteria that define the axis of the 2D design space representation. It results in the representation illustrated in figure 2 with *Throughput* as criteria 1 and *Resources* as criteria 2.

Target Region Depending on the maximization, minimization or lower/upper bound conditions, specific regions (or volumes) of interests can be defined. The target region is defined according to the selected optimization criteria as illustrated in figures 2 (2D design space).

Point Each point in the design space represents one triplet: a CAL program, a Schedule, a Partition. Obviously, it is not possible to evaluate a design if it is not completely characterized. The partition is a one-to-one correspondence between actors and processing elements. The schedules represent the ordering of actions onto each processing unit.

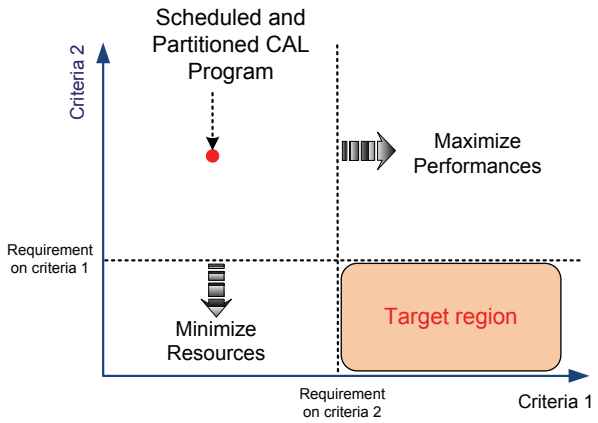


Fig. 2: Representation of the design space.

Starting from a given Scheduled and Partitioned CAL Program (SPCP) located in the design space, the designer has several possibilities to move towards the target region:

1. Refactor the CAL program, and then find again a new partitioning and scheduling
2. Apply a new partitioning and scheduling to the CAL program

3. Apply a new scheduling to the CAL program, keeping the same partitions

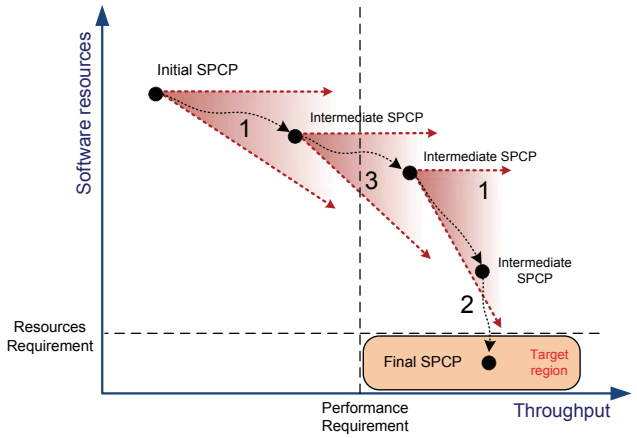


Fig. 3: Exploring the design space with different transformations.

Figure 3 presents an example of exploration of the design space.

Starting from an initial CAL program, a CAL transformation (1) leads to an intermediate CAL program which results in a better throughput with lower resource usage. Another scheduling (3) or partitioning (2) applied to this intermediate model lead to another location in the design space.

6. CASE STUDY: RVC MPEG-4 SP DECODER

The described design flow has been applied for the design case of the RVC MPEG-4 Simple Profile decoder. The CAL program has been written starting from the C/C++ reference software given by the MPEG specification. The example illustrates the design space exploration aiming at achieving higher performances in terms of throughput, while using the minimum number of processors. The steps of the design flow are described in more details in the following paragraphs.

Characterization This first step in the design flow aims at profiling the CAL network. Static and dynamic analyzes are performed to extract metrics such as the computational load of each actor, the data transfers between actors, the number of dynamic calls of actors and actions, the nature of the actors, the critical path of the execution.

Weighting This step consists in assigning execution times to actions of the CAL program. The characterization step provides the computational load of actions. As a first step, the computational load of actions are converted into executions times on target processors in terms of clock cycles.

Partitioning/Scheduling This step outputs the assignment of actors to processing units and the scheduling of actions onto each processing unit. The tool predicts the behavior of the whole system given the execution time of each action in clock cycles, the partitioning of actors and the scheduling of actions. The tool estimates the performances in terms of throughput of the system. The algorithm used for the partitioning and the scheduling is based on a simulated annealing approach [1]. This step is applied by considering 2, 4, 8, 16, 32 and 64 processors.

Evaluation The makespan is the metric used to evaluate the performance of the design. It corresponds to the number of clock cycles necessary for the decoder to decode the input bitstream. This step is applied by considering 2, 4, 8, 16, 32 and 64 processors. The resulting makespans before the optimization are reported in the table presented in section 7.

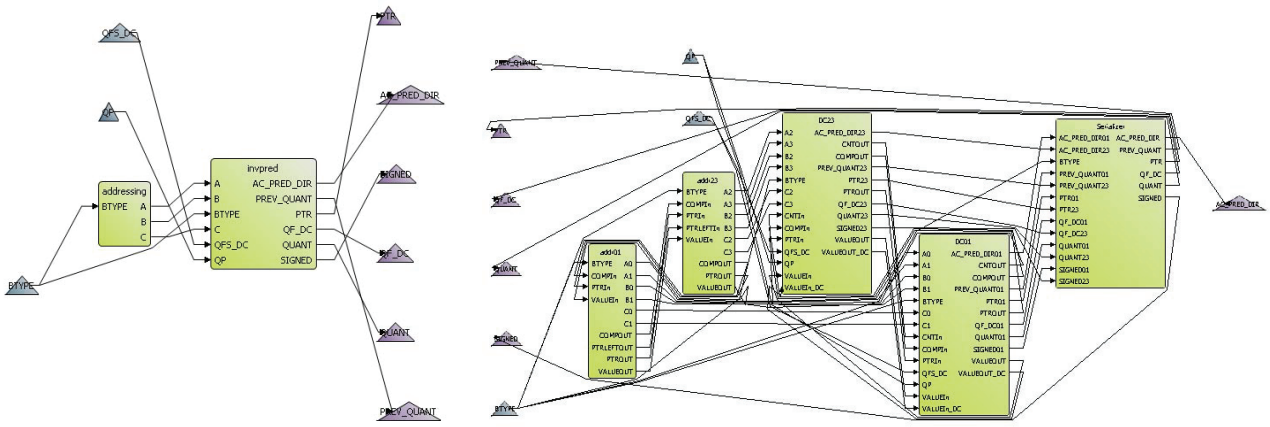
CAL Transformations Designers have several possibilities to increase the performances of a system. as mentioned in 3.5. The designer can refactor part of the CAL code and/or can apply different partitioning / scheduling heuristic. In large CAL programs (such as the one studied in this case study, i.e. 324 actions shared out in 63 actors), it may be difficult to determine which actions/actor refactoring provides the highest potential gains without appropriate metrics and corresponding measures. For this reason, an algorithm has been developed to detect the most interesting actions to optimize.

The algorithm is based on the causation trace and on the measure of the length of the critical path. A causation trace of a dataflow program is a directed acyclic graph such that:

- every node is a firing of an action of an actor in the program,
- every edge from *v1* to *v2* is a dependency (either through a token, state or port) from *v2* on *v1*, implying that therefore $v1$ has to be executed before *v2*.

The critical path is the shortest weighted path from the source node of the causation trace to its sink node,

[1]This algorithm has been developed by Martin Niemeier and Andreas Karrenbauer, from the Chair of Discrete Optimization (DISOPT, EPFL) http://disopt.epfl.ch/



(a) Before (b) After

Fig. 4: Refactoring of the DC Reconstruction network.

the weights being the execution time of actions. The critical path of the execution is provided by the CrossCAL tool and the causation trace by TraciGen (see section 4). The granularity of the critical path is at the action level. Correlating the computational load of actions and their contribution to the critical path provides the bottlenecks (i.e. actions) of the system on which design efforts (i.e dataflow program refactoring) must be focused. This algorithm applied on the RVC MPEG-4 Simple Profile decoder indicates as optimizations with the highest potential of throughput improvement:

1. 30 % of action `copy` of the actor `IAP` (Inverse AC Prediction)
2. 10 % of action `ac` of the actor `IQ` (Inverse Quantization)
3. 10 % more of action `copy` of the actor `IAP` (Inverse AC Prediction)
4. 10 % of action `read_write` of actor `IS` (Inverse Scan)

To implement optimizations to the listed critical actions, i.e. reducing the execution time of these actions, the designer has several possibilities: rewrite the action body (atomic and purely sequential) in order to optimize its sequence of operations; partition the actor into several ones in order to expose more explicit parallelism and to reduce the critical path, i.e. sequential part of the actor.

Weighting This step consists in assigning execution times to actions. In order to figure out the potential improvement of the speed of the system thanks to these optimizations, new execution times are set to actions. These execution times are computed according to the optimizations output by the algorithm described in last paragraph: execution time of action `IAP/Copy` is reduced of 40 %, `IQ/ac` of 10 % and `IS/read_write` of 10 %.

Partitioning/Scheduling The same simulated annealing algorithm is used in order to compute new partitioning and scheduling after the optimization of the actions. This step is applied by considering 2, 4, 8, 16, 32 and 64 processors.

Evaluation This step is applied by considering 2, 4, 8, 16, 32 and 64 processors. The resulting makespans after the optimization are reported in table presented in section 7.

6.1. Discussion of the results

This case study shows an example of design space exploration, showing the main conceptual steps of the design flow, even if all the possibilities and functionality of the tools supporting the design flow are not used in this example. Figure 5 reports the results in form of a graph in which the percentages values represent the obtained improvement of the makespan after optimization for each number of processors. Thanks to the established correspondence between the CAL level and the low level, optimizing the program at the high level in terms of makespan will automatically result in a faster design after implementation. The advantage of working at high level and having the guarantee that the improvements at the high level impact directly on the implementation, is the real gain of time during the design process.

Table 1 reports the makespans obtained before and after optimization, considering the mapping onto 1, 2, 4, 8, 16, 32 and 64 processors respectively.

It may be argued why an optimization of more than 30 % of the makespan results only in an improvement of

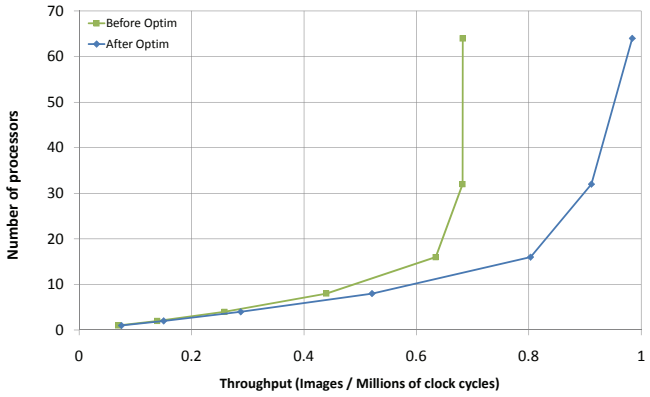


Fig. 5: Exploration of the design space.

Nb of proc.	Before opt.	After opt.	Speedup
1	28 759 336	26 570 383	7.6 %
2	14 422 202	13 301 576	7.8 %
4	7 734 369	6 957 486	10.0 %
8	4 549 236	3 838 300	15.6 %
16	3 152 493	2 489 814	21.0 %
32	2 931 854	2 194 023	25.2 %
64	2 930 104	2 032 209	30.6 %

Table 1: Comparison of results before and after optimization.

15.6 % when considering eight processors. The critical path can be considered as being the makespan of the full execution of the program on a platform in which there are as many processing units as actors in the program. Thus, instead of considering 63 processing units, we consider only four, then all the tasks that were not of the critical path must be packed on these eight processors, resulting in a longer makespan, and reducing the potential impact of the optimization of the critical path.

An interesting result is that the optimization of the critical path has a higher impact when considering the mapping on a larger number of processors. It is coherent with the fact that the algorithm which detects bottlenecks is somehow considering a system constituted with as many processors as actors. Thus, increasing the number of considered processors is making the design case closer to the one considered in the algorithm.

6.1.1. Analysis of the bottlenecks

The algorithm for detecting bottlenecks by means of the analysis of the critical path indicates that the actor "Inverse AC Prediction" (IAP) in the Y channel is the next bottleneck point. One can see that the 8x8 blocks inside a Y macroblock are processed sequentially by this actor. Thus, this constitutes a serialization point that can be parallelized.

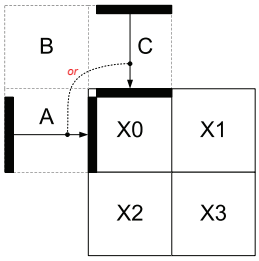


Fig. 6: Principle of the AC prediction.

Figure 6 illustrates the prediction process and the underlying dependencies between blocks. Currently, the IAP actor processes the four blocks of the Y macroblock sequentially. It can be parallelized such that block `X0` is processed first, then blocks {`X1`, `X2`} and finally block `X3`. There are dependencies between blocks {`X1`,`X2`} and block `X0` and between `X3` and {`X1`,`X2`}. Figures 7 and 8 illustrates how the IAP actor can be parallelized.

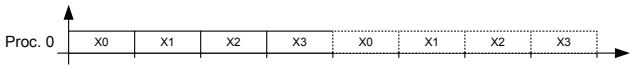


Fig. 7: Current sequential processing of the IAP actor.

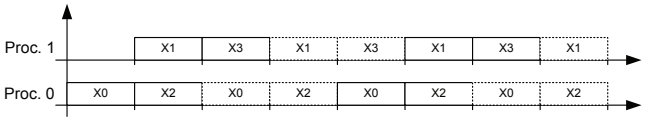


Fig. 8: Potential parallel processing of the IAP actor.

The actor Inverse AC Prediction can be potentially optimized by 50 % because instead of outputting four blocks in a given amount of time, the refactored actor outputs twice more.

7. GENERAL DISCUSSION AND FUTURE WORK

The case study presented in this paper is a good illustration of the attractive features of this design flow based on CAL language, namely the ability to explore the design space at a high level of abstraction by testing different partitioning and scheduling policies without rewriting the whole low level implementation code. Furthermore, the rise of the level of abstraction with CAL language enables designers to have a unified representation for hardware and software. It allows designers to focus more on the design of the application instead of spending time trying to deal with low-level implementation details. Consequently, designers can concentrate more on how to find more efficient partitions of the system, how the platform can exploit the available parallelism

explicitly exposed in the CAL program, how to design the application so that it allows reusability. The proof of concept has been successfully done and the proposed design flow is promising.

Concerning future work, efforts must now be focused on raising additional implementation details to upper levels (i.e. at CAL level), like communication costs between processing units. Unlike in imperative programming (e.g. C/C++), it is straightforward to include communication costs because in CAL, actors exchange data between each others through FIFO buffers. The aim of taking into account these implementation details is to lower the level of abstraction of the evaluation and to refine it in order to get closer to the execution model. Considering communications costs will improve undeniably the evaluation process. Furthermore, there are still other possible improvements of the evaluation process: memory access time, scheduling overhead and others.

8. CONCLUSION

This paper presented a new model-based design flow for complex embedded systems. The originality of our work resides in the use of a language capable of unifying the hardware and software worlds under a unique representation, of expressing both architectural and algorithmic concepts, of composing a design in a modular way, of partitioning straightforwardly complex programs, to design heterogeneous multi-core systems. The design flow has been successfully applied to the MPEG-4 SP decoder as described by the RVC specification. The case study is the proof of concept that the designer can explore the design space without rewriting the program. This work provides a complete methodology to designers to implement decoders defined by the new ISO/MPEG RVC standard. By raising the level of abstraction, designers can build more efficiently embedded systems.

9. REFERENCES

[1] J. Eker and J. Janneck, "CAL Language Report," Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, Dec. 2003.

[2] Christophe Lucarz, Marco Mattavelli, Matthieu Wipliez, Ghislain Roquier, Mickael Raulet, J+rn W. Janneck, Ian D. Miller, and Dave B. Parlour, "Dataflow/Actor-Oriented language for the design of complex signal processing systems," in *Workshop on Design and Architectures for Signal and Image Processing (DASIP)*, Bruxelles, Belgium, Nov. 2008, pp. 168–175.

[3] Lucarz Christophe, Ihab Amer, and Marco Mattavelli, "Reconfigurable video coding : Objectives and technologies," in *IEEE International Conference on Image Processing*, Cairo, Egypt, Nov. 2009.

[4] Shuvra S. Bhattacharyya, Johan Eker, Jorn Janneck, Christophe Lucarz, Marco Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *Journal of Signal Processing Systems*, 2009.

[5] Edward A. 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.

[6] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete, "Cyclo-static dataflow," *IEEE transactions on signal processing*, vol. 44, no. 2, pp. 397–408, 1996.

[7] Joseph T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Ph.D. thesis, EECS Department, University of California, Berkeley, 1993.

[8] Jani Boutellier, Christophe Lucarz, Sbastien Lafond, Victor Gomez, and Marco Mattavelli, "Quasi-Static scheduling of CAL actor networks for reconfigurable video coding," *Journal of Signal Processing Systems*, 2009.

[9] Ruirui Gu, Jorn W. Janneck, Mickael Raulet, and Shuvra S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, vol. 0, pp. 565–568, 2009.

[10] Ghislain Roquier, Matthieu Wipliez, Mickal 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 *IEEE Workshop on Signal Processing Systems (SiPS 2008), Washington, D.C., USA*, 2008, pp. 281–286.

[11] "The Open RVC-CAL Compiler Sourceforge open source project," http://sourceforge.net/projects/orcc/.

[12] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and J.-F. Nezan, "Synthesizing hardware from dataflow programs," *Journal of Signal Processing Systems*, 2009.

[13] "The CAL Design Suite Sourceforge open source project," http://sourceforge.net/projects/caldesignsuite/.