CAL Methodology

Responsible: Ecole Polytechnique Fédérale de Lausanne (EPFL)

Christophe Lucarz (EPFL), Pascal Faure (Akatech SA), Ghislain Roquier (EPFL), Marco Mattavelli (EPFL), Vincent Noël (Akatech SA)
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>3</td>
</tr>
<tr>
<td>1 Abstract</td>
<td>5</td>
</tr>
<tr>
<td>2 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>1 State of the Art</td>
<td>7</td>
</tr>
<tr>
<td>2 Introduction to the CAL language</td>
<td>7</td>
</tr>
<tr>
<td>3 Definition of terms</td>
<td>9</td>
</tr>
<tr>
<td>4 Preliminary notions</td>
<td>10</td>
</tr>
<tr>
<td>4.1 Parallelism taxonomy</td>
<td>10</td>
</tr>
<tr>
<td>4.2 Nature of Actors</td>
<td>10</td>
</tr>
<tr>
<td>5 What the implementation of CAL programs implies?</td>
<td>12</td>
</tr>
<tr>
<td>3 How to design the right CAL program?</td>
<td>13</td>
</tr>
<tr>
<td>1 How to reduce sequentiality?</td>
<td>14</td>
</tr>
<tr>
<td>1.1 Lower bound</td>
<td>15</td>
</tr>
<tr>
<td>2 How to extract parallelism?</td>
<td>15</td>
</tr>
<tr>
<td>2.1 Functional decomposition</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Data decomposition</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Lower bound</td>
<td>17</td>
</tr>
<tr>
<td>4 Target Platforms and Optimization criteria</td>
<td>19</td>
</tr>
<tr>
<td>1 Target platforms</td>
<td>19</td>
</tr>
<tr>
<td>2 Optimization criteria</td>
<td>19</td>
</tr>
<tr>
<td>2.1 Performance maximization</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Minimization of resources with a lower bound on the performance</td>
<td>20</td>
</tr>
<tr>
<td>5 Design space representation</td>
<td>21</td>
</tr>
<tr>
<td>1 How to represent the design space?</td>
<td>21</td>
</tr>
<tr>
<td>2 Localization of a design point in the design space</td>
<td>22</td>
</tr>
<tr>
<td>3 Several types of evaluation</td>
<td>23</td>
</tr>
<tr>
<td>6 How to explore the design space?</td>
<td>25</td>
</tr>
<tr>
<td>1 CAL program transformations</td>
<td>25</td>
</tr>
<tr>
<td>1.1 Splitting of actors</td>
<td>26</td>
</tr>
<tr>
<td>1.2 Merging of actors</td>
<td>26</td>
</tr>
<tr>
<td>1.3 Splitting of tokens</td>
<td>26</td>
</tr>
<tr>
<td>1.4 Vectorization of tokens</td>
<td>27</td>
</tr>
<tr>
<td>2 Partitioning</td>
<td>28</td>
</tr>
<tr>
<td>3 Scheduling</td>
<td>28</td>
</tr>
<tr>
<td>3.1 Synchronous Data Flow</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Dynamic Data Flow</td>
<td>32</td>
</tr>
</tbody>
</table>
7 Design Flow
1 Sequential to Parallel transformation ........................................ 37
2 Characterization of the CAL program .................................... 39
   2.1 CAL level measures .................................................. 39
   2.2 At lower levels of abstraction ....................................... 40
3 Weighting measures ......................................................... 40
4 Partitioning and Scheduling ................................................. 40
5 Evaluation of the design point .............................................. 41
   5.1 Determining the feasibility ........................................... 42
   5.2 Determining execution time of actions assigned to software ...... 42
   5.3 Determining execution time of actions assigned to hardware .... 43
   5.4 Construction of the Gantt Chart ..................................... 43
6 Designer Choice ............................................................. 43
7 CAL program transformations ............................................... 44
8 Code Generation ............................................................. 44

8 Implementation of the design flow ...................................... 47

9 Case Study: Inverse Discrete Cosine Transform ......................... 51
1 Building the right CAL program .......................................... 51
   1.1 Critical path analysis .................................................. 51
   1.2 Data decomposition .................................................... 52
2 Analysis: first iteration of the design loop ............................ 54
   2.1 Characterization ....................................................... 54
   2.2 Weighting of the measures .......................................... 55
   2.3 Partitioning and scheduling ....................................... 55
   2.4 Evaluation .............................................................. 55
3 First optimization: feedback loop ........................................ 55
   3.1 Refactoring: redesign of the program ............................ 55
   3.2 Characterization ....................................................... 56
   3.3 Weighting of the measures .......................................... 56
   3.4 Partitioning and scheduling ....................................... 56
   3.5 Evaluation .............................................................. 56
4 Second optimization: feedback loop .................................... 57
   4.1 Refactoring: apply new partitioning algorithm ................. 57
   4.2 Evaluation .............................................................. 57
5 Synthesis of the CAL program ............................................ 57

10 Conclusion .................................................................. 61

11 Appendixes ................................................................ 63
1 Example of static, cyclo-static and dynamic Actors .................. 63
   1.1 Static actor .............................................................. 63
   1.2 Cyclo-static actor ....................................................... 63
   1.3 Dynamic actor .......................................................... 65
2 Examples of Design Space Exploration applied to target platforms . 68
   2.1 Single sequential processor platform ............................ 68
   2.2 Single FPGA platform ................................................ 69
   2.3 One processor and one FPGA platform ......................... 72
3 Results of the case study .................................................. 75

Bibliography ................................................................. 77
Chapter 1

Abstract

This deliverable addresses and describes concepts, methodologies and tools that are studied by ACTORS WP1 (Task 1.1) for the development of appropriate CAL programs. Developing appropriate CAL programs means either developing a CAL program from any imperative sequential code such as C/C++, or developing a CAL program that better matches a specific implementation scenario.

This deliverable intends to answer some fundamental questions:

1. How can a classical sequential algorithm be rewritten into a parallel CAL program?
2. How to reach a fair CAL program?
3. Starting from a fair CAL program, what are the optimization criteria that drive the transformations that will lead to an efficient implementation on a specific target platform?
4. How do they translate into the steps of a CAL program development methodology?
5. What are the implementation scenarios to which optimization criteria and methodology can be applied?
6. How the design flow has to adapt to the different implementation scenarios?

The document is structured as follows:

- Chapter 2 introduces the context and the problems that are faced. The CAL language is introduced and fundamental notions are presented.
- Chapter 3 gives a definition of what is a right CAL program, explains different ways to obtain it from an initial specification written in C/C++.
- Chapter 4 defines what are the considered targeted platforms and the optimization criteria taken under account in project.
- Chapter 5 describes how a design space can be represented when dealing with CAL programs. According to the optimization criteria fixed by the designer, a 1D, 2D or multi-dimensional design space can be used.
- Chapter 6 describes how CAL programs can be localized in the previously defined design spaces.
- Chapter 7 describes how the designer can modify the CAL program in order to ensure that the corresponding implemented system fulfills the constraints fixed by the designer.
• Chapter 7 fully describes the design flow based on the ACTORS approach. It
details all the steps, starting from any specification down to a digital embed-
ded system.

• Chapter 8 gives an overview of the developed tools and how they are used
within the design flow. Deliverable WP1-D1B and WP1-D1C fully describe
the tools.

• Chapter 9 illustrates the theoretical aspects presented in the previous chapters
with a concrete example: Inverse Discrete Cosine Transform (IDCT), part of
an MPEG-4 decoder.

• Chapter 10 concludes the deliverable.
Chapter 2

Introduction

Designing and implementing complex signal processing systems on heterogeneous or multi-core platforms is a very difficult task. It is even more complicated when the process must result into an efficient implementation.

1 State of the Art

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 multicore era, imperative programs (i.e. a sequential paradigm) in which operations are executed in a very strict order is no longer the most appropriate way to specify programs targeted to run on several processing units. Furthermore, this sequential paradigm hides the intrinsic algorithm concurrency which must be exploited to improve the performance of the system.

Even if many designs are already exposing parallelism / concurrency using threads for example, it is more and more difficult to guess the behavior of the final application. It becomes obvious that this model of computation is not appropriate and must be reviewed.

2 Introduction to the CAL language

The CAL language is capable of concisely expressing complex signal processing algorithms exposing, when needed, the intrinsic algorithm parallelism. CAL is a dataflow and actor-oriented language that has been initially specified and developed by two members of the ACTORS project as a subproject of the Ptolemy project at the University of California at Berkeley. The initial CAL language specification was released in December 2003 [EJ03]. The CAL language describes algorithms using a set of encapsulated dataflow components called actors communicating with each other. An actor is a modular component that encapsulates its own state. The state of any actor is not shareable with other actors. Thus, an actor cannot modify the state of another actor. The only interaction an actor has with another actor is through input and output ports. The behavior of an actor is defined in terms of a set of actions. The operations, an action can perform, are to consume (read) input tokens, to modify internal state, and to produce output tokens. The topology of the connections between actor input and output ports constitute what is called a network of actors. It is expressed using an XML dialect known as network language (XDF) that also includes the possibility to have parameters and attributes that may be different for diverse instantiations of the same (parameterizable) actor in a net-
work of actors. Each action of an actor defines the kind of transitions that internal states can undergo and actions can be fired under specific conditions:

1. The availability of input tokens.
2. The value of input tokens.
3. The state of the actor.
4. The priority of that action.
5. The availability of free space to store output tokens.

Inside an actor, all the operations are executed sequentially. The execution of actions respects the following cycle:

1. Determine, for each action, whether it is enabled, by testing all the conditions specified in that action. This depends on the availability of token(s) at the requisite input(s), the value of input tokens, the state of the actor and the priority of each action.
2. If one or more actions are enabled, select one of them to be fired next.
3. Fire that action.
4. Go to (1).

Figure 1 illustrates a the structure of a CAL program.

Figure 1: CAL model structure

The transitions inside an actor are purely sequential: actions are fired one after another and each action is modeled as a pure sequential and atomic processing unit. At the network level, all the actors are completely independent and can work concurrently, each one executing their own sequential operations (consuming and generating tokens by firing the enabled actions). The selection order and the firing conditions for actions form the core of the design of an actor. CAL language provides a number of constructs for describing action selection, which include guards (conditions on the values of input tokens and/or the values of actor state variables), a Finite State Machine and priorities (actions may be related to each other by a partial priority order). The order of execution of the actors is in general not known a priori. CAL language provides a great flexibility to schedule actions according to the particular requirements and constraints of the targeted device chosen for the
final implementation. Very naturally, the language allows also hierarchical system
design. Each actor can be a network of actors. This approach facilitates modu-
larity and readability. Internal specification of any actor can be modified without
impacting other actors.

Ease of use The CAL language is a true programming language and not an in-
termediate format to automatically generate code. The notation is convenient to
write, with consistent syntax rules, keywords, and structures.

Minimal semantic core The CAL language is built on a very small set of semantic
concepts. Thus, it simplifies the compiler construction when transforming any
program into an equivalent program in the core language. Code generators can
be rapidly developed to compile the CAL program in any specific implementation
language.

Implementation independence and retargetability The first target for CAL ac-
tors was the Ptolemy II platform \texttt{pto}. A complete framework (Open DataFlow
\texttt{ope}) is currently being developed for simulating CAL networks and for generat-
ing hardware and software code. OpenDF is built under the Eclipse environment.
It makes these tools available for a large set of operating systems. CAL programs
are technology and architecture agnostic. This makes it possible to write, test and
run programs very quickly using OpenDF. Designers do not need special hardware
or libraries to design their own system in CAL. The implementation of CAL pro-
grams is done by means of appropriate hardware and software code generators
\cite{jmp08,wr08}.

3 Definition of terms

Causation Trace (CT) A causation trace of a dataflow program is a directed acyclic
graph such that:

\begin{itemize}
  \item every node is a firing of an action of an actor in the program,
  \item every edge from the node $v_1$ to the node $v_2$ is a dependency (either through
        a token, state or port) from $v_2$ on $v_1$, implying that therefore action $v_1$ has to
        be executed before action $v_2$.
\end{itemize}

Computational Load It corresponds to the amount of operations an action/actor
needs to perform during an execution of a CAL program.

Critical Path The CP of a program is the longest time-weighted sequence of op-
erations during an algorithm execution.

Dependency Graph (DG) It represents the same thing as the causation trace but
instead of having a node per firing of action, the DG is composed of one node per
action. The weight of an edge between $v_1$ and $v_2$ corresponds to the number of time
the dependencies between $v_1$ and $v_2$ exists. The weights on the node corresponds
to the number of times this action has been executed.

Partitioning This is the process which consists of assigning actors to processing
units.
**Processing Unit (PU)**  A PU is a component able to execute operations: processors, FPGA, DSP, ASIC, etc.

**Scheduling**  This is the process which consists of finding an ordering of the actions assigned to a processing unit.

**Scheduled and Partitioned CAL Program (SPCP)**  A SPCP is a CAL program in which actors have been assigned to processing units (partitioning) and an ordering of the actions on each processing unit has been defined (scheduling).

## 4 Preliminary notions

This section introduces several important notions in the context of the ACTORS approach for a better understanding of the deliverable.

### 4.1 Parallelism taxonomy

CAL language allows to expose different types of parallelism, from coarse-grained to fine-grained parallelisms. Coarse-grained parallelism means explicit parallelism in a CAL program (at actor level). Two kinds of (coarse-grained) parallelism can be identified: **task** and **pipeline** parallelism.

**Task parallelism** refers to pairs of actors that do not have precedence relations in a dataflow program. Precedence relations are inferred from the data dependencies between actors in a dataflow program (A precedes B means B cannot execute until completion of A). Task parallelism can be exploited by mapping those actors on distinct processing units.

**Data parallelism** refers to actors that has no dependence between successive executions. Thus, it is possible to parallelize instances of the actor that will work on portions of the input data. Data parallelism can be exploited by mapping different instances of the same actor on distinct processing units.

**Pipeline parallelism** refers to chain-structured region of the dataflow program. An actor does partial processing and then forwards the result to another. Pipeline parallelism can be exploited by mapping those actors on distinct processing units. This kind of parallelism is useful when dealing with a stream of data on which a certain number of processing should be applied. The gain will be observed on a set of data and a small cost induced by the initialization of the pipeline (that imply latency on the first processing).

Figure 3 depicts these different kinds of parallelism applied to the dataflow program of Figure 2. We assume that all actors have the same computation load, except C which is three times more complex and data parallelizable, and communication cost is not considered.

### 4.2 Nature of Actors

According to the way actors fire their actions, several types of actors can be distinguished:

**Static actor (SDF)**  Static actors, also known as synchronous actor [LM87], are the most simple to write and to analyze. They consume and produce a fixed
Figure 2: A dataflow program

Figure 3: Parallelization of the dataflow program
number of tokens each time they fire. They contain one or more actions that must have the same token consumption and production. A key feature of this model is that a static code analysis is feasible to detect if the program can be scheduled at compile-time. A static analysis produces a static schedule (a predefined sequence of actor firings), if it exists, which is free of deadlock and that uses bounded memory. More details on this topic is given in section 1.1 of chapter 11.

**Cyclo-static actor (CSDF)** Cyclo-static actors [BELP96] extends static actors to enable actors to cyclically change their behaviors. They contain one or more actions that may have different token consumption and production. As SDF actors, CSDF actors retain the ability to be scheduled at compile-time. The reader can refer to section 1.2.

**Dynamic actor (DDF)** Dynamic actors [Buc93] are the most general "deterministic" actors (an actor is deterministic if for a given input sequence of tokens, there is a unique output). They contain several actions with different token consumption and production, and they may test on token availability and values to execute their actions. This enable to express a large class of actors. Unfortunately, in general, compile-time scheduling is not possible anymore. Absence of deadlock and bounded memory consumption cannot be known at compile-time. As a consequence, DDF actor require to be scheduled at run-time. The reader can refer to section 1.3 for an example of dynamic actor.

5 What the implementation of CAL programs implies?

The level of abstraction of CAL programming language is high. Thus, implementation details are not described in this high-level language. Tackling implementation of CAL programs may reveal overheads that must be considered. First, communication channels between actors are turned into real FIFO implementations which may introduce additional synchronization protocols in case of concurrent memory accesses. Indeed, the implementation of actors introduces a controller (namely action scheduler) which is responsible for firing actions. Based on the current state, the token availability and the guards, the action scheduler selects the next action to fire. At a network level, if several actors are mapped on a single processing unit (PU), a scheduling policy must be defined to execute actors in the right order. For hardware processing units (FPGA, ASIC), the system is "self-scheduled" since all actors can run in parallel. Conversely, for software PU, the scheduling policy needs to be implemented since actors have to share the resources of the PU. Actors are executed concurrently, managed by a controller (namely actor scheduler). All of these "implementation details" may introduce overheads in terms of processing time, silicon usage, etc. This document presents methods to achieve the implementation of CAL programs while minimizing the overhead due to these control mechanisms (scheduling of actions, FIFOs, etc) implied by the CAL programming language.
Chapter 3

How to design the right CAL program?

As already stated, most of the applications are typically specified as sequential programs that do not reveal the available algorithmic parallelism, thereby hindering the efficient mapping of this application onto a parallel multi-processing units platform. This chapter addresses the topic of translation of a sequential specification into a CAL program which expresses parallelism. CAL language has interesting features (already discussed in 2) to represent both paradigms: actions are purely sequential pieces of code (sequential paradigm) but every actor works concurrently (parallel paradigm). As in any programming language, there are possibly infinite ways to write CAL programs, exposing more or less parallelism. Among all this set of programs, there exists a set of CAL programs that highlight the inherent parallelism and sequentiality of the algorithm. This subset of programs are considered to be right because automatic and efficient implementation from these CAL programs for specific platforms is possible. For instance, if the design is targeted on a pure sequential processor, refactoring can be automatically applied in order to create a pure sequential program. Conversely, if the design is targeted on a multi-processing unit platform (Multicore, FPGA, DSP, etc.), refactoring can be automatically applied in order to exploit as most as possible this parallelism. This refactoring is described in the next chapter. It is possible that programs express more parallelism than needed for achieving efficient implementation. Given a pipeline of static actors with a feedback loop from the last actor to the first one. The pipeline is not useful because a second execution of the first actor cannot be done until the last actor flows its output on the feedback loop.

The following sections explains how to build one of these right CAL programs.

Overview of the procedure  Before presenting the procedure, it is important to consider Amdahl’s law which is expressed by the following equation:

\[ S = \frac{1}{\gamma + \frac{1-\gamma}{P}} \]

with

- \( S \): speedup between sequential execution of the design and parallel execution on \( P \) processors.
- \( \gamma \): proportion of sequential part of the design.

A pure sequential program (\( \gamma = 1 \)) will provides no speedup if it is executed on a parallel platform. A pure parallel program (\( \gamma = 0 \)) will provides the maximum speedup on a parallel platform (\( S=P \)).
In a sequential specification ($\gamma = 1$), it can exist sequential parts and potentially parallelizable part as expressed by the following equation:

$$\gamma = \gamma_{seq} + \gamma_{PPS}$$

In order to reduce $\gamma$, two main axes can be followed: reducing $\gamma_{seq}$ or reducing $\gamma_{PPS}$.

Consequently, to build a right CAL program, comparable axes can be used:

1. reducing the sequentiality of the program in order to maximize the weight of the parallelism. Useless sequentiality may have been introduced in the specification. This operation is described in Section 1.
2. extracting more parallelism by splitting the program into concurrent actors. This operation is described in Section 2.

1 How to reduce sequentiality?

Every algorithm contains an incompressible and inherent sequentiality. The other operations of the algorithm can be done in parallel. Nevertheless, some unnecessary dependencies may have been introduced by the designer when writing a sequential program. Thus, it is useful to detect such unnecessary dependencies and to make the designer wonder if these dependencies are really necessary or not. For instance, the implementation of the sum of four values ($a, b, c, d$) can be implemented according to the graph presented in figures 3.1(a) and 3.1(b).

In the first implementation (figure 3.1(a)), the sum is calculated step by step, creating a useless dependency between each operation, while, in the second implementation (figure 3.1(b), $\text{Sum}_1$ and $\text{Sum}_2$ can be calculated in parallel on two PUs.

In order to measure the sequentiality of a design, it is useful to compute its critical path (CP) which is defined as the longest weighted sequence of events from the start of the program to its termination. Moreover, the contributions to the critical path (CCP) of each subset of the design (functions calls, compound statements, etc.) should be captured in order to detect where should be focused optimization efforts. The critical path is illustrated in figure 2, and can be calculated by an appropriate tool such as, for instance, the Software Instrumentation Tool (SIT).

Taking into account this concept, the following procedure can be defined:
1. analyze the specification (in C language) with a tool able to extract the critical path (CP) of the program and the contribution of each subsets of the program to the critical path.
2. select, in a decreasing order, the subsets of the program (procedure, function, comp. statements) with the higher contribution to the critical path.
3. the designer must ask himself if this contribution is really necessary and if the code can be rewritten to reduce it. If a redesign has been done, go to (1). Otherwise, go to (2). If there is no more subset to study, the procedure ends.

1.1 Lower bound

There is a theoretical minimum bound of the critical path of the execution of an algorithm (it is the incompressible and inherent sequential part of the algorithm). Unfortunately, it is not possible to determine automatically what the minimum bound is. The designer has an idea of the approximate value of the minimum possible critical path of the algorithm.

2 How to extract parallelism?

2.1 Functional decomposition

In order to extract the functional parallelism (MISD, MIMD, MPMD) of an existing sequential algorithm, it is important to locate where the actual processing is done. The Computational Load (CL) of each subset of the application can be used in such a way. It is defined by the sum of the number of operations of type $i$ (addition, division, subtraction... ) that were performed during a given execution, multiplied by their weight as shown in the following equation:

$$CL = \sum_i \text{NumberOfOperations}(i) \times \text{Weight}(i)$$
The weight of each type of operation should be individually set (typically, on a risk processor, a floating division takes longer processing time than bit shifting) in order to allow to take into account the characteristics of a specific processing unit.

In addition, the computational load can be composed with the critical path in order to know where should be focalized the designer efforts:

\[
\text{ParallelPotential} = \frac{\text{ComputationLoad}}{\text{ContributionCriticalPath}}
\]

This equation highlights the fact that it is interesting to work only on the subsets of code that are complex and have a low contribution to the CP. Knowing the ratio (i.e. the potential parallelism) of each subset of the program, the designer can split these subsets into smaller pieces of code that can be processed in parallel.

The transformation of the sequential specification into a parallel program is simplified if the program is structured in small units of independent pieces of computation because they can be directly converted into distinct actors.

2.2 Data decomposition

In order to extract the data parallelism (such as SIMD or SPMD) of an existing sequential algorithm, it is important to understand how the data are flowing through the program. The Function Data Transfers (FDT) is a measure that gives this information. It is defined as a record of the data transfers between the different parts of the program: a variable which is created or modified by a subset of program and accessed by another implies a data transfer between the two subsets of the program. Figure 3 illustrates a data transfer from the subset of program F1 to F2-2.

![Figure 3: FDT simple example](image)

Considering all the data transfers of the program, the designer can structure the program in subsets of code that can operate concurrently on the chunks of data. A few common examples include the following:

**Array-based computations** Concurrency can be defined in terms of updates of different segments of the array. If the array is multidimensional, it can be decomposed in a variety of ways (rows, columns, or blocks of varying shapes).
Recursive data structures  The designer can think of, for example, decomposing the parallel update of a large tree data structure by decomposing the data structure into subtrees that can be updated concurrently.

Moreover, by visualizing the graph of the transfers of data, the designer can directly detect the subset of code that can be converted into a pipeline structure that suits well to the data flow approach because the computation nodes can be translated straightforwardly to actors. Figure 4 gives an example of data flow graph which highlights several regions that can be pipelined.

Figure 4: Example of Data Flow graph extracted from a C program

2.3 Lower bound

Parallelism has been sufficiently exposed when the makespan of the theoretical Gantt chart of the execution of the program (considering that each actor is independent and communication cost is null) is equal to the critical path.
Chapter 4

Target Platforms and Optimization criteria

This chapter describes the different architectures of the target platforms and the optimization criteria addressed by ACTORS project.

1 Target platforms

The following target platform architectures are taken under consideration in the project:

- Single sequential processor
- Single FPGA
- One (sequential) processor and one FPGA
- Multi-processors (Multi-cores or multiprocessor platform)
- Multiple FPGA platform (N FPGAs)
- Multi-processors (N cores) and one FPGA
- Multiple processors (N cores) and Multiple FPGA (M)

Obviously, many other possible target platform architectures exist. However, this representative set should be sufficient in order to develop a design flow and allows considering other cases of interest.

2 Optimization criteria

Two main classes of optimization criteria are considered in ACTORS project:

2.1 Performance maximization

The aim is to achieve a design which exploits at best the available resources for maximizing the performance. Usually, in signal processing and telecom applications, performance is measured in terms of data throughput, data latency, sample frequency or similar metrics.

Single sequential processor platform  Obviously, there is no need of a CAL program composed of several actors for this type of platform. The best would be to have a single actor because it does not introduce any overhead for its implementation (taking care of FIFO, action scheduling). Exposing parallelism would not provide better results in terms of performance and resources. Thus, transformations of the CAL program must guide towards a reduction of the number of actors of the CAL program in order to achieve maximum performance.
Single FPGA platform In increasing the maximum working frequency of the design is a way to maximize performance. But in order to increase the frequency, the size of the logic blocks must be reduced. By considering the synthesis approach implemented in CAL2HDL (see Section 8), in which each actor is translated into a single process, reducing the size of the logic block implies either optimizing the code of the action or splitting this actor into several smaller actors. Exposing better the task and data parallelism (see Section 4.1) is another way to increase performances. Multiplying actors has a cost in terms of memory and hardware resources and the designer must be careful about the available size of the targeted FPGA.

Any platform containing several processing units Because partitioning of the CAL program has a large influence on its performance, increasing performance means finding the best partitioning of actors on processing units (PUs). It implies solving combinatorial problems, resulting either in optimal or approximate solutions. Surely, the performance can also be increased by optimizing actors according to their assigned target. But the main optimization potential lies in finding a partitioning (of actors on processing units) which exploits the parallelism provided by the platform. However, a CAL program exposing an appropriate level of parallelism of the application is necessary to fully exploit the processing power of the multi-PU platform.

2.2 Minimization of resources with a lower bound on the performance

The aim is to achieve a design which fulfills performance constraints (e.g., minimum throughput) while minimizing resources. There are several aspects in resource minimization:

- number of cores (for software)
- use of Configurable Logic Block (CLB) (for FPGA)
- total CPU usage
- power consumption
- memory requirements

Single sequential processor platform It aims at reducing memory usage, CPU usage, and/or power consumption. In terms of CAL programs, the same transformations as those specified for maximizing the performance should be done.

Single FPGA platform It aims at reducing surface usage (CLB), memory usage (BRAM), and/or power consumption (for example, by powering off some banks of the FPGA). Reducing the glue logic induced by controllers (action schedulers) is a way to achieve better performances. Merging actors results in saving resources but decreases performances.

Any platform containing several PU Minimizing resources in this case may have a twofold objective: the first is the same has already specified for single processing unit platform (reducing the memory usage, the CPU usage or the surface usage . . .). The second is to possibly find a CAL program that leads to the usage of the minimum number of processing units necessary to satisfy the given bound on performances.
Chapter 5

Design space representation

1 How to represent the design space?

Designing highly complex digital system does not result in a single solution. There are infinite ways to build a system. The design space represents the infinite set of solutions for a given system. The design space representation provides a mean to visualize graphically and to compare different designs according to given criteria (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. Generally, a 2D design space representation is sufficient to cover common design constrains. 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 1 (1D design space) and 2 (2D design space). For instance, to maximize the performance, an initial value can be set and increased progressively.

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 represents the ordering of actions onto each processing unit.

Figure 1: Representation of the Design Space taking under account one criterion
2 Localization of a design point in the design space

The efficient application of the methodology described in this deliverable requires to be able to position design points in the design-space characterized by meaningful design dimensions. This is essential to estimate and validate the effects of refactoring, transformations, partitioning of the CAL program on the performances of the implementations. This section briefly discusses the issues of localizing a CAL program and/or its associated implementation into the design space. We can distinguish different levels of abstraction for the CAL program and its corresponding design, each of them can be represented in a design space defined by different parameters of the program or different characteristics of the implementation on a target platform. Although many levels can be defined, we just limit our discussion to three main levels.

High level of abstraction: CAL program At this level the system is modeled by the asynchronous data flow paradigm. So the dimensions of the design space for instance are: the token throughput of each actor for a given data input, the minimum fifo size for concurrent execution of all actors and all profiling information on action execution per given input data. The measures of the value that position the CAL program in such design space are obtained by running the CAL program under a simulation environment interpreting the execution of the program according to asynchronous data flow computation model.

Intermediate level of abstraction: partitioned and scheduled data flow program execution taking into account a model of the target platform At this level the results of an execution at high level of abstraction are modified according to a given partitioning and scheduling of action that is compatible with the constraints of the data flow model of computation. At this level characteristics of the processing such as action complexity and cost of token communications estimated on a "model" of the target platform are taken into account. Metrics extracted by synthesized C/C++ code or synthesized HDL to RTL are used, but no profile measure on actual execution of code or HW on the target platform are taken. At this level characterization tools (see ) analyze statically and dynamically the CAL program executions and extract measures that are processed according to the platform model and to the mapping of the CAL program on it. Exploration tools (see ) use such pro-
cessed measures to evaluate the positioning of the CAL program and the associated mapping parameters into the design space. For instance this can be done by constructing a Gantt Chart of the execution of a CAL program for a given partitioning scheduling and for which the execution time of actions and token communication cost is estimated according to a given platform model.

**Low level of abstraction: native SW and RTL for each CAL program partition on a given platform** At this level SW and RTL performances can be profiled on the target platform and the positioning of the design in the space is characterized by dimensions such as execution time in the space dimensions. Such position is only possible after synthesis of SW and HW\[IMP^{+08}, WRR^{+08}\], compilation of software code and real executions on the target platform are necessary to obtain the necessary measures.

3 Several types of evaluation

There are several methods for evaluating a design, from high to low levels of abstraction:

Evaluations from pure analytical models can be too pessimistic (and thus not representative of the reality) since these models often consider the worst-case only. Considering Worst Case Execution Time (WCET) is not very representative, especially in signal processing algorithms. This type of analysis is suitable for CAL programs composed of static actors (see 4.2) because there is no uncertainties, their behavior is totally deterministic.

Simulations-based evaluations are well suited to study dynamic and unforeseeable effects in CAL programs whereas formally verifiable CAL programs require a deterministic behavior, given any stimuli. This type of analysis is necessary for CAL programs composed of dynamic actors (see 4.2) because of the uncertainties due to the dynamic behavior of these CAL programs.

Trace-based evaluation consists of recording the execution of the model. In CAL, it consists of recording the order of execution of the actions. The problem is that the evaluation is based on a given stimuli and may not reveal the real evaluation of the model for any stimuli. The designer must be very careful on the choice of the input data which must be representative of a usual execution of the CAL program. As for the simulation-based evaluation, this type of analysis is necessary for dynamic actors.

Cycle-accurate evaluation provides a good accuracy of evaluation because the level of refinement is defined by a single clock-cycle. It means that at any given clock cycle, the state of the simulator must be identical with the state of the evaluation. It is at a very low level of abstraction and is far from the CAL program.
Chapter 6

How to explore the design space?

This section describes how to explore the design space in order to reach a CAL design which fulfills the requirements, i.e. obtain a design point which is evaluated in the target region defined by the constraints.

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:

- transform the CAL program, and find a new partitioning and scheduling: section \[1\]
- apply a new partitioning and scheduling to the CAL program: section \[2\]
- apply a new scheduling to the CAL program, keeping the same partitions: section \[3\]

Figure \[1\] presents an example of exploration of the design space.

Figure 1: Exploring the design space with different transformations

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

1 CAL program transformations

One of the possibilities to explore the design space is to modify the CAL program directly. This section details the different possible transformations.
1.1 Splitting of actors

This transformation can be applied when more task parallelism or pipelining parallelism is required. As illustrated in Figure 2, it consists of substituting a single actor (or a network of actors) by another network of actors while respecting the I/O token functionality of the initial actor (or network of actors). It is the case when the program is composed of complex actors (i.e., actors firing complex actions) for which the potential task parallelism or pipeline parallelism of the actions need to be explicitly exposed by the program.

![Figure 2: Actor splitting transformation](image)

1.2 Merging of actors

This transformation can be applied when task parallelism or pipeline parallelism is not necessary to be exposed. As illustrated in Figure 3, the transformation consists of substituting a network of actors by a single actor while respecting the I/O token functionality. It is the case when the program is made of a large number of simple actors which have to be mapped for example onto an architecture made of a small number of processing elements, the exposed parallelism is too high. The benefit of merging is to reduce FIFO communications and synchronization overhead (token availability tests and guard checks).

![Figure 3: Actor merging transformation](image)

1.3 Splitting of tokens

This transformation should be applied when the granularity of the tokens flowing between actors is too coarse. The transformation consists of splitting the stream of input tokens and distributing the different part of the stream across several instances of the initial actor. Figure 4 illustrate this transformation where the initial input stream of "the edge detector filter" actor is splitted in two parts two instances of the initial actor. The main benefit of this transformation is that it increase the parallelism by exposing the implicit data parallelism of the model. However, it may introduce overhead due to the additional communications and instances of
actors (viz. availability tests and guards checks). Thus the trade-off between potential parallelism increase and loss of efficiency has to be carefully considered. This implicit data parallelism can be extracted using specific static code analyses, as exposed in section 3.1.

1.4 Vectorization of tokens

This transformation should be applied when the granularity of the tokens consumed and produced by actors is too fine. Vectorization, in the context of CAL program, is a transformation which increases the number of consumed and produced tokens per firing of an action. Listings 1 and 2 depict vectorization where the action of Listing 2 consumes and produces 64 times as many tokens as the one of Listing 1. The major benefit of vectorization is to reduce the number of firing which consequently reduces the control overhead (token availability tests and guard checks) of the actor.

Listing 1: Before vectorization

```java
actor add () int I1, int I2 ==> int O :
  action I1:[ a ], I2:[ b ] ==> O:[ a + b ] end end
```

Listing 2: After vectorization

```java
actor add () int I1, int I2 ==> int O :
  action I1:[ a ] repeat 64,
  I2:[ b ] repeat 64
  ==> O:[ [a[i] + b[i] : for int i in Integers(0,63)] ] repeat 64
end end
```
2 Partitioning

The partitioning consists of selecting a processing unit for each actor of the CAL program. This process needs to be driven by appropriate measures which are extracted and weighted by characterization tools. The complexity of this process increases with the number of actors and processing units. It results in combinatorial problems that grows exponentially and become NP hard. Thus, optimal solution is very hard to determine but heuristics helps in approximating the optimal solution. Partitioning is illustrated figure 5.

![Partitioning of a CAL program](image)

Partitioning static actors is much easier because the execution is known a priori. The problem gets harder when dealing with dynamic actors for which the execution is not known a priori. Dependencies between actions change according to input data.

3 Scheduling

The scheduling consists in ordering the execution of the actions (or actors) of the CAL program, independently from the partitioning. Scheduling is a very complicated issue because the CAL language can express different types of dataflow models. For each type of dataflow models, different scheduling strategies exist. The hard issue is to try to find a common and systematic methodology for scheduling such mixed models, all represented in CAL language.

The next section details the different existing approaches on the different type of dataflow models. Firstly, Synchronous Data Flow (SDF) is detailed and the second section presents the scheduling of Dynamic Data Flow (DDF). The last part concludes the scheduling issue.

3.1 Synchronous Data Flow

Synchronous Data Flow (SDF) [LM87] is a restricted dataflow model that allows for static analysis. Many studies have been already done on scheduling on Synchronous Data Flow (SDF) which considers a constant production and consumption rate of actors. The SDF restriction enforces token rates to be fixed a priori. An actor consumes and produces the same amount of tokens during firings. This restriction reduces the expressivity of the model but offers a high degree of analyzability. More precisely, actors can be scheduled statically with finite memories at compile-time. In other words, the execution time and the buffer requirement may be known at compile time, which is well-adapted for a variety of applications and especially critical realtime systems.
Figure 6: Example of a simple Synchronous Data Flow model

Figure 6 represents a CAL program composed of three actors. When actor A fires, it produces 2 tokens on its output port. When actor B fires, it consumes 3 tokens and outputs 2 tokens. When actor C fires, it consumes 3 tokens.

SDF programs reveal task parallelism but not data parallelism. To this end, the SDF model is turned into an equivalent expanded model as a directed acyclic graph (DAG, also referred as *acyclic precedence expanded graph* or *task graph* in the literature). The DAG exposes explicitly the available data parallelism. The DAG is characterized by actor weight that corresponds to the computation load and edge weight that correspond to the cost of communication. Figure 7 is the DAG of the SDF model presented in Figure 6. For instance, \( B_1 \) may be fired in parallel with \( A_3 \) in the following example.

Figure 7: Associated DAG of the SDF model

3.1.1 Analysis of SDF programs

The analysis of SDF programs is performed in the following steps:

- Checking consistency
- Calculating the repetition vector

**Consistency** A (weighted) incidence matrix, called topology matrix, is constructed from the SDF graph. Weights correspond to production and consumption rates. A SDF program is inconsistent if executing indefinitely this program leads to unbounded memory requirement. Consistency may be checked by analyzing the rank.
of the topology matrix. A SDF graph is consistent if its rank equals the number of actors reduced by one.

The topology matrix is:
\[
\Gamma = \begin{bmatrix}
2 & -3 & 0 \\
0 & 2 & -3
\end{bmatrix}
\]
and \(\text{rank}(\Gamma) = 2 = \text{card}({A, B, C}) - 1\).

**Repetition vector** Once consistency is checked, the repetition vector can be calculated. The repetition vector defines the number of times each actor must be fired to execute the program in bounded memory with no deadlock. The repetition vector \(q\) is the minimal integer solution of \(\Gamma \times q = 0\).

The equation for finding the repetition vector is the following:
\[
\begin{bmatrix}
2 & -3 & 0 \\
0 & 2 & -3
\end{bmatrix} \times \begin{bmatrix}
q_1 \\
q_2 \\
q_3
\end{bmatrix} = \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]
that lead to \(q = \begin{bmatrix}
9 \\
6 \\
4
\end{bmatrix}\).

The analysis of this SDF program indicates that these are 9 occurrences of A, 6 of B and 4 of C.

### 3.1.2 Scheduling for single processors

A schedule is a sequence of actor firings. A schedule is valid if each actor fires at least once (as stated in the repetition vector), does not deadlock (there is no buffer underflow) and there is no accumulation of tokens in fifo buffers \([LM87]\). In the case of single processor, SDF scheduling may be done following two main criteria. They consist in either minimizing the program (in term of line of code) or minimizing the data memory requirement.

On the one hand, the Single Appearance Scheduling (SAS) is optimal to minimize the program memory use by reducing actor repetitions to one using loops. SAS schedules does not always exist for a given SDF graph. But in some cases SDF graphs always have SAS schedules (for instance, every consistent acyclic SDF graphs) \([BLM96]\). Constructing a SAS for acyclic SDF graphs may be done like this: make an ordered list of actors by a topological sort, and then extend actors by their repetitions numbers. In the example, the topological sort gives ABC and then the schedule AAAABBBBCC, or \((9A)(6B)(4C)\) in a more compact form, is SAS. More details about existence and construction of single appearance schedules may be found in \([BL94]\).

On the other hand, some scheduling techniques aim at minimizing memory requirement. Unfortunately, the problem of finding a valid schedule that is buffer-optimal is NP-complete. That's why heuristics still must be used. The pull scheduling is based on a demand-driven scheduling policy: a token is produced if and only if the successor actor requires it. It results in executing "sink" actors of the program as soon as possible in the graph. In the example, AABABCABCABCABC is a pull schedule.

Table 6.1 shows the effect of scheduling strategy in term of lines of code and buffer requirements. Lines of code are counted symbolically and reflect the number of node changes in the schedule. The pseudo code of Listing 3 shows a possible synthesis from the SAS schedule.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Lines of Code</th>
<th>Buffer requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>3</td>
<td>18+12=30</td>
</tr>
<tr>
<td>Pull</td>
<td>16</td>
<td>4+4=8</td>
</tr>
</tbody>
</table>

Table 6.1: Scheduling strategy example
for(i=0; i<9; i++) A();
for(i=0; i<6; i++) B();
for(i=0; i<4; i++) C();

Listing 3: pseudo code associated to the SAS schedule

A variety of hybrid scheduling techniques exists, which make a trade-off between program and data memory minimization [BLM96].

SDF analysis and scheduling techniques of CAL programs are more described in Task 1.4 "Model Compiler" (D1e deliverable).

3.1.3 Scheduling and partitioning for multiprocessors

In case of multiprocessors, partitioning has to be considered as well as scheduling. Partitioning means determining processors where actors will be allocated. The goal is to find a partition (where actors are located) and a schedule (how actors are ordered on each processor). Hereafter, the term multiprocessor scheduling, or simply scheduling, is used to refer to the scheduling and partitioning.

The problem of scheduling a DAG on multiprocessor is NP-complete. Therefore, heuristics with polynomial-time complexity are widely used. Most of scheduling techniques are based on list scheduling [KA99]. It consists in making an ordered list of nodes (topological sort, etc.), assigning them priorities based on a specific strategy (as-soon-as-possible start-time, longest processing time, critical path, etc.) and finally scheduling nodes according to priorities in order to optimize a given criterion.

Considering the maximization of the throughput of the system, the solution of the scheduling and partitioning problems would ideally lead to the Gantt chart presented in figure 8.

Figure 8: The ideal schedule to maximize the throughput

In this example, it is assumed that communication cost is null and the number of processor is unlimited. However, in the real world, such assumptions are not conceivable. Hereafter, it is assumed that scheduling is done on P processors with communication cost and interconnection topology of the network of processors are arbitrary.

The relevant measures in the multiprocessor case are:

- Communication cost (edge weight: time for a communication)
- Computation cost (node weight: execution time of each actor)

Examples of optimization criteria for multiprocessor scheduling and partitioning are:

- Minimizing the makespan (the time difference between the start and finish of the first iteration).
- Maximizing the throughput (time delay of complete application execution - even if actors are in different iterations).
• Minimizing the resource for a given throughput.

Suppose a DAG with two adjacent actors, the computation cost is assumed to be equal to 2 for the two actors on both processors $P_1$ and $P_2$, and the communication cost is equal to 1 from a processor the other via the communicator $C$ and 0 on a same processor. Scheduling the DAG to 2 processors may leads to different schedules according to given optimization criteria. The first schedule minimize the makespan ($4<5$), while the second maximize the throughput ($1/3 > 1/4$).

![Figure 9: Scheduling two nodes on two processors](image)

Scheduling and partitioning DAG for multiprocessors from CAL programs are described in Task 1.3 "Design Space Exploration" (D1e deliverable) and Task 1.6 "Optimizing performance" (D1g deliverable).

3.2 Dynamic Data Flow

By definition, the behavior of a Dynamic DataFlow (DDF) program cannot be predicted at compile time. As a reminder, the reader can read the definition of a dynamic actor in section 4.2. CAL programs are generally not completely dynamic and some actors are static or cyclo-static. Thus, some parts of the CAL program can be studied statically in order to define the scheduling of the actions corresponding to these static portions.

This section describes how to deal with such model and presents a systematic way to derive schedules for dynamic CAL models. The approach is based on the fact that a dynamic actor is in fact a composition of several execution paths. According to the values and/or the presence of input tokens, a execution path is ran. An execution path is a sequence of actions of actors that is not dependent on the presence or value of tokens. It is a static part of the actor, meaning that it can be scheduled statically. In this execution path, the predecessors and successors in the execution of actions are known and is not dependent on the execution but depends on the way the CAL program is written. We assume that it is possible to enumerate all the executions paths of dynamic actors.

Consequently, each dynamic actor is composed of two parts: a dynamic and a static part (executions paths). The dynamic part acts as a "multiplexor" which triggers the execution of one of the Execution Path (the set of all the Execution Paths constitutes the static part of the dynamic actor). The dynamic part cannot be removed: it is part of the algorithm to make a decision on the value of a previous data. This decision must be taken only when the previous data is known, i.e. this
decision can be only taken at run-time. As a consequence, dynamic part is not schedulable \textit{a priori}.

A dynamic actor can be represented as shown in Figure [10].

![Diagram](image)

Figure 10: Each dynamic actor can be represented as a dynamic and a static part

The dynamic part triggers one of the Execution Path according to the value(s) of token(s) coming from the "IN Control" port. The number of tokens consumed by each of the Execution Path (EP) on the IN Data port may not be the same. For example, EP 1 consumes only one token and EP 2 consumes 5 tokens. Moreover, it may happen that all the EP do not consume tokens from the same port(s). This case is not represented in Figure 10.

A CAL program is composed of static and dynamic actors. Static actors can be scheduled statically. Dynamic actors are sets of static parts called Execution Paths (EP). The new approach outputs a dependencies graph between actions of the dynamic and static actors of the entire (or a part of) CAL model. Consequently, a dependence graph can be built only by considering only a given EP for each dynamic actor. Evidently, the number of possible schedules increases very quickly as far as the number of dynamic actors increases. An example of dependencies graph is provided in Figure [11] considering the Execution Path contained in the actions involved in Texture Decoding and Motion Compensation.

![Diagram](image)

Figure 11: Dependencies Graph

This graph represents several actors. This graph means that in order to execute the green action, the red actions must be previously executed. It represents the dependencies between actions. A specific schedule has not been decided yet. This graph represents all the dependencies that admissible schedules must respect.
From this dependencies graph (which can be considered as a SDF graph), a schedule can be determined. Constructing a proper schedule for a couple (scenario, optimization criterion) is a combinatorial problem which can be solved using the approach presented in Section 3.1. Depending on the requirements of the application, it might be necessary to minimize the memory resources or minimizing the delay or even maximizing the throughput. According to these optimization criteria, different schedules can be generated from this dependencies graph. Generating a schedule is performed at compile time.

The approach presented in Chapter 3.1 performs the scheduling and the partitioning at the same time. These two aspects (partitioning and scheduling) are closely linked. Determining the partitioning influences the scheduling and vice versa. The previously presented approach schedules and determines the partitioning as illustrated in Figure 12.

![Figure 12: Scheduling of 5 actors](image)

The approach determines the scheduling and the partitioning at the same time. Thus, it outputs also the schedule of each processing element.

Once the static actors along with the execution paths of dynamic actors have been scheduled statically, there are still the dynamic parts of the dynamic actors which must be scheduled. This schedule is done compulsory at run-time because the value of tokens are not known \textit{a priori}. The dynamic part of the actor reads and evaluates tokens and decides which Execution Part must be ran. The selection of an EP in a dynamic actor can lead to the execution of a Statically Schedulable Region (composed of static actors) because EP and SSR are static and can be scheduled at compile time. But these computed schedules can only be triggered at run-time. Figure 13 illustrates the concept.

![Figure 13: Off-line schedules composed of EP and SSR are triggered at run-time](image)

### 3.2.1 Possible limitations of this approach

The main limitation of this approach is to find the running modes of the CAL programs. For instance, in the MPEG-4 SP decoder, one can schedule statically the
decoding of an INTRA block or an INTER block. The problem is to find a systematic methodology for finding the actions involved in this running mode. This running mode is not explicit in the CAL program because the decoder is composed of several independent actors. These running mode can be detected through a given token called BTYPE which is like an identity card of the current block being decoded. The MPEG-4 SP decoder could have been written differently. Thus, finding a generic and systematic methodology is problematic. Furthermore, according to the application and the way that the CAL program is written, it may result in very complex representations of the models and thus complicated to analyze.
Chapter 7

Design Flow

This chapter summarizes the different steps of the design flow aiming at elaborating a complete scheduled and partitioned digital embedded system starting from a high level CAL program. Figure 1 illustrates the main steps of the design flow.

The design flow is composed of one preliminary step and two loops:

1. A preliminary step aiming at building a fair CAL program from any specification. This step must result in an architecture-agnostic CAL program which exposes the inherent parallelism of the algorithm and in which unnecessary sequentiality is removed.
2. An iteration loop that is responsible for building an appropriate CAL program (using CAL transformations, scheduling, partitioning) for the considered target platform (red loop in Figure 1).
3. An iteration loop that is responsible for evaluating the design, given a table of characteristics that describes and characterizes the architecture (blue loop in Figure 1).

The interlacing of these two loops is the strength of the design flow because it allows the detection of bottlenecks and the limitation of the current CAL program at the very early stages of the design flow. Thus, it allows the designer to take the right decisions very early and to modify the CAL program.

1 Sequential to Parallel transformation

When designing a new digital embedded system, either the application does not exist and the designer writes it entirely or a specification already exists. Thus, the designer does not have the choice and must deal with this specification to write the CAL program. There are several ways to specify an application:

- Textual description
- C/C++ models (like in ISO/MPEG standardization organization)
- UML models
- Other type of models and graphs …

The specification must be translated into a CAL program whatever the form of the specification of the application. Section 3 introduced the notion of the goodness of CAL programs. Before starting the design flow, it is preferable to start with a fair CAL program as soon as possible. This first step aims at describing a systematic procedure for building a good CAL program. The procedure has been described in Section 3.
Figure 1: Illustration of the design flow
2 Characterization of the CAL program

Static and dynamic (trace-based and simulation-based) analysis are performed to characterize as much as possible the CAL model in order to extract meaningful measures useful for the different application scenarios. Chapter 4.1 describes the measures that can be extracted during this step.

2.1 CAL level measures

From a CAL program, a large number of measures can be extracted, either at action level, at actor level or at network level.

Action level  The production rates of each action, the size of the ports can be extracted from the action.

The following is a non exhaustive list of measures at the action level:

- Number of executions of each action during a complete execution
- Complexity (number of operators executed and associated data types) by an action in one execution.
- Potential parallelism of each action (calculated as ratio between the action complexity and the critical path)
- Measure of the length of the critical path of each action

Actor level  From the way tokens are consumed and produced and the structure of the code, the type of actor can be deduced (static, cyclo-static or dynamic). Moreover, the analysis can also establish statistics on the number of operators used, locate loops, etc…

The following is a non exhaustive list of measures at the actor level:

- Type of actor (static, cyclo-static or dynamic)
- Number of consumed and produced tokens per cycle for cyclo-static actors
- Number of consumed and produced tokens for static actors
- Size of the ports of each actor
- Execution paths of the dynamic actors
- Contribution of the critical path of each action
- Total complexity (number of operators executed and associated data types) by an actor during an execution

Network level  As soon as two actors are connected, their consumption and production of tokens are closely linked. In some cases, relationship between the rhythm of production and the rhythm of consumption can be deduced. Moreover, the dependencies between actors are clearly expressed in CAL description (an actor must wait tokens from these actors…).

The following is a non exhaustive list of measures at the network level:

- Dependencies between actors
- Dependencies between actions between actors (predecessors: A->B, C->D,… ) thanks to the signature of each action and the port connections between actors
- Relation between the number of execution of two connected actors
- Predecessor of each action at each execution
- Sequence of actions executed during the simulation
- Memory bandwidth of data exchanges of each action.
• Contribution of each function of an action to the action critical path (intended as longest path of the execution flow graph).

2.2 At lower levels of abstraction

The reason why characterization of the CAL program is also done at lower levels of abstraction (i.e. C/C++ or VHDL) is that the profiling is much more precise. Indeed, profiling at high level or low level of abstraction can provide different results. That is the reason why a characterization though generated implementation code is very valuable. Specificities of the target platform are more taken under account. For example, in C, an integer variable is assumed to be stored on 4 bytes for 32 bits processors or 8 bytes for 64 bits processors. Conversely, in CAL language, the size of an integer variable can be specified (from 1 bit to N bits). Thus, the estimated memory usage can differ according if the characterization is done at high or low level.

The tools available today, such as Visual Studio Performance Suite, ISE, Valgrind, will be used in order to extract relevant measures.

3 Weighting measures

During the characterization step, static and dynamic analysis are performed to extract raw measures from the CAL program. These measures must be adapted to the considered target platform architecture. This step outputs quantified measures which are used by the library of algorithms and heuristics in order to define a partitioning and scheduling (section 4) of the program according to a targeted platform. Using non weighted measures in these algorithms would result in bad results. Weighting the measures according to the targeted architecture aims at evaluating the current design much more precisely.

As an illustration, let’s consider the computation of the execution time of the actions. Actions are atomic piece of sequential code. A certain amount of operations are executed, addition, substraction, multiplication, division, memory accesses … These operations does not take the same time on each processor. Digital Signal Processors (DSP) have special instruction sets for executing multiplications and additions. Thus, these operations won’t take the same time as in an usual processor. The weighting process aims at dealing with this issue.

4 Partitioning and Scheduling

Once the measures have been extracted and weighted in the previous steps, it is time for using them when searching for a partitioning of actors and scheduling of actions of the CAL program. This step aims at assigning actors to processing elements (partitioning, see section 2) and an ordering of actions on each processing element (scheduling, see section 3).

Partitioning and scheduling are so closely linked that there are many different approaches:

• Choose partitioning and determine a scheduling
• Choose a scheduling and determine a partitioning
• Determine scheduling and partitioning at the same time
• Choose partial partitioning and/or scheduling and then fill the holes as best as possible.

Some examples of algorithms are:
Computational load balancing  The idea is to distribute the actors on processing units according to their computational load. ProfiCAL provides the computational load of each actor for a given execution. The actors are distributed such that each processing element has approximately the same load which is the sum of the computational loads of each actor of the processing element.

Clustering  The idea is to distribute actors on processing units according to their connections to other actors. Two actors are said strongly connected when the amount of data exchanged between these two actors is large. When actors do not exchange data, they can be placed on two different processing units. The aim is to make \( N \) clusters of connected actors in order to assign them to \( N \) processing elements.

Serialization points  The aim is to determine which part of the application is the best candidate for being executed in software. The analysis of dependencies and the potential parallelism of each actions is necessary. The idea is to group actors of the largest set of connected actors which constitutes a serialization point in the algorithm.

5 Evaluation of the design point

In a standard conception process, the project includes hardware limitations, probably to take into account hardware’s price, compatibility with the manufacturer product’s range etc. The design team will usually be limited by hardware choices and will make a first hardware pruning by choosing one model of FPGA, a type of processor or memory model. It is important that the algorithm is coherent with the architecture on which it is going to be mapped. The evaluation process aimed at evaluating in the early stages of the design flow if the design is going to fit into the targeted architecture. The problem is that doing this evaluation at high level of abstraction can results in a wrong evaluation. The aim is to take under account more and more characteristics of the targeted platform in a iterative process in order to evaluate the design as most precise as possible and to detect bottlenecks as soon as possible. Figure 2 illustrates the principle.

The evaluation process is represented by the blue loop in figure 1. More specifically, it consists in localizing the designed CAL program in the design space. The localization issue has been already discussed in section 2. If the designer chooses to keep the CAL program unchanged with the computed partitioning and scheduling, he must refine the evaluation until all the characteristics are taken under account. The characteristics are added one by one to shorten the processing time. By this way, at any point the designer can choose another option. The mapping is considered as finished when all the characteristics have been taken under consideration.

The evaluation process is based on the construction of a Gantt Chart which models the execution of the CAL program with given input data. Taking more and more characteristics of the targeted platform when building this Gantt Chart guarantees a more and more precise evaluation of the design according to the optimization criteria.

A non-exhaustive list of the characteristics taken under account during the construction of the Gantt Chart is the following:

- working frequency of the target
- architecture and size of data memories
- size of the FPGA
- speed of the communication protocols between Processing Elements
Figure 2: The precision is more and more precise when taking under account more and more specifics of the target platform

- scheduling policies

5.1 Determining the feasibility

It may happen that a pure sequential implementation of the algorithm in the projected platform is enough to meet throughput requirements. It can be easily checked by comparing the total computational load, its distribution amongst the different operators (as previously mentioned, a division often takes a long time compared to a bit shift…) and the hardware performances to avoid a useless partitioning / design phase if sequential implementation is enough.

It may happen also that the projected target platform will never be able to sustain the computational load, whatever the partitioning is. To check this point, the user can compare the algorithm critical path and the hardware performances. The critical path being the shorter sequence of operation to complete execution, it is certain that an hardware not able to perform this computational load in the desired time will never meet the requirement, regardless to the distribution of the load in the available processing units. If the design is in this position, the projected hardware has to be upgraded.

5.2 Determining execution time of actions assigned to software

Execution time of each operation (addition, memory access, substraction, multiplication,…) belonging to actors assigned to software depends on:

- communication cost between processors due to the exchanges of tokens between actors
- duration of memory accesses operated in actions
- execution time of the operations according to the capabilities of the target
- time for the action scheduler of each processing element to choose the next action to execute

Figure 3 illustrates the different characteristics to take under account in order to evaluate the scheduled and partitioned CAL program.
5.3 Determining execution time of actions assigned to hardware

Execution time for each action of the actors assigned to hardware. It is assumed that the execution time of an action is equal to a clock cycle of the FPGA (as in the CAL2HDL code generator). The length of a clock cycle depends on the depth of the logic implemented on the FPGA for this action. The longer the depth of the logic in the FPGA is, the slower the frequency is. Given the maximum possible frequency, the execution time is known.

5.4 Construction of the Gantt Chart

Knowing the execution time of each action of the program, one can construct the corresponding Gantt Chart. The execution time of the actions can be determined more or less precisely according to the platform specifics that are taken under account. The generation of the Gantt Chart is based on the causation trace (provided by TraciGen) which provides the order of execution of the actions while revealing the dependencies between them.

6 Designer Choice

According to the evaluation, the designer can choose between four options:

- Consider that the design point is close enough to the target region (the region defined by optimization criteria) and keep the triple (CAL model, Partitioning, Scheduling) and go to the evaluation step (blue loop).
- Choose a new algorithm or heuristic to modify the partitioning and/or the scheduling
- Transform the CAL model if the modification of the scheduling and the partitioning are not sufficient to reach the target region (chapter 5.1)
- Change target platform if all the meaningful refactorings have been done and there is no other solution (target platform are briefly described in chapter 2.1).

Because of the complexity and the multiplicity of the decisions needed to map an algorithm on a given architecture, the problem cannot be solved in one iteration loop (red loop). Several iterations are necessary (thus, several refactoring) in order to obtain a good CAL model for a given scenario.
7 CAL program transformations

Transformation of the CAL program occurs when the designer decides that the CAL program is not good enough for the targeted platform. Transformation of CAL program has been already discussed in section 1.

In the case of an architecture including fixed elements, most likely, the sequential implementation will not be compliant with the requirements. On the contrary, in a pure actor language approach or in a totally free hardware, the purely parallel implementation will be far enough. The issue in this simple statement is that the first case is not good enough for the design, and the second is far too expensive. Therefore a tradeoff is needed, the user will generally choose to partition the algorithm in several computing units, in order to spread the complexity across computation units, and thus optimize the throughput.

8 Code Generation

The entry point of this step is the scheduled and partitioned CAL model. This step consists in deriving an efficient implementation from the CAL program.

When generating code for a hardware platform, there is no problem of efficiency because the CAL programming language suits very well for hardware code generation. When dealing with software implementation, some efficiency problems arise. The major problem is the overhead induced by:

1. the tests of the availability of tokens in FIFO
2. the evaluations of actions guards for deciding if the action can be fired.

Furthermore, several types of software platform exist. Figure 4 shows the different kinds of software targets.

According to the type of target, the generated code can not run optimally. Several levels of parallelism can be distinguished in software platforms. From an assigned network of actors on a target, different cases can be distinguished. The network of actors can be assigned on:

- A pure sequential target, no parallelism can be exploited.
- A target which presents some flow parallelism. In this case, some kind of parallelism can be exploited by the compiler so that to use the features of the target.
- A multi-core processor. Deeper analysis must be done in order to exploit the multiple cores.
Figure 4: Illustration of the different kinds of software targets
Chapter 8

Implementation of the design flow

This section describes an overview of the tools which implement the methodology described in section 7. The tool infrastructure has been developed in Java under the Eclipse Framework. It is composed of a multitude of plug-in, each one implementing a given feature. Figure illustrates the tools infrastructure by showing where each plug-in takes place in the design flow described in section 7.

The list of implemented modules are categorized as follows:

Characterization Tools

StatiCAL performs only static analyzes of the CAL code.

ProfiCAL performs a dynamic analysis of the CAL code given an input stimulus. It records the computational load of the actions composing the model.

TraciGen records an execution of the CAL program by mean of a dependency graph and a causation trace.

TraciCAL analyzes the causation trace and dependency graph and extracts statistics.

Exploration Tools

WeightCAL weights the raw measures extracted from the other characterization tools.

CrossCAL crosses the measures extracted by the other tools in order to build advanced measures.

SchedulCAL helps in creating partitioning and scheduling of the CAL program.

EvalCAL evaluate how good the designed CAL program (the scheduled and partitioned CAL program) is.

AnalytiCAL displays the results obtained with the other characterization and exploration tools.

All the modules (except AnalytiCAL) have command lines capabilities and can be run independently. Thus, it is easy to make a complete scripted design flow with these tools. These modules are described more in details in deliverable D1B (CAL Mapping Tools) and D1C (Exploration Tools). An user-friendly framework has been also developed in order to ease the use of these different tools. This framework is called CAL Design Suite. Figure illustrates the architecture of this environment.
Figure 1: The Tools infrastructure
This environment is composed of the different modules cited above. The CAL Design Suite is a wrapper which allows launching all the tools and save the results of the different modules in a given repository. Each tool need inputs and generate outputs. The CAL Design Suite manages the inputs and outputs of each tool and guide the designer which operation can be done next.
Chapter 9

Case Study : Inverse Discrete Cosine Transform

This chapter illustrates the design flow presented in chapter 7 through a case study, Inverse Discrete Cosine Transform (IDCT). This case study is the ISO/IEC MPEG fixed-point implementation of two-dimensional Inverse Discrete Cosine Transform (2D-IDCT) [ISO07]. A module that performs fixed-point 2D-IDCT separably, by applying the MPEG standardized fixed-point 1D-IDCT to the coefficients first in horizontal direction, followed by a transpose operation, and then finally another MPEG standardized fixed-point 1D-IDCT in vertical direction.

This module is a computational intensive part of the MPEG4-SP video decoder. For instance, the designer may expect to achieve the following requirements for the MPEG4-SP decoder:

**Performances** it must be capable of decoding 25 frames per second with a 4CIF (704x576) format, i.e. decoding 39600 macroblocks per second.

**Resources** the target platform is composed of four identical processors, working at 400 Mhz.

Assuming that the IDCT should not take more than 22.7% of the total processing power, the IDCT must process a macroblock at most in 5.73$\mu$s. Under a working frequency of 400 Mhz and considering that each operation takes one tick of clock, the expected makespan (discrete time) of the Idct must be less or equal to 2293 ($=5.73 \mu s * 400$ Mhz).

1 Building the right CAL program

This step consists of building the right CAL program from the ISO-IEC reference software in C of the IDCT.

1.1 Critical path analysis

The structure of the code of the reference software is illustrated in Figure 1.

Table 9.1 shows that the highest contributor to the critical path (CP) is the scaled_1d_idct function. As explained in Chapter 3 Section 1, efforts should be focused on the parallelization of this function to reduce the sequentiality of the reference software.

By studying the code, the designer can note that the processing of the scaled_1d_idct is applied to two distinct sets of data: odd and even elements are processed differently. So, if these processings are made in parallel, the global sequentiality of the
design is reduced. Consequently, one can implement the CAL model presented in figure 2.

1.2 Data decomposition

Data decomposition consists in extracting the dependencies between each subset of data and how the data are flowing through the design. The FDTs (Function Data Transfers) are useful measures that simplify the work of data decomposition. These measures can be presented graphically as shown in figure 3. Each node represents a function call or a compound statement (block, conditional branch, switch, loop) and each edge represents a data dependency expressed in Byte.

In figure 3, there are no data transfers (no dependencies) between each scaled-1d_idct-Func functions. It means that their processing can be done in parallel on distinct PUs. Moreover, the design is clearly exposing a pipeline structure (scale, idct1d for rows, idct1d for columns and downshift) that allows to distribute the processing on several PUs. This last optimization may increase the throughput of the system.
Figure 3: Function Data Transfer
Taking into account all the previous comments, the CAL network presented in figure 4 can be implemented.

Figure 4: data and pipeline parallelism of the ISO/IEC IDCT

2 Analysis: first iteration of the design loop

2.1 Characterization

Static and dynamic analyzes are performed during the characterization. Static information like port sizes and consumption rates are extracted from the CAL code. Dynamic analyzes are performed with an input of 10 coded blocks. For the sake of clarity, all the results of the characterizations, weighting and partitioning/scheduling have been gathered in table 1.1 in the annex section

2.1.1 Computational load and number of calls

The total computational load (CL) of each action is extracted. It corresponds to the amount of operations an action needs to perform during the execution of a CAL program. Figure 5 illustrates the results: each node represents an actor and the size of the node is proportional to its computational load. The most complex actor are: scale, transpose, retranspose, splitRow, splitCol, joinRow, joinCol, rshift. There are many instances of the "col" and "row" actors which are not very complex.

Figure 5: Computational load of each action
2.2 Weighting of the measures

The weighting of the computational load measure of each action aims at predicting the execution time of the action on a given target. For a first iteration of the design loop, every operations have all the same weight equals to 1. Surely, this is not an accurate prediction but it constitutes a nice overview of the execution. The results are presented in table 11.1 in the annex section 3.

2.3 Partitioning and scheduling

The partitioning consists of assigning actors to processing units. The scheduling is defined as a priority list for each processing unit. In case several actions are assigned to a same processor and several of these action are fireable at the same time, the scheduling policy must be able to decide which action to fire next. Thus, a priority is assigned to each action in order to decide which action must be fired next.

The partitioning is computed with a load balancing algorithm in which the criteria is to assign to each processing unit a given amount of computational load. The scheduling of actions is defined thanks to a priority list in which the most called action has the highest priority. The results of the partitioning / scheduling is shown in table 11.1 in annex section 3, considering four processors.

2.4 Evaluation

The evaluation consists in building a Gantt chart of the execution with a maximum of specificities of the platform so that it predicts as precisely as possible the behavior of the future system. The Gantt chart takes into account the partitioning and the scheduling, the execution times of actions (decided in the weighting phase) and the dependencies between actions. The resulting Gantt chart is shown in figure 6. The makespan of the execution (on 10 blocks) is equal to 29481. It is over the requirements.

Figure 6: Complete Gantt chart of the execution of the initial program

The target platform is composed of four processors. The CAL program is composed of 74 actors. The exposed parallelism cannot be fully exploited by the platform. Having numerous actors implies an additional computational load (CL) in order to distribute data between actors: the total CL equals 263030. As the number of processors is limited to four, merging actors in order to delete this extra computational load is a solution to make the system more efficient for this platform.

3 First optimization: feedback loop

3.1 Refactoring: redesign of the program

The bottleneck mentioned above can be solved by merging actors. The resulting CAL program is show in figure 4. Each of the networks of actors "row_i" and "col_j"
are replaced by a single and monolithic actor. Thus, the number of actors has been reduced from 74 to 26.

3.2 Characterization

Dynamic analyzes are still performed with the same input of 10 blocks as in the first iteration. For the sake of clarity, all the results of the characterization, weighting and partitioning/scheduling have been gathered in table 9.2.

3.2.1 Computational load and number of calls

Figure 7 shows graphically the results. Green bars represent the average computational load (CL) of each action. 100 % corresponds to a CL of 2611 operations. Yellow bars represent the total CL of each action. 100 % corresponds to a CL of 26110 operations. Green bars represent the average CL of each action. The number of firing of each action is also extracted and shown in table 9.2.

3.2.2 Data transfers

Knowing the number of executions of each action, their respective consumption rates and the size of each port of the actors, the data transfers between actors can be easily determined. Figure 8 illustrates the size of the data transfers between actors. For the sake of clarity, the reported results are at the actor granularity but tools can indicate what is the contribution of each action to the data transfers.

3.3 Weighting of the measures

Every operation still have all the same weight equals to 1.

3.4 Partitioning and scheduling

The partitioning and scheduling are computed with a load balancing algorithm in which the criteria is to assign to each processing unit a given amount of computational load. The results of the partitioning / scheduling is shown in table 9.2 still considering four processors.

3.5 Evaluation

The results of the characterization, weighting, scheduling and partitioning are reported in table 9.2.

<table>
<thead>
<tr>
<th>Table 9.2: Results of the first analysis of the CAL program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>action0</td>
</tr>
<tr>
<td>action0</td>
</tr>
<tr>
<td>action0</td>
</tr>
<tr>
<td>action0</td>
</tr>
<tr>
<td>action0</td>
</tr>
<tr>
<td>action0</td>
</tr>
<tr>
<td>action0</td>
</tr>
<tr>
<td>action0</td>
</tr>
<tr>
<td>action0</td>
</tr>
</tbody>
</table>
The resulting Gantt chart is shown in figure 9. The makespan of the execution (on 10 blocks) is equal to 23046. The system still does not fit the requirements but the performances can be improved by applying a better heuristic.

The makespan has been improved. As the load balancing is a basic heuristic, a more efficient heuristic can be applied to determine a better partitioning and scheduling of the CAL program.

4 Second optimization: feedback loop

4.1 Refactoring: apply new partitioning algorithm

The new partitioning and scheduling is computed with a simulated annealing algorithm in which the criteria is to reduce the makespan of the total execution on four processors. The results of the partitioning / scheduling is shown in table 9.3.

4.2 Evaluation

The new makespan is 22783. The resulting Gantt chart is shown in figure 10.

The CAL program fulfills the requirements.

5 Synthesis of the CAL program

The optimized CAL program built thanks to the design flow can be synthesized by using the tools corresponding to the target platform. Surely, the scheduling and partitioning are taken into account in order to benefit from the previous study. Profiling the generated code allows to extract more precise measures that can be re-used into the tools at CAL level. For instance, the weight of each operator can be adjusted, and the communication cost can be inserted as an additional characteristic. By this way, the CAL program can be analyzed again. Consequently, the partitioning/scheduling may change and a better evaluation can be done.
Figure 7: Total and average computational loads of each action
Figure 8: Data transfers between actors

Figure 9: Complete Gantt chart of the execution of the refactored program

Figure 10: Complete Gantt chart of the execution of the final program
<table>
<thead>
<tr>
<th>Actor</th>
<th>Action</th>
<th>Target</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>idct2d$transpose</td>
<td>action0</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>idct2d$col_0</td>
<td>action0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>idct2d$retranspose</td>
<td>action0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>idct2d$row_4</td>
<td>action0</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>idct2d$joinRow</td>
<td>action0</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>idct2d$col_1</td>
<td>action0</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>idct2d$row_1</td>
<td>action0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>idct2d$col_4</td>
<td>action0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>print</td>
<td>action0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>idct2d$joinCol</td>
<td>action0</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>idct2d$row_6</td>
<td>action0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>gen</td>
<td>action0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>idct2d$row_2</td>
<td>action0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>idct2d$col_7</td>
<td>action0</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>idct2d$scale</td>
<td>action0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>idct2d$col_5</td>
<td>action0</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>idct2d$splitCol</td>
<td>action0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>idct2d$row_7</td>
<td>action0</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>idct2d$row_5</td>
<td>action0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>idct2d$col_6</td>
<td>action0</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>idct2d$col_2</td>
<td>action0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>idct2d$rsfhl</td>
<td>action0</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>idct2d$col_3</td>
<td>action0</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>idct2d$splitRow</td>
<td>action0</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>idct2d$row_0</td>
<td>action0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>idct2d$row_3</td>
<td>action0</td>
<td>2</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 9.3: Partitioning and scheduling results with the new heuristic
Chapter 10

Conclusion

This deliverable presented a design flow based on an actor and dataflow based language called CAL to design complex embedded systems. The originality of the work resides in the use of this 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 and multicore systems. The new ISO/MPEG standard, Reconfigurable Video Coding (RVC) [?], innovates by proposing to specify decoders in a modular way, using CAL. This work is also in line with the RVC standard because it provides a complete methodology for implementing decoders defined by the standard. By raising the level abstraction, the designer can design efficiently embedded systems more easily and quicker.
Chapter 11

Appendixes

1 Example of static, cyclo-static and dynamic Actors

1.1 Static actor

At each firing of the action, one token on port R, one token on port G and one token on port B are consumed and one token on port Y, one token on port Cr and one token on port Cb are produced.

```plaintext
actor RGBtoYCrCb(dynamicSZ)
  int(size=8) R, int(size=8) G, int(size=8) B
  ==> int(size=dynamicSZ) Y, int(size=dynamicSZ) Cr, int(size=dynamicSZ) Cb :
    int(size=18) yr = 306;
    int(size=18) yg = 601;
    int(size=18) yb = 117;
    int(size=18) ur = 151;
    int(size=18) ug = 296;
    int(size=18) ub = 446;
    int(size=18) vr = 630;
    int(size=18) vg = 527;
    int(size=18) vb = 102;

    function mul(int(size=18)a, int(size=18)b) : a*b end

    action R:[r],G:[g],B:[b] ==>
        Y:[rshift((yr*r)+(yg*g)+(yb*b),18-dynamicSZ)],
        Cr:[rshift((vr*r)-(vg*g)-(vb*b)+lshift(128,dynamicSZ-8),18-dynamicSZ)],
        Cb:[rshift((ub*b)-(ur*r)-(ug*g)+lshift(128,dynamicSZ-8),18-dynamicSZ)] end
end
```

1.2 Cyclo-static actor

```plaintext
actor MPEG4_algo_Interpolation_halfpel ( int PIX_SZ, int FLAG_SZ )
  int(size=PIX_SZ) RD, int(size=FLAG_SZ) halfpel ==> int(size=PIX_SZ) MOT :
    // Compensation function
    function compensate( po0, p10, p01, p11 ) :
      if flags = 0 then po0 else
          if flags = 1 then
              // interpolate y only
              rshift((po0 + p01 + 1) - round, 1 )
          else
              if flags = 2 then
                  // interpolate x only
                  rshift((po0 + p10 + 1) - round, 1 )
              else
                  // interpolate x and y
                  rshift( (po0 + p10 + p01 + p11 + 2) - round, 2 )
              end
          end
      end
```
end

end

int(size=5) x;
int(size=5) y;
int(size=3) flags;
int(size=2) round;

start: action halfpel:[ f ] ==>
do
  x := 0;
y := 0;
  flags := rshift(f,1);
  round := bitand(f,1);
end
done: action ==>
guard
  y = 9
end

int(size=PIX_SZ ) d0;
int(size=PIX_SZ ) d1;
int(size=PIX_SZ ) d2;
int(size=PIX_SZ ) d3;
int(size=PIX_SZ ) d4;
int(size=PIX_SZ ) d5;
int(size=PIX_SZ ) d6;
int(size=PIX_SZ ) d7;
int(size=PIX_SZ ) d8;
int(size=PIX_SZ ) d9;

row_col_0: action RD:[d] ==>
guard
  (x = 0) or (y = 0)
do
  d9 := d8;
  d8 := d7;
  d7 := d6;
  d6 := d5;
  d5 := d4;
  d4 := d3;
  d3 := d2;
  d2 := d1;
  d1 := d0;
  d0 := d;
  x := x + 1;
  if x >= 9 then
    x := 0;
y := y + 1;
end
other: action RD:[d] ==> MOT:[ p ]
var
  p = compensate(d9, d8, d0, d )
do
  d9 := d8;
  d8 := d7;
  d7 := d6;
  d6 := d5;
  d5 := d4;
  d4 := d3;
  d3 := d2;
  d2 := d1;
  d1 := d0;
  d0 := d;
  x := x + 1;
  if x >= 9 then
    x := 0;
y := y + 1;
end
end

64
By following the finite state machine and the assignment of state variables, one can deduce the order of execution of actions of this cyclo-static actor. The sequence of execution is the following:

1. start \( (x=0, y=0) \)
2. row\_col\_0 \( (x=1, y=0) \)
3. other \( (x=2, y=0) \)
   ...
9. other \( (x=8, y=0) \)
10. other \( (x=0, y=1) \)
   ...
11. row\_col\_0 \( (x=1, y=1) \)
   ...
18. other \( (x=8, y=1) \)
19. other \( (x=0, y=2) \)
20. row\_col\_0 \( (x=1, y=2) \)
   ...
26. other \( (x=8, y=2) \)
27. other \( (x=0, y=3) \)
28. row\_col\_0 \( (x=1, y=3) \)
   ...
34. other \( (x=8, y=3) \)
35. other \( (x=0, y=4) \)
36. row\_col\_0 \( (x=1, y=4) \)
   ...
42. other \( (x=8, y=4) \)
43. other \( (x=0, y=4) \)
44. row\_col\_0 \( (x=1, y=5) \)
   ...
50. other \( (x=8, y=5) \)
51. other \( (x=0, y=5) \)
52. row\_col\_0 \( (x=1, y=5) \)
   ...
58. other \( (x=8, y=5) \)
59. other \( (x=0, y=6) \)
60. row\_col\_0 \( (x=1, y=6) \)
66. other \( (x=8, y=6) \)
67. other \( (x=0, y=7) \)
68. row\_col\_0 \( (x=1, y=7) \)
   ...
74. other \( (x=8, y=7) \)
75. other \( (x=0, y=8) \)
76. row\_col\_0 \( (x=1, y=8) \)
77. other \( (x=8, y=8) \)
78. other \( (x=0, y=9) \)
79. other \( (x=1, y=9) \)
80. done \( (x=1, y=9) \)

After the firing of 75th action, the actor is in its initial state.

### 1.3 Dynamic actor

```c
actor MPEG4_algo_Add (  
  int PIX_SZ,  
  int MB_CAND_SZ,  
  int BTYPE_SZ,  
  // Command flags from parser  
  int NEWVOP,  
  int INTRA,  
  int ACCODED  
)  
int(size=PIX_SZ) MOT,  
int(size=PIX_SZ) TEX,  
int(size=BTYPE_SZ) BTYPE ==> int(size=PIX_SZ) VID :  
int(size=8) count := 0;

cmd.newVop: action BTYPE: [ cmd ] ——>
```
guard
  bitand( cmd, NEWVOP ) != 0
  do
    println("newvop:" + cmd);
  end

// Pure texture
  cmd.textureOnly: action BTYPE:[ cmd ] -->
  guard
    bitand( cmd, INTRA ) != 0
  do
    println("textureonly:" + cmd);
  end

// Pure motion
  cmd.motionOnly: action BTYPE:[ cmd ] -->
  guard
    bitand( cmd, ACCODED ) = 0
  do
    println("motiononly:" + cmd);
  end

// Mixed texture and motion (Also used to skip vop w,h)
  cmd.other: action BTYPE:[ cmd ] -->
end

done: action -->
  guard
    count = 64
  do
    count := 0;
  end

texture: action TEX:[ tex ] --> VID:[ tex ]
  do
    count := count + 1;
    println("texture:" + tex);
  end

motion: action MOT:[ mot ] --> VID:[ mot ]
  do
    count := count + 1;
    println("motion:" + mot);
  end

combine: action MOT:[ mot ], TEX:[ tex ] --> VID:[ if s < 0 then 0 else if s > 255 then 255 else s end end ]
  var
    int(size=PIX_SZ+1) s = tex + mot
  do
    count := count + 1;
    println("combine:" + mot + tex);
  end

schedule fsm cmd:
  cmd ( cmd.newVop ) --> skipw;
  cmd ( cmd.textureOnly ) --> texture;
  cmd ( cmd.motionOnly ) --> motion;
  cmd ( cmd.other ) --> combine;
  texture ( done ) --> cmd;
  texture ( texture ) --> texture;
  motion ( done ) --> cmd;
  motion ( motion ) --> motion;
  combine ( done ) --> cmd;
  combine ( combine ) --> combine;
  skipw ( cmd.other ) --> skiph;
  combine ( cmd.other ) --> cmd;
end

priority
  cmd.newVop > cmd.textureOnly > cmd.motionOnly > cmd.other;
  done > combine;
  done > texture;
  done > motion;
end
Guards contain condition on values of the tokens consumed at a given action. In this case, in state cmd, the actor checks the value of the BTYPE token and fires the right action:

- if the condition `bitand( cmd, NEWVOP ) != 0` is true, the cmd.newVop action is fired
- if the condition `bitand( cmd, INTRA ) != 0` is true, the cmd.textureonly action is fired
- if the condition `bitand( cmd, ACCODED ) = 0` is true, the cmd.motiononly action is fired
- if all the above conditions are not satisfied, the cmd.other action is executed.
2 Examples of Design Space Exploration applied to target platforms

This chapter aims at explaining more precisely how to explore the design space according to the targeted platform. Only three target platforms are studied: single processor (see section 2.1), single FPGA (see section 2.2) and one processor plus a FPGA (see section 2.3). Resources and throughput are the two main criteria studied. Studying this platform helps in understanding better how scheduled and partitioned CAL programs can fit on sequential processors.

2.1 Single sequential processor platform

It seems a priori quite paradoxical to study this pure sequential platform because it is claimed that the multicore era is more and more present and the Language is a good way to deal with it. But, studying this platform will help us to understand more complicated platforms, as multicore software platform or one processor plus one FPGA platform.

2.1.1 Degrees of freedom

This section presents the possibilities the designer have in order to reach the target region. The designer have the following choices:

**Merge static regions** Knowing the type of the actors and the dependencies, static regions can be detected. Existing tools are able to determine static schedule for each static region described as a SDF graph (Dataflow Interchange Format [1], SDF3 [2], SDF4J [3] . . .). After the schedule of the actions composing this static region has been established, it is possible to merge safely the actors. The order of execution of the actions is known a priori and is not dependent on input data. This merging eliminates the need of the FIFO memories between the actors and removes the overhead induced by scheduling those actors (only one actor is remaining). Moreover, memory usage may be reduced if the single actor needs a lower amount of memory than the one used for the FIFO between actors belonging to the static region.

**Rewrite actors** With the measures extracted from the characterization step, the designer can detect which actor needs an additional designing effort. Thus, the designer will concentrate on redesigning only the actors that is worth modifying. This transformation has a limited effect in the sense that it does not modify the global architecture of the model because it applies only to actors.

**Modify partitioning and/or scheduling** this operation consists in reconsidering the scheduling and/or the partitioning in order that it fits better with the requirements. This operation does not appear on figure [1] because the results of this operation can move the design point in any direction.

Figure [1] illustrates the degrees of freedom for the exploration of design space in case of a single sequential processor platform.

2.1.2 Criteria: maximizing performances

Figure [2] illustrates a path in the design space induced by a unique optimization criterion, throughput. Starting from an initial CAL program which yields, after the
evaluation, a value below the required bound in terms of throughput, a first step can be achieved by removing as much as possible overhead due the multiple actors in the model. Remember that the scenario is a single processor, thus, exposing too much parallelism is not necessary. This first operation consists in merging actors to reduce the exposed parallelism. As the CAL program does still not fill the requirement, a second operation is necessary. Some actors are rewritten in order to optimize their inner algorithm. After the evaluation of this second CAL program, the target region has been reached.

2.1.3 Criteria: minimizing resources with lower bound on performances

Figure 3 illustrates a path in a 2D design space (throughput vs software resources) in order to reach the target region defined by the designer. From the initial SPCP, the designer merges some (partial) static regions (1), optimizes the scheduling (3), merges again (partial) static regions (1) and finally rewrites some CAL actors (2).

2.2 Single FPGA platform

Studying this platform is very useful to understand how CAL program can fit on pure parallel platform such as FPGA.

2.2.1 Degrees of freedom

In case of a FPGA platform, the designer can perform the following transformations in order to explore efficiently the design space and to reach the target region as fast as possible.
Split actors the aim is to expose better the task and data parallelism. Splitting actors increases the necessary resources in the FPGA, but increases the throughput.

Merge actors Conversely, in order to minimize hardware resources, merging simple actors into larger actors can be an effective solution. Merging actors reduces the overhead induced by the handling of tokens (FIFOs and glue logic). Usually, it implies a slight reduction of performance but this is acceptable if the bound is still satisfied.

Delete Finite State Machine (FSM) The aim is reduce the necessary glue logic which must be implemented in hardware in order to deal with FSM.

Optimize actions This operation consists in reducing the critical path of this action, to make it shorter. It lowers the utilized resources and may increase the throughput.

Splitting or merging actors can be guided by the results of the computational load of the actors and the critical path of the program. If an actor has a large critical path and has a large computational load, it indicates that this actor must be split.

Figure 4 illustrates the degrees of freedom for the exploration of design space by considering only performances, i.e. throughput.

Figure 5 illustrates the degrees of freedom for the exploration of design space by considering resources and performances.
2.2.2 Criteria: maximizing performances

Maximizing the throughput in case of a single FPGA consists of transforming the actors to smaller and simpler actors in order to expose and exploit a higher level of task and data parallelism. Deleting FSM and reducing the critical path of the actions aims also at maximizing the performances. The FPGA is a completely parallel platform capable of handling highly parallel models.

2.2.3 Criteria: minimizing resources with lower bound on performances

The aim is to minimizing hardware resources while guaranteeing a lower bound on performances. The designer can:

Reduce the intrinsic critical path and complexity of the actors it reduces the number of operators generated.

Delete useless FSM it reduces the generated for the glue logic.

Merge actors it reduces the generated logic for handling the communication between actors (FIFOs, ports ...). The designer must be careful not to downgrade the throughput by reducing task or data parallelism.

Figure 6 illustrates an example of design space exploration. The designer performs three transformations: first, he splits actors (1), increasing the hardware resources and the throughput. The required throughput is satisfied but not the re-
resources requirement. Thus, the designer merge actors (2) that are not critical for the speed of execution (typically, actors that have a low contribution to the critical path). As he is close to the target region, he simplifies some FSM.

Figure 7: CAL transformations to minimize resource usage targeting a single FPGA

2.3 One processor and one FPGA platform

The study of the two previous platform helps in understanding and handling this platform composed of a sequential processor and a FPGA platform.

2.3.1 Degrees of freedom

The designer has more and more possibilities as the number of processing elements increases. And that is really the problem with multicore platform. How to manage the algorithm and to partition it so that it runs as fast as possible with the less resources.

The designer still have the transformations that have been discussed in the last two sections (2.1 and 2.2). These transformations are represented in figures 4, 5 and 1.

A new aspect appears when dealing with several processing units: the partitioning of actors onto processing elements. Figure 8 illustrates what are the implications in terms of performances and resources of moving actors from hardware to software and vice versa, taking under account both performance and resources.
2.3.2 Criteria: maximizing performances

In order to maximize the throughput, the designer can split actors assigned to hardware and merge actors assigned to software. Merging actors assigned to software aims at reducing the overhead induced by the CAL MoC, i.e. the choice of the next action to fire. Splitting actors assigned to hardware aims at increasing the level of exposed parallelism which can be efficiently exploited by the FPGA.

Figure 9 illustrates an example of design space exploration.

2.3.3 Criteria: minimizing resources with lower bound on performances

Minimizing the resources in this case implies for example reducing the memory use concerning the actors assigned to software and reducing the area used by the actors assigned to hardware.

When considering a sub-part of the network which is entirely assigned to hardware or software, one can refer to section 2.2 for hardware and to section 2.1 for software for the CAL program transformations allowing the exploration of the design space.

The resulting throughput of the entire system is equal to the throughput of the slowest partition. Consequently, making a transformation in the hardware or soft-
A hardware part may influence the global performance. For example, the modification of the schedule of actions onto the processor may increase or decrease the throughput. It results in moving the design points both in hardware and software. When moving an actor from hardware to software (or the contrary), the properties of the two partitions change. Consequently the two design points in hardware and software moves in the design space. This aspect illustrated in figure 8.

Figure 10 illustrates an example of design space exploration when considering an heterogeneous platform and by taking under account resources usage and performances. The hardware transformation (1) moves the design point $h_1$ to $h_2$. Thanks to this modification, the throughput of the system is increased. Thus, the design point $s_1$ moves to $s_2$. Then, a software transformation (2) is applied in order to move the partition region. It moves the design point from $s_2$ to $s_3$. It modifies the throughput, thus moving the point in hardware from $h_2$ to $h_3$. All the transformations are applied following this principle in order to reach the target region (the points $h_5$, $h_6$ and $s_6$ are in the target region).

Figure 10: One processor and one FPGA scenario and refactoring

1 - Split complex and critical HW actors/actions into simpler actors/actions
2 - Merge Statically Schedulable Regions (SSR) in SW
3 - Merge non critical and simple HW actors
4 - Simplify FSM and re-arrange HW actions
5 - Rewrite SW actors so that it minimizes the memory use
3 Results of the case study

Table 11.1 relates to the first analysis of the CAL program (section 2).

Table 11.1: Results of the first analysis of the CAL program

<table>
<thead>
<tr>
<th>Action</th>
<th>Actor</th>
<th>Characterization</th>
<th>Weighting</th>
<th>Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Avg. CL</td>
<td>Total CL</td>
<td>Calls</td>
</tr>
<tr>
<td>action0</td>
<td>gen</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_0$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_0$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_0$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_0$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_1$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_1$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_1$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_1$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_2$even</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_2$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_2$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_2$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_3$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_3$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_3$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_3$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_4$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_4$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_4$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_4$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_5$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_5$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_5$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_5$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_6$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_6$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_6$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_6$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_7$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_7$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_7$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$col_7$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$joinCol</td>
<td>534</td>
<td>5340</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$joinRow</td>
<td>534</td>
<td>5340</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$retranspose</td>
<td>652</td>
<td>6527</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$row_0$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$row_0$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$row_0$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$row_0$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$row_1$seven</td>
<td>28</td>
<td>285</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$row_1$odd</td>
<td>40</td>
<td>405</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$row_1$sort</td>
<td>59</td>
<td>590</td>
<td>10</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d$row_1$unsort</td>
<td>15</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>Action</td>
<td>Actor</td>
<td>Characterization</td>
<td>Weighting</td>
<td>Partitioning</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>------------------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Avg. CL</td>
<td>Total CL</td>
<td>Calls</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_2</td>
<td>even</td>
<td>28</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_2</td>
<td>odd</td>
<td>40</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_2</td>
<td>sort</td>
<td>59</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_2</td>
<td>unsort</td>
<td>15</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_3</td>
<td>even</td>
<td>28</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_3</td>
<td>odd</td>
<td>40</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_3</td>
<td>sort</td>
<td>59</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_3</td>
<td>unsort</td>
<td>15</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_4</td>
<td>even</td>
<td>28</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_4</td>
<td>odd</td>
<td>40</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_4</td>
<td>sort</td>
<td>59</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_4</td>
<td>unsort</td>
<td>15</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_5</td>
<td>even</td>
<td>28</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_5</td>
<td>odd</td>
<td>40</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_5</td>
<td>sort</td>
<td>59</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_5</td>
<td>unsort</td>
<td>15</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_6</td>
<td>even</td>
<td>28</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_6</td>
<td>odd</td>
<td>40</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_6</td>
<td>sort</td>
<td>59</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_6</td>
<td>unsort</td>
<td>15</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_7</td>
<td>even</td>
<td>28</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_7</td>
<td>odd</td>
<td>40</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_7</td>
<td>sort</td>
<td>59</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>row_7</td>
<td>unsort</td>
<td>15</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>rshift</td>
<td></td>
<td>471</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>scale</td>
<td></td>
<td>505</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>splitCol</td>
<td></td>
<td>468</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>splitRow</td>
<td></td>
<td>468</td>
</tr>
<tr>
<td>action0</td>
<td>idct2d</td>
<td>transpose</td>
<td></td>
<td>652</td>
</tr>
<tr>
<td>action0</td>
<td>print</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Bibliography


[pto] Ptolemy project: heterogeneous modeling and design at UC berkeley, EECS. http://ptoley.eecs.berkeley.edu/.