Optimization of Real-Time Application on Mixed Architecture Using AAA Methodology Extension

O. Feki, T. Grandpierre, N. Masmoudi, M. Akil and Y. Sorel

Abstract

Mixed Architectures contain programmable devices and reconfigurable devices. They provide a powerful answer to meet the computational requirement of latest digital signal processing applications. But the complexity of the corresponding algorithms and the multiplicity and diversity of computing components usually lead to a huge number of possible implementations. Architecture Algorithm Adequation (AAA) is one of the rapid prototyping methodologies which allow to explore the solution set to build an optimized application. However, it requires improvement in order to support mixed architectures containing both programmable and configurable components. This paper suggests an extension of the AAA methodology to support mixed architectures. The AAA architecture model is first extended to mixed architecture. Then, we present the coupling of existing tools (SynDEx and SynDEx-IC) in order to support mixed architectures. Finally a communication IP is proposed to manage FPGA communication and its synchronization with other components of the mixed architecture.

Key Words

— rapid prototyping, AAA, co-design, real-time, H264.

I.

INTRODUCTION

Digital signal processing uses more and more complex algorithms to perform better and innovative functionalities. This complexity growth goes hand in hand with the increasing computing demand. Unfortunately, single-processor systems cannot meet this demand. Therefore system designers use multi-component heterogeneous systems containing different types of computation components interconnected by different types of communication media. These computation components can be programmable circuits (DSP, RISC, CISC ...) or reconfigurable circuits (FPGAs …). In the following, we refer to architectures which contain programmable devices and reconfigurable devices as mixed architectures. Optimizing algorithm implementation on these mixed architectures means finding a compromise between temporal performances and used logic blocks number. Indeed, increasing parallel computation leads to latency decrease in return of used resources increase.

Using Mixed Architectures for complex algorithms leads to a huge number of possible implementations which requires long time to be explored. Consequently, rapid prototyping methodologies and their associated tools must be used to obtain an efficient implementation in the least possible time. Among these tools, CoFluent [1] and PtolemyII [2] allow complex architecture modeling and functional simulation, ASTEX [3] permits repetitive extraction of codelets whose execution on coprocessors can accelerate application execution time. Diamond 3L [4] permits code generation for applications on mixed architectures based on TI DSPs and Xilinx FPGAs. It also generate necessary communications. Bourgos et al. [5] propose a method to model algorithms and multiprocessor architectures for the sake of simulation only. Finally, none of mentioned tools perform automatic partitioning or optimizations on the target architecture. On the contrary, Gedae [6] and DOL [7] perform automatic scheduling but only on multiprocessor architectures. ECOS [8] allows scheduling minor complexity algorithms on mixed architectures, but it cannot deal with real life algorithms. CODEF [9] is a tool which permits scheduling algorithms on mixed architectures, but the description that it uses is fine-grained, which requires much time to perform specifications and scheduling. The Algorithm Architecture Adequation (AAA) [10] methodology targets multiprocessor architectures. It was previously extended to target reconfigurable architecture but it does not yet cover the mixed architectures.

This paper presents our AAA methodology extension and is organized as follows: section II presents the AAA methodology foundation. It presents the models used to specify the application algorithm, programmable architectures and reconfigurable architectures. It also exposes optimization heuristics and software tools supporting AAA methodology (SynDEx and SynDEx-IC). Section III, is dedicated to our AAA methodology extension. It presents the AAA models extension and our new software tool based on the coupling of SynDEx and SynDEx-IC. It finally details the automatic synthesis of communication between programmable and reconfigurable parts of the architecture. Section IV describes an actual concretization of the obtained result and in section V a conclusion is provided and perspectives are drawn.
AAA is a rapid prototyping methodology that has been developed to help real-time system designers implement and optimize their algorithms on programmable and reconfigurable components. It is devised for programmable and ASIC components. It is a formal methodology based on graphs transformations in multicomponent architectures. AAA is a formal methodology for real-time system design that allows the implementation of their algorithms on programmable and reconfigurable components. The left side of figure 1 shows the design flow of SynDEx-IC that targets a single FPGA architecture [12].

The SynDEx software tool [10] has been implemented to help real-time system designers implement and optimize their algorithms on programmable and reconfigurable components. This software tool allows the implementation of the algorithm graph on the architecture. The SynDEx-IC optimizes the algorithm graph by executing an off-line optimization heuristic which distributes and schedules the algorithm graph on the architecture.

In this section, we have presented the algorithm graph computing operations by a single pattern and marking each edge of the set of edges connecting them [12]. Each vertex of the graph (O, D) where O is the set of vertices and D is the set of edges connecting them.

AAA is an extension for reconfigurable ones that are based on the grayed part in figure 1-b exposes the proposed design environments.

SynDEx-IC is an algorithm graph model common to SynDEx and SynDEx-IC. In the following, we present SynDEx and SynDEx-IC.

The factorization feature of AAA allows the simplification of their presentation. Factorization is the reduction of the execution time and the number of memories. It can optionally contain communicators. By analogy with the finite repetition of a sequence acquisition-computation-action, we introduce the infinite factorization frontier which is a border that includes the entire algorithm (the sequence acquisition-computation-action).

In this paper, we target mixed architecture made of programmable and reconfigurable circuits. This extension has led to a specific extension for the deployed methodology that supports the scheduling real-time application on heterogeneous multi processor systems using the presented algorithm graph.

In this section, we have presented the algorithm graph computing operations by a single pattern and marking each edge of the set of edges connecting them [12]. Each vertex of the graph (O, D) where O is the set of vertices and D is the set of edges connecting them.

AAA addresses reactive real-time systems that react to any intermediate implementation. This model is based on a graph (S, A) where S is the set of vertices and A the set of edges connecting them. Each element of S is a finite state machine that models a DMA channel. Then a processor is modeled by one or more operators connected to one or more memories (used for storing data). The operator models computing operations, communicators (executing data transfer operations), bus/mux/demux and arithmetic and logical units.

The set of vertices models a system where each element of the set models a data producer. A data consumer is modeled by one or more operators connected to a processor, memories, the system's environment and another consumer.

As an example, figure 3-a depicts a mono-processor architecture graph examples.
algorithm (fig 5-b) and architecture graphs (fig 5-a) of the execution of data transfers (send and receive) in order to exhibit the allocation of memories and the added during the distribution and scheduling process of the architecture. “Alloc”, “R” and “S” nodes are scheduling of algorithm’s operations on each operator graph. This latest graph highlights the distribution and architecture graph, and a possible implementation figure 4 depicts respectively an algorithm graph, an implementations graph. The top, center and bottom of there are a large but finite number of possible implementations graph. The transformed algorithm graph is called an transfer) is added to the algorithm graph. This corresponding allocation node (memory allocation, (memory, bus/mux/demux, communicator), a graph. Once one is selected, on each node of the path two operators must be chosen in the architecture accomplish this data transfer, a path connecting these operator that will execute the consuming one. To operator that executes the producer operation to the same operator, the data must be transferred from the two data dependent operations are not executed by the executed before the data consumer operation. When data dependence, the producer operation must be following subsection). When two operations are in them must be chosen to execute it (as explained in the architecture graph able to perform it. If several operators are able to perform an operation, one of operation; and for every operation of the algorithm graph there must be at least one operator of the operation O and its biggest starting date which does not involve critical path length raise. After that, the heuristic list is updated by removing the scheduled operation O between the smallest possible starting date of the operation O and its end date. This operator is designated as the best operator of the operation. Among this list, we choose to schedule the operation O operation end date. This operator is designated as the associate the best operator which minimizes the cost function σ(O). We use heuristics. AAA heuristic is based on a greedy complete. To avoid exploring the whole solution set, this optimization problem is known to be NP-unscheduled operation of algorithm graph which has maketime efficient systems using the least possible resources. The set of possible implementations is huge make time efficient systems using the least possible resources. The set of possible implementations is huge.
fork factorization node, a demultiplexer for the arithmetic operation node, a multiplexer for the either a hardware component of a VHDL library for algorithm graph vertex type. These operators can be asserts that there is an operator type for each data dependency by an operator connection. This vertex of the algorithm graph by an operator and each automatically generated: data path and control path.

2)

constraint.

this cost function is selected, the frontier deleted from factorization degree). The couple that most minimizes for each couple (factorization frontier, optimal time constraint. Finally a cost function is calculated minimum critical path length or one lower than the frontier of this list, the heuristic compute the "greatest which are the ones on the critical path. Then for each unroll. This heuristic use a list of candidate frontiers temporal constraint, a dedicated optimization heuristic graph, the characteristics of a targeted FPGA and a algorithm latency is reduced in exchange of increasing hardware resources. Given an algorithm architectures. Therefore, it was extended in order to solve the hardware implementation optimization problem in the case of a unique FPGA target systems. This extension aims at building an architecture [12]. This extension is offered by the FPGA so that it could be used by architecture model. It is the same on both side reconfigurable components thanks to loop unrolling. implementation of operations scheduled on programmable and reconfigurable components. allows to target mixed architectures made of both programmable and reconfigurable components.

AAA methodology extension to reconfigurable circuits: Code generation

AAA circuit : Code generation

AAA circuit : Optimization

AAA  circuit  supporting  tool: SynDEx-IC

For the classical synthesis tools (ISE, Quartus …) to generate synthesizable VHDL. Finally, the user uses it allows to run the defactorization heuristic and to target FPGA proprieties and the time constraint. Then allows the user to specify the algorithm graph, the software tool SynDEx-IC [14]. Its graphical interface reconfigurable circuits is implemented in the CAD system designer constraints, SynDEx can generate the corresponding executives. If not, the user the heuristic constraints or to modify the graphs.

Graphical interface allows the user to add some functions necessary to generate control signals for the multiplexers, the registers and the other circuits of the system.
electric lines and fuses that can be connected or disconnected during the configuration phase. The configurable logic blocks are modeled by a set \( S_{OPRe} \) of vertexes named elementary operators \( \text{OPRe} \). RAM blocks are modeled by a set of vertex \( S_{reg} \) of registers. The programmable network is modeled by a network \( R \) of edges. Therefore, an unconfigured FPGA is modeled by a graph \( G_{FPGA} = (S_{OPRe} \cup S_{reg}, R) \).

FPGA configuration is performed in two steps by synthesis tools: configuration of the logic blocks, and configuration of the connection network. Starting with an unconfigured FPGA, the configuration process transforms elementary operators and registers into a set \( S_{OPRd} \) of degenerate operators (OPRd): a degenerate operator is an operator able to perform only one operation of the algorithm graph. It is made of a set of configured elementary operators and RAM memories connected by a configured connection network. In the following, the function \( \rho \) models the configuration applied to an unconfigured FPGA.

\[
\rho: S_{OPRe} \cup S_{reg} \cup R \\
\rho: S_{OPRe} \cup S_{reg} \cup R \\
\]

with \( S_{OPRe}' \subseteq S_{OPRe}, S_{reg}' \subseteq S_{reg} \) and \( R' \subseteq R \).

\[ S_{OPRe}' \cup S_{OPRe}'' = S_{OPRe}, S_{reg}' \cup S_{reg}'' = S_{reg}, R' \cup R'' = R \]

where \( S_{OPRe}', S_{reg}' \) and \( R' \) correspond to respectively the set of elementary operators, the registers, and the part of the interconnection network used to obtain the set of degenerate \( S_{OPRd} \). \( S_{OPRe}'' \), \( S_{reg}'' \) and \( R'' \) correspond to the set of elementary operators, registers, and the part of the interconnection network still not configured. Figure 6-a presents a basic example of algorithm graph made of two operations A and B which have to be implemented on the unconfigured FPGA of figure 6-b. Finally, figure 6-c presents the configured FPGA. In this last figure the configuration process builds two degenerated operators “OPRd1” and “OPRd2” that will respectively able to execute operations A and B. We can observe that “OPRd1” relies on the elementary operator \( \text{OPRe}1 \) and \( \text{OPRe}2 \), while “OPRd2” relies on \( \text{OPRe}7 \) and \( \text{OPRe}8 \). Finally, the interconnection network configuration (bold red edges of figure 6-c) connects the degenerated operators.

This model extension enables the use of classical AAA distribution and scheduling processes on a FPGA.

B. SynDEx/SynDEx-IC heuristics coupling

The previous models are used internally by SynDEx. From the user side, the graphical interface allows the user to specify the algorithm graph \( G_{AL} \) and the architecture graph \( G_{AR} \). \( G_{AL} \) is the set of the operations \( (O_1, O_2, \ldots, O_q) \) and data dependences of the application algorithm. \( G_{AR} \) is the set of programmable operators \( (\text{opr}_1, \ldots, \text{opr}_n) \), FPGAs \( (\text{FPGA}_1, \ldots, \text{FPGA}_m) \) of the target architecture, including the communication mediums connecting them \( (\text{com}_1, \ldots, \text{com}_p) \). At this point it is important to notice that each FPGA is simply specified using a single node. For example, figure 7-a shows an algorithm graph example. This graph is made of 9 operations among which operations C, F, and G encapsulate repetitions.

(a) Algorithm graph

(b) Architecture graph

Figure 7-b shows the initially specified architecture graph. This architecture graph is made of 2 operators: one programmable (processor) and one reconfigurable...
The user has to fill a table (table 1) that specifies the worst case execution duration of each operation on each operator able to execute it. One can see that for this simple example, the processor is able to perform all operations of the algorithm graph while the FPGA is able to perform only the operations C, F and G. For the FPGA theses duration correspond to a fully sequential execution (factorized graph).

Table . initial operation duration on each operator

<table>
<thead>
<tr>
<th>Processor</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
</tr>
<tr>
<td>G</td>
<td>8</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
</tbody>
</table>

Since SynDEx does not consider FPGA potential parallelism we need to expose this parallelism in SynDEx before we can couple heuristics. For this purpose we apply the algorithm 1 to the couple of input graph (algorithm and architecture). It transforms the architecture graph by replacing each FPGA of the initial architecture graph by the set of degenerate operators it may contain: the FPGA node is deleted from Gal, and for each operation schedulable on the FPGA, a degenerate operator is added. This exhibits explicitly the internal FPGA parallelism.

Inputs: G\text{AR} and G\text{AL}

Outputs: transformed G\text{AR}

\begin{align*}
1: & \quad \text{for each FPGA } i \in G\text{AR} \\
2: & \quad \text{FPGA } i' = \{\} \\
3: & \quad \text{for each FPGA connection with other components} \\
4: & \quad \text{FPGA } i' = \text{FPGA } i' + \{\text{com-IP}\} \\
5: & \quad \text{end for each} \\
6: & \quad \text{for each operation } O \in \mathcal{P}' - 1 (\text{FPGA } i) \cap G\text{AL} \\
7: & \quad \text{FPGA } i' = \text{FPGA } i' + \text{OPRd}(O) \\
8: & \quad \text{end for each} \\
9: & \quad \text{FPGA } i' = \text{FPGA } i' + \text{com internal} \\
10: & \quad \text{FPGA } i = \text{FPGA } i' \\
11: & \quad \text{end for each} \\
\end{align*}

Algorithm 1: heuristic for mixed circuit

This transformation applied to the architecture graph given figure 7-b would produce the architecture graph given figure 8 with the 3 degenerative operator nodes “OPRd_C”, “OPRD_F” and “OPRd_G”. It is also important to notice that a communication node “COM_IP” is inserted between the FPGA and these new degenerative operator nodes. This communication node models the serialization of the FPGA communication with the other components of the architecture graph (inter-FPGA communications). Finally, we model the configurable connection network of the FPGA (intra-FPGA communications) by a multi point SAM with no cross time for all data types: setting a cross time to zero is effectively equivalent to a completely connected network.

Fig . architecture graph after transformation

Thanks to this preliminary architecture graph transformation, SynDEx can now take into account the internal parallelism offered by the FPGA. In our extension, the distribution and scheduling is performed in 2 steps.

Step 1: SynDEx its optimization heuristics to distribute and schedule the algorithm graph operations on the operators of the transformed architecture. At this level, factorization frontiers are scheduled sequentially even on FPGA. Applied to figure 7-a algorithm and figure 8 architecture graphs, SynDEx build the implementation graph of figure 9 (operators and communication are displayed horizontally on the top, while operations scheduled on each operators and media are displayed vertically).

Fig . implementation graph after G\text{AR} transformation

Step 2: SynDEx-IC heuristic is use to optimize the implementation of sub graphs distributed on each FPGA operator, but it only deal with operations that belong to the critical path of the implementation graph. As previously explained, SynDEx-IC will search for the optimized defactorization degree for each of these operations. This optimization is done in two steps:

a) Execution time minimization: SynDEx-IC will unroll all factorization frontiers. To do so, for each sub graph we take the proprieties of the FPGA on which it is distributed. Those FPGA proprieties are the execution time and the used surface for each operation it can perform. They are specified by the
user  when  defining  the  architecture  graph.  Given
FPGA  proprieties,  the  sub  graph  and  the  time
constraint (C=1), SynDEx-IC calculates the smallest
execution time of the sub graph. Figure 10 presents
the  resulting  implementation  graph  of  the  figure  7
example.

Fig . Implementation graph after execution time optimization

b)  Space  minimization:  this  step  focus  on
CLB  reduction  without  increasing  the  critical  path
(CP) length. So doesn't optimize space for sub graphs
containing operations on the CP. So each sub graph
previously  optimized

is  considered  as  a  single
operation "OP" if it does not contain operations
on
the

critical  path.  Duration  of  this  operation  is  denoted
"d
OP
". Then SynDEx is used to compute calculate cost
function named "schedule flexibility" [15]: SF

OP
it is
the difference between
its latest
start date which
does
not extend
critical path
and its earliest start date. This
value is send back to SynDEx-IC which compute a
new temporal  constraint  TC

OP
= d
OP
+ SF

OP
and  search
for  the  optimize  defactorization  degree.   This
temporal constraint guarantees that  the critical path
length  does  not  increase.  The  final  implementation
graph  corresponding  to  the  example  is  shown  on
figure 11. We notice that C execution duration was
increased due to space optimization without affecting
the total execution duration.

Figure . Final implementation time

At this level, the optimization process is done and
the  final  implementation  graph  can  be  used  by
SynDEx  to  generate the executives while each  sub
graphs are transformed into VHDL codes by SynDEx-
IC.  However,  SynDEx-IC  does  not  still  generate
components  to  manage  communication  and
synchronization with programmable parts. This is the
objective of the next section.

C. Communication IP generation

As presented in figure 1, SynDEx not only perform
the distribution and scheduling of the application, it
also  generate  executives  for  the  multi  processors
architectures. This includes the code of each operator
but  also  the  necessary  communication  and
synchronization. On the other side, since SynDEx-IC
is devoted to mono-FPGA optimization, it is only able
to generate VHDL to a stand-alone component (i.e.
with  no  communication).  Then  we  should  create  a
circuit  to  handle  the  communication  and
synchronization between FPGA and programmable
components.  During  distribution/scheduling
operation, SynDEx adds to the algorithm graph the
necessary  communication  operations.  These
communication  operations  are  associated  to
communication medium vertexes of the architecture
graph.  On  programmable  components  connected  to
such  communication  medium  vertex  side,  these
scheduled communication operations are transformed
to a sequence of data transfer functions call. On FPGA
side, the same sequence is used to configure a generic

circuit that we present now.

The communication IP, presented in figure 12, is
used for every
connection
between
an
FPGA
and any
other component.  It is a degenerative operation able
to send or receive data to or from other components in
synchronization  with  operations  executed  on  the
FPGA side.  This  operation  IP is  composed  of  two
parts: a data path and a control unit.

The data path consists of a multiplexer to specify
the transfer orientation and registers. Unlike sent data,
the registration of received data is needed to maintain
a  steady  signal  at  the  operator's  inputs.  Then  a
multiplexer  is  necessary to  switch  data  to  be sent.
Command signals for the multiplexer and the registers
are generated by the control unit.

The  control  unit  is  made  up  of  a  finite  state
machine,  package  and  data counters,  a comparator,
sent and received package counters, a multiplexer for
received  data  register  commands,  a  ROM  memory
(see  hereunder)  and  synchronization  unit
"oprd_synch"  with  OPRd  blocks.   The  FSM
synchronizes the CPU communicator with the FPGA
communicator.  It  has  seven  input  ports:  the  clock
(Clk)  and  initialization  (init)  ports,  three  ports  for
synchronization with OPRds and the CPU:
req_f_oprd,
req_f_cpu,
ack_f_cpu,
ack_t_cpu
and
req_t_cpu
for synchronization with CPU,
wr
for data
writing enabling and two output ports to command the
package counter (start
to increase the value and
reset
to reset it). FSM operates as shown in figure 13 and
output signals for each state are regrouped in table 2.

Fig : FSM functioning
<table>
<thead>
<tr>
<th>State</th>
<th>Output ports values</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>wr= 0, start= 0, reset = 0, req_t_cpu = 0, ack_t_cpu=0</td>
</tr>
<tr>
<td>receive1</td>
<td>wr= 1, start= 1, reset = 0, req_t_cpu = 0, ack_t_cpu=1</td>
</tr>
<tr>
<td>receive2</td>
<td>wr= 0, start= 0, reset = 0, req_t_cpu = 0, ack_t_cpu=0</td>
</tr>
<tr>
<td>reset</td>
<td>wr= 0, start= 0, reset = 1, req_t_cpu = 0, ack_t_cpu=0</td>
</tr>
<tr>
<td>send1</td>
<td>wr= 0, start= 0, reset = 0, req_t_cpu = 1, ack_t_cpu=0</td>
</tr>
<tr>
<td>send2</td>
<td>wr= 0, start= 1, reset = 0, req_t_cpu = 0, ack_t_cpu=0</td>
</tr>
</tbody>
</table>

The ROM is built from the scheduled sequence of communications performed by SynDEx during the distribution process. It contains at the $i^{th}$ address the packet number in the $i^{th}$ data coded on $n$ bits. The $(n+1)^{th}$ bit is used to indicate a send or a receive operation. And the $(n+2)^{th}$ bit is used to store acknowledgement or request sent to the OPRd. The synchronization with OPRd bloc has five input ports: synchro and in_out connected respectively to the $(n+2)^{th}$ and the $(n+1)^{th}$ bit of the ROM, en indicating the end of current data transfer and the req_f_oprd and ack_f_oprd. It has two output ports which are req_t_oprd and ack_t_oprd. If synchro = 1 and in_out = 0 then in rising edge of en, req_t_oprd = 1, this value is maintained until the acknowledgment from OPRd (req_f_oprd) is received. If synchro and in_out are equal to 1, then on en rising edge, the ack_t_oprd = 1 and this value is maintained until req_f_oprd = 0. To allow data transfer by packets, each packet is registered separately. Then we use the demultiplexer to switch the write (wr) signal to the appropriate register. The control signals for this demultiplexer and the send_pck_mux are the outputs of two counters which count respectively received and sent packages.

Fig. Communication IP circuit

IV. VALIDATION

A. Description
In order to validate our optimization heuristic on a mixed architecture, we present the implementation of intra 16x16 prediction decision of H.264 video encoding algorithm [16]. This decision is made by comparing different predicted blocks using different prediction modes and the original frame block. The closest resembling predicted block is chosen. The most used graphic comparison criterion is the SAD (Sum of Absolute Difference); its equation (3) shows that it consists on a repetition of subtractions, absolute value and accumulation. Since all the iterations of the SAD operation are independent, they can be executed in parallel on a FPGA.

(3)
The three outputs of this algorithm are the DC value (DC_val), the value of the minimum SAD found (min_SAD) and its correspondent mode (best_mode).

This algorithm is based on five operations:

• Calc_dc_val: calculates the mean of the 32 neighbor values.
• SAD_dc: calculates the SAD of the DC mode prediction.
• SAD_v: calculates the SAD of the vertical mode prediction.
• SAD_h: calculates the SAD of the horizontal mode prediction.
• SAD_comparator: compares the three calculated SADs and gives the smallest value and its correspondent mode (0 if vertical mode, 1 if horizontal mode and 2 if DC mode).

The target architecture is composed of a processor CPU and an FPGA connected through a bidirectional bus.

The corresponding architecture graph is given in figure 15.

Figure target architecture graph

The execution duration of each operation of the algorithm graph on architecture graph operators is shown on table 3. Each operator can only execute operations for which the duration of execution is given.

Table: operation duration on each operator

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>TOP</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>LEFT</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Calc_DC_val</td>
<td>33</td>
<td>18</td>
</tr>
<tr>
<td>SAD_V</td>
<td>768</td>
<td>296</td>
</tr>
<tr>
<td>SAD_H</td>
<td>768</td>
<td>296</td>
</tr>
<tr>
<td>SAD_DC</td>
<td>768</td>
<td>296</td>
</tr>
<tr>
<td>SAD_comparator</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>DC_val</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Min_SAD</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Best_mode</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

B. Schedulig using current SynDEx version

When we execute the initial (not modified) optimization heuristic of SynDEx on these graphs, we get the following results (table 4).

Table: sheduling result without architecture graph transformation

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>0 - 256</td>
<td></td>
</tr>
<tr>
<td>TOP</td>
<td>256 - 272</td>
<td></td>
</tr>
<tr>
<td>LEFT</td>
<td>272 - 288</td>
<td></td>
</tr>
<tr>
<td>Calc_DC_val</td>
<td>288 - 321</td>
<td>18</td>
</tr>
<tr>
<td>DC_val</td>
<td>321 - 322</td>
<td>18</td>
</tr>
<tr>
<td>SAD_H</td>
<td>322 - 1090</td>
<td>296</td>
</tr>
</tbody>
</table>

Measures can be obtained automatically using SynDEx/SynDEx-IC chronometer feature.
Numerical values in table 4 correspond respectively to the as soon as possible start and end dates for every operation. The column containing these dates corresponds to the operator executing the corresponding operation. We notice that operation SAD_V starts at 874 while the SAD_DC operation ends at 874. This suggests that these two operations are scheduled sequentially on FPGA when they can be executed simultaneously to optimize more the whole algorithm latency. The critical path length for this solution is 1211. This implementation does not consider the parallelism offered by FPGA.

### C. Scheduling using the architecture graph transformation

Now, if we apply the architecture graph transformation (as explained in the beginning of section III-B p.9), we obtain the architecture graph shown on figure 16: each computing operation that has execution duration on the FPGA (table 3) has lead to a new degenerated operator. Those degenerated operators associated to the communication operator and the medium connecting them form a sub graph (enclosed with the red box "FPGA") modeling the FPGA.

![Transformed architecture graph](image)

Then on this new architecture graph, the SynDEx optimization heuristic compute the following distribution/scheduling giving the execution dates sum up in table 5.

<table>
<thead>
<tr>
<th>CPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>0 - 256</td>
</tr>
<tr>
<td>TOP</td>
<td>256 - 272</td>
</tr>
<tr>
<td>LEFT</td>
<td>272 - 288</td>
</tr>
<tr>
<td>Calc_DC_val</td>
<td>288 - 321</td>
</tr>
<tr>
<td>DC_val</td>
<td>321 - 322</td>
</tr>
<tr>
<td>SAD_H</td>
<td>578 - 874</td>
</tr>
<tr>
<td>SAD_DC</td>
<td>578 - 874</td>
</tr>
<tr>
<td>SAD_V</td>
<td>578 - 874</td>
</tr>
<tr>
<td>SAD_comparator</td>
<td>877 - 915</td>
</tr>
<tr>
<td>Best_mode</td>
<td>915 - 916</td>
</tr>
<tr>
<td>Min_sad</td>
<td>916 - 917</td>
</tr>
<tr>
<td>Total latency</td>
<td>917</td>
</tr>
</tbody>
</table>

For the sake of simplicity in this paper, we do not specify the degenerated operator executing each operation distributed on FPGA, but it is implicitly known since each degenerated operator can perform only one operation type. In this solution, the three SAD operations are executed in parallel (they have the same start and end dates) on FPGA. Thus, the critical path length is reduced from 1211 to 917. Each one of the three SAD operations constitutes one sub graph distributed on FPGA.

### D. Latency optimization using SynDEx-IC

As explained in step 2-a of page 10, we use SynDEx-IC to optimize the implementation of three SAD operations scheduled on the FPGA. The factorization frontiers contained by these SAD operations are fully defactorized. The new execution duration of the three SAD operations is 41. The new start and end dates of all operations on algorithm graph are shown on table 6.
### Table. Scheduling result after time optimization

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>0 - 256</td>
<td>TOP 256 - 272</td>
</tr>
<tr>
<td>LEFT</td>
<td>272 - 288</td>
<td></td>
</tr>
<tr>
<td>Calc_DC_val</td>
<td>288 - 321</td>
<td></td>
</tr>
<tr>
<td>DC_val</td>
<td>321 - 322</td>
<td></td>
</tr>
<tr>
<td>SAD_H</td>
<td>578 – 619</td>
<td>SAD_DC 578 – 619</td>
</tr>
<tr>
<td>SAD_V</td>
<td>578 – 619</td>
<td></td>
</tr>
<tr>
<td>SAD_comparator</td>
<td>622 - 660</td>
<td></td>
</tr>
<tr>
<td>Best_mode</td>
<td>660 - 661</td>
<td></td>
</tr>
<tr>
<td>Min_sad</td>
<td>661 - 662</td>
<td></td>
</tr>
<tr>
<td>Total latency</td>
<td>662</td>
<td></td>
</tr>
</tbody>
</table>

Reducing the execution time of sub graphs distributed on FPGA can lead to the reduction of the critical path length (if one of these sub graphs is on critical path). In our example, the critical path length is decreased from 917 to 662. This reduction corresponds to the diminution of execution duration of SAD operations.

### Table. Final scheduling result

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>0 - 256</td>
<td>TOP 256 - 272</td>
</tr>
<tr>
<td>LEFT</td>
<td>272 - 288</td>
<td></td>
</tr>
<tr>
<td>Calc_DC_val</td>
<td>288 - 321</td>
<td></td>
</tr>
<tr>
<td>DC_val</td>
<td>321 - 322</td>
<td></td>
</tr>
<tr>
<td>SAD_H</td>
<td>578 – 619</td>
<td></td>
</tr>
<tr>
<td>SAD_DC</td>
<td>578 – 620</td>
<td></td>
</tr>
<tr>
<td>SAD_V</td>
<td>578 – 620</td>
<td></td>
</tr>
<tr>
<td>SAD_comparator</td>
<td>622 - 660</td>
<td></td>
</tr>
<tr>
<td>Best_mode</td>
<td>660 - 661</td>
<td></td>
</tr>
<tr>
<td>Min_sad</td>
<td>661 - 662</td>
<td></td>
</tr>
<tr>
<td>Total latency</td>
<td>662</td>
<td></td>
</tr>
</tbody>
</table>

We note that despite the increase of the execution duration of SAD_DC and SAD_V, the critical path is still unchanged. So it was beneficial to refactorize them since it uses less area on the FPGA cells. This last result is used to carry out the automatic code generation of the whole mixed application.

Notice that all of these steps are done automatically thanks to a program coded in python that runs and connects the existing tools. Finally the user specifies its algorithm and architecture in SynDEx and will get the optimizations results after few dozen of seconds. Both the code for programmable part and the VHDL code are generated in few seconds.

### V. Conclusion

An AAA methodology extension is suggested as a solution to deal with the increasing complexity of both algorithms and architectures relative to embedded systems. The proposed solution allowed us to optimize algorithm implementation on mixed architectures composed of programmable and reconfigurable components. This extension couples two tools based on AAA i.e. SynDEx and SynDEx-IC. It treats the whole development chain: the algorithm and architecture modeling, the implementation optimization and the executive generation considering communication and synchronization between the different operators. To illustrate the execution of our algorithm coupling SynDEx and SynDEx-IC tools, the intra 16x16 mode decision of the H.264 encoder example was displayed giving an execution total time decrease from 1211 to 662 which represents a gain of...
45%. Thanks to this extension, the AAA methodology could be used to optimize algorithm implementation on mixed architectures. The software tool coupling the AAA heuristics will be available for download for non-commercial use.

REFERENCES


