Transformation-based Exploration of Data Parallel Architecture for Customizable Hardware: A JPEG Encoder Case Study

Rosilde Corvino*, Erkan Diken*, Abdoulaye Gamatié†, Lech Józwiak*
* Technische Universiteit Eindhoven, Eindhoven, The Netherlands
Email: {r.corvino, e.diken, l.jozwiak}@tue.nl
†CNRS/LIFL, Lille, France
Email: abdoulaye.gamatie@lifl.fr

Abstract—In this paper, we present a method for the design of MPSoCs for complex data-intensive applications. This method aims at a blend exploration of the communication, the memory system architecture and the computation resource parallelism. The proposed method is exemplified on a JPEG Encoder case study by describing all the design steps. Our method allows for a JPEG encoder implementation having a throughput increase of 84% and an increase of the achievable FPGA maximum frequency $f_{\text{max}}$ of 64% with an area overhead of $6\times$ with respect to a reference solution. Our method is also assessed with additional explorations of applications from different domains.

Index Terms—Design space exploration, loop transformations, data parallelism, design automation, data transfer and storage architecture

I. INTRODUCTION

Multiprocessor System on Chip (MPSoC) are capable of serving the requirements of new data-intensive streaming applications with an adequate high computing performance and parallelism. However, the high complexity of MPSoC design represents an obstacle in exploiting MPSoC capabilities and may eventually prevent from meeting the design constraints of streaming applications. The MPSoC design complexity has a twofold nature. On one hand, it concerns the deployment of an ad-hoc number of parallel computing resources that should correctly serve the application requirements. On the other hand, it concerns the design of a data transfer and storage micro-architecture [1] that should guarantee a bottleneck-free highly optimized data distribution to the MPSoC parallel computing resources.

The design of MPSoCs for streaming applications has been a hot research topic for the last two decades and, in general, its complexity is addressed at two abstraction levels: 1) the system level [2], [3], [4], targeting the synthesis of system communication and storage mechanisms; 2) the processor level [5], targeting the computing kernel parallelization, scheduling and mapping. However, such a separation of concerns between communication structure and computing resource design, which is required to handle the complexity of MPSoCs design, introduces a separation between data-related design concerns that, especially in the case of data-intensive streaming applications, should be addressed and solved within a unique optimally orchestrated solution.

Processor design for streaming applications has benefit from loop-based High Level Synthesis (HLS) [5], which takes as input a loop-based C-code specifying the streaming behavior of an application and produces as output a Register Transfer Level (RTL) model, usually realizing a VLIW machine. Such loop-based HLS methods largely exploit loop transformations, such as loop unrolling, skewing and tiling, to enhance data locality and application parallelization possibilities [6], [7], [8], [9]. The ultimate aim of these loop transformations is to optimize the result of the processor architecture design and synthesis by improving the quality of the input application specification. In particular, these techniques are used to estimate the storage requirements [9], improve the instruction level parallelism (ILP) [9], [6], [8], optimize the loop iteration scheduling, reduce the redundant memory traffic [9] and improve the memory re-use [7], [10].

However, most of existing works using loop transformations for hardware synthesis fail to capture the interaction between different loop nests [11] and optimize the synthesis of computing data paths only for single loop nests. In contrast, more abstract methods based on synchronous data flows (SDFs) [12] are commonly used to model system communication structure and memory hierarchy, allowing for static scheduling and mapping analysis of streaming applications with multiple communicating loop nests. Unfortunately most SDF models do not take into account the multidimensionality of the transferred data and, consequently, they are not well-suited to describe the effects of loop-transformation-like operations that can be used to efficiently explore the data parallelism of an application. In contrast with these existing methods, we propose a method that is aimed at the orchestrated exploration of the communication and memory system architecture and the computation resource parallelism. Our method selects application-specific optimized solutions allowing for massive data parallelism and, as a consequence, targets multiple communicating loop nests, i.e. loop nests that exchange multidimensional data. Our proposition can be used complementary to the previous solutions in a more general framework: our method optimizes the data transfer
and storage architecture, as well as, the data parallelism, at
the system level, while the previous methods improve
the ILP and reduce redundant memory accesses inside single loop
bodies. Methods capturing interactions between loop nests
are quite unique, involve some serious limitations and are
not automated. For instance, [11] is not automated, it does
not consider the latency of the communication with a shared
memory or bus, it does not support scratch-pad memories,
and it does not consider the data granularity as a parameter of
the design exploration. Due to the high problem complexity,
our method has still several limitations, e.g. it only supports
applications with a predictable behavior. However, it has the
advantages of being fully automated, rapid and efficient.

The contribution of this paper is the application of our
method to an industrial case study: the JPEG encoder. The
presented description goes through all the steps of the design
flow detailing the used models and the obtained results.

The rest of the paper is organized as follows: Section
II presents the Case Study including the description of the
application and the customizable hardware platform; Section
III describes step by step our customization flow applied to
the JPEG encoder, in particular, it introduces the formalism
of the application specification and the analytical model of
the hardware platform, it describes how the Pareto solutions
are selected and used to customize a generic VHDL model of
the hardware platform; Section IV discusses the results of the
method when applied to a considerable number of applications
with different characteristics. Finally, Section V presents the
conclusion and future research directions.

II. CASE STUDY

In this Section, the application and the customizable hard-
ware will be described. The application processes data-streams
and its design involves constraints on the provided through-
put. The customizable hardware architecture includes many
processors, each providing a customized data access control,
distributed local storage and data parallel computing units.

A. JPEG encoder

The JPEG encoder [13] is used in multiple multimedia ap-
lications. It takes as input raw images and produces as output
a compressed bitstream. This example has been chosen as case
study because it is representative of streaming applications and
is a part of more complex compression standards as MPEG4.

After, a downsampling reduces the resolution of the image in
the chrominance components and buckets the pixels in macro-
blocks of $16 \times 16$ or $8 \times 8$ for luminance and chrominance
respectively. A Discrete Cosine Transform (DCT) converts
the YUV into a frequency domain representation. A quantization
reduces the number of encoded variations of luminance and
chrominance into a number of variations that are perceptible
from the human eye. This step corresponds to a frequency
low-pass and as a consequence, it increases the homogeneity
between the values of the pixels in a macro-block. Success-
ively, an entropy encoding compresses the bitstream of data in
the macro-blocks by reading data in a zig-zag order, and by
applying a run-length encoding (RLE) algorithm, which only
encodes the variations of data values.

B. Customizable Hardware Platform

The customizable architecture template includes a library of
customizable elements and a set of rules for the element inter-
connection and customization. An instance of the architecture
template is shown in Fig. 2. It contains pipelines ($PIPE_i$ in
the figure) of Processing Tiles.

![Fig. 2: Customizable architecture template](image)

The Processing Tiles within a single PIPE communicate
through point-to-point links, while the PIPEs communicate
with each other through a shared memory.

Each Processing Tile contains blocks of three basic types:
1) a Double Buffering (DB) block, which implements a double buffer mechanism, i.e., a storage point with two buffers alternatively accessed by two concurrent resources (CTRL and CU), one producing and the other consuming data.

2) a controller (CTRL), managing the Processing Tile data access and storage from and to the shared memory and to the DB blocks.

3) a computing unit (CU), which implements the computations of an application task core, i.e., a datapath on which repetitions of a loop body execute.

These three block types can be entirely customized for a given application with respect to the data width, the data parallelism, the size of local memories and the realized functionality.

Listing 1 presents the HDL code of a generic Processing Tile which has generic, customizable parameters (lines 2-5), i.e., data width, data parallelism, etc.

```
entity Processing_tile is
  generic(
    width : integer := 8; // data width
    n : integer := 4; // data parallelism
    depth : integer := 32; // local memory depth
    ad : integer := 5; // address width
  );
  Port ( 
    clk : in std_logic;
    init : in std_logic;
    send : out std_logic;
    data_in : in std_logic_vector (n-1 downto 0);
    data_out : out std_logic_vector (n-1 downto 0);
  );
end entity;

architecture Behavioral of processor1 is
  -- Components Declaration
  component CTRL generic (...); port (...); end component;
  component DB generic (...); port (...); end component;
  component CU generic (...); port (...); end component;

  -- Signals Declaration
  ...
  signal T_read_addr : std_logic_vector (n-1 downto 0);
  signal T_write_addr : std_logic_vector (ad-1 downto 0);
  signal T_data : std_logic_vector (n-1 downto 0);
  signal T_datal : std_logic_vector (n-1 downto 0);

begin
  -- Instantiate the components
  data_out <= T_datal;
  send <= send_out;
  ctrl_seq : CTRL generic map (...) port map(...);
  par_store : for i in 0 to n-1 generate
    par_store : DB generic map (...) port map(...)
      T_read_addr((i+1)+ad-1 downto ad),
      T_write_addr, data_in((i+1)+width-1 downto i+width),
      T_data((i+1)+width-1 downto i+width);
  end generate;
  par_compute : for i in 0 to n-1 generate
    par_compute : CU generic map (...) port map(...)
      T_datal((i+1)+width-1 downto i+width);
  end generate;
end Behavioral;
```

Listing 1: VHDL code of a generic Processing Tile

Such a VHDL Processing Tile model instantiates the correct number of required parallel resources by combining the concurrent Generate statements with Generate looping capabilities (lines 37-48). In order to minimize the area overhead due to data access management, a single CTRL is used to spread data among the Processing Tiles parallel resources (line 37). The data distribution to the Processing Tile parallel resources can be customized through an adequate address computation (line 39-41 and 46-47).

III. HARDWARE CUSTOMIZATION FLOW

Our method aims at analyzing, restructuring and parallelizing a data-oriented application specification and, from this, at:

- inferring the customization parameters needed to construct application-specific types of Processing Tile’s basic blocks: CTRL, DB and CU.
- selecting the Processing Tile customization parameters such as data width, data parallelism, etc. needed to construct one or more required types of Processing Tiles.
- selecting a system macro-architecture organizing the Processing Tiles in order to realize task pipelines and orchestrate the processor tile communications in locally distributed and globally shared interconnections.

As shown in Fig. 3, our method explores several different loop transformations for an application with multiple communicating nested loops.

![Fig. 3: Customization Flow.](image-url)

From these transformations, such as tiling, paving change and fusion, our method infers some promising architecture template customizations as will be explained in Section III-B. The selected hardware solutions are represented by abstract clocks [14], [15], which are used to capture the scheduling and mapping information of the iterative tasks of an application. The abstract clocks are binary words marking Processing Tiles activities and stalls with 1’s and 0’s respectively. The abstract clocks also capture the scheduling and mapping modifications due to the loop transformations. They are also used to compute the values of some quality indicators such as internal memory.
size (IM), energy consumption (E) and system throughput (T), assessing the exploration results. The solution exploration and selection is performed through a genetic algorithm (GA) implemented on the top of Opt4J [16] framework. In our implementation, the genes encode the applied loop transformations; the explored solutions are represented by abstract clocks and the quality indicators provide information for the best solution selection. Opt4J provides a GA backbone that ensures the evolution of an analyzed population towards Pareto solutions evaluated in a multi-objectives optimization search.

A. Application Model

The input specification of the customization flow is given in an Array Oriented Language (ARRAY-OL) [17] that specifies all the information on task and data dependencies of the application needed for the application specific hardware synthesis. Indeed, an ARRAY-OL model instance contains two levels of abstractions for the application description. A higher level (see Fig. 4a and Fig. 4b) specifies task dependencies similarly to a SDF model [18], where tasks consume and produce data tokens at a certain rate and with constraints of output production conditioned by the input availability. At this level, the Array-OL model can be used for a static scheduling and mapping analysis, by using, for instance, abstract clocks. Also demonstrated by following results, a hierarchical specification of the application prevents some code transformations to be performed, e.g. the fusion of two sub-tasks belonging to two different hierarchical tasks, as for instance DCTY0 and QTY0 in Fig. 4b. Consequently, the hierarchy can be used to limit the number and kind of loop transformations and thus to bound the exploration complexity or prevent unwanted hardware instantiations corresponding to specific loop transformations.

At a lower level, i.e. intra task level (see Fig. 5), an ARRAY-OL model instance captures details about data accesses and data dependencies of a task and expresses them in a matrix form equivalent to a polyhedral model. At this level, parallel versions of the loop nests and their interactions can be explored.

In our flow, the ARRAY-OL specification is given in a textual form (see Listing 2). For each task, equivalent to a loop nest, it is mandatory to specify the task name, the number of repetitions of the task, and the polyhedral model of each input and output array accessed in the loop body (see Fig. 5).

Tasks can be hierarchical. For instance, in Fig. 4b, a hierarchical specification of the JPEG encoder is given. However, as also demonstrated by following results, a hierarchical specification of the application prevents some code transformations to be performed, e.g. the fusion of two sub-tasks belonging to two different hierarchical tasks, as for instance DCTY0 and QTY0 in Fig. 4b. Consequently, the hierarchy can be used to limit the number and kind of loop transformations and thus to bound the exploration complexity or prevent unwanted hardware instantiations corresponding to specific loop transformations.

B. Application parallelization, mapping and scheduling

Loop-transformation-like operations can be applied to an Array-OL specification, such as: 1) task fusion, which merges iterative tasks in the same iteration space; 2) tiling, which adds a level of depth to a loop nest (in our method, the tiled iterations are systematically flattened and this is equivalent to unrolling of iterations in sequentially executed loops); 3) paving-change, which changes the data granularity of a repeated task. This last transformation corresponds to a vectorization in a sequential code. Mapping and scheduling rules assign each Array-OL repeated task to a Processing Tile of the MPSoC and each tile of data produced and consumed by a task to a local double buffer. The aforementioned loop transformations directly set some mapping and scheduling rules, as follows: 1) The task fusion determines the communication structure. Indeed, when two tasks are merged they repeatedly exchange small multidimensional data blocks. Therefore, they are mapped onto a pipeline of Processing Tiles with point-to-point communications between them. They also benefit from
parallel read and write accesses to the local double buffers. By contrast, two unmerged tasks exchange large multidimensional arrays, that cannot be stored internally. Consequently, they are mapped onto processing tiles communicating via the shared bus and global memory with exclusive read and write memory access modes. 2) The paving-change is used to explore different sizes of the local double buffers. 3) The tiling (systematically flattened and unrolled) is used to increase of the parallelism level of the computing resources by multiplying the number of processing elements in a Processing Tile.

Fig. 6 gives an example of the appliance of loop transformations to the model of a flat JPEG encoder, as specified in Fig. 4a. In such a structure, three pipelines of tasks (DCT, QUT, ZiG) are identified for the luminance (Y) and chrominance (U and V) components. Fig. 6 also shows other important parameters generated for the subsequent hardware configuration such as the selected tiling (and unrolling) factor, indicated as a multiplication factor \( \times k \) in Fig. 6, and the selected data granularity, indicated as the multiplicand of the tiling factor. For instance, task DCTv in Fig. 6 has a tiling factor of \( \times 8 \) and a data granularity of 768. These pieces of information are used to customize the VHDL generic model of Fig. 6 by setting the data parallelism and the local memory depth respectively.

After mapping an application onto a corresponding instance of the generic tile-based MPSoC template, the execution of the application is described by abstract clocks, which give the rhythm of data consumption and production for each task, as well as, the synchronization between tasks. The task activities, i.e. data computation and production are marked by 1’s, while the synchronization instants are marked by 0’s.

Several extensions of the method are under development, as, for instance, the introduction of new mapping rules capable of assigning more application tasks to a same Processing Tile or the exploration of design using single buffers for local data storage instead of double buffers. This last modification will force the sequentialization of the data access and computation in a single Processing Tile.

C. Analytical Hardware Model

From the abstract clocks, it is possible to compute three indicators of the application restructuring and related MPSoC quality in order to assess the obtained solutions and guide the design exploration.

Such quality indicators are:

1) Amount of internal memory:

\[
IM = \sum_t \left( \left\{ \sum_{i=1}^{\Pi} (2 \times |\Pi_i(i)|) \right\} + \sum_{t \rightarrow \text{sink}} \left\{ \sum_o (|\Pi_t(o)|) \right\} \right)
\]

where \( |\Pi_t(i)| \) (respectively \( |\Pi_t(o)| \)) indicates the number of 1’s in the period \( \Pi_t \) of an abstract clock associated with an input port \( i \) (respectively output port \( o \)) of a task \( t \). The factor \( |\Pi_t(i)| \) represents the number of input data needed to be available on the port \( i \) so that the task \( t \) can fire \( |\Pi_t(o)| \) outputs. The factor 2 is due to the double buffering mechanism. The index \( t \rightarrow \text{sink} \) indicates all tasks \( t \) communicating with a sink.

2) Throughput:

\[
T = \frac{\sum_{t \rightarrow \text{sink}} \left\{ \sum_{i=1}^{\Pi} (|\Pi_t(i)|) \right\}}{\sum_p \left\{ r \times \Delta \Pi_P \right\}}
\]

where the numerator is the total number of produced output data and the denominator is the latency of the computations needed to produce the total output data.

In the numerator, \( o \) indicates an output port of a task \( t \) communicating with a sink. In the denominator, \( \Delta \Pi_P \) is a latency computed as \( \Pi_P \times r \), where \( \Pi_P \) is the period of an abstract clock associated with two or more tasks merged in a pipeline \( p \) and \( r \) is the number of times this period is repeated until the output data of the pipeline are produced.

3) Energy consumption due to the shared memory accesses:

\[
E = E_{\text{read}} \times \sum_{t \leftrightarrow \text{source}} |\Pi_t(i)|_1 + E_{\text{write}} \times \sum_{t \leftrightarrow \text{sink}} |\Pi_t(o)|_1
\]

where \( |\Pi_t(.)|_1 \) is defined as in the above formulas. \( E_{\text{read}} \) and \( E_{\text{write}} \) are the energy consumption per read and write depending on the used technology and the size of the shared memory. In our explorations, we use 0.35 \( \mu m \) SRAM. The size \( EM \) of the shared memory is the sum of all the input and output array sizes, for all the tasks communicating with the source and sink tasks. It depends on the transformations applied to the Array-OL specification and on the used technology. Given \( EM \), \( E_{\text{read}} = a_r + b_r \times EM \) and \( E_{\text{write}} = a_w + b_w \times EM \), with \( a_r = 6.37504 \times 10^{-4}, b_r = 1.186 \times 10^{-1}, a_w = 4.75004 \times 10^{-4} \) and \( b_w = 3.65 \times 10^{-2} \).

D. Multi-objective Exploration and Pareto Solutions Selection

A JPEG encoder exploration has been run by setting parameters that guide the exploration as specified in Table I. For instance the tiling and paving change factors (shortly called loop transformation factors) determine the loop transformations applied to the application specification model and the consequent hardware customizations as specified in Section...
The computing units, realizing the DCT, the quantization, etc., have been synthesized, placed and routed with ISE Xilinx 13.2, on a FPGA Virtex-6 XC6VCX75T. The architecture has been optimized to achieve better results, which clearly dominate the results of the OpenCores solution. Our solution can process up to 480 frames of 640×480 pixels per second (Fps) with respect to 97 Mhz of the OpenCores solution. Our solution can process up to 162 frames of 640×480 pixels per second (Fps) with respect to 136 Fps of the OpenCores solution. The main reason of the throughput and maximum frequency increase for our solution is the usage of a higher degree of parallelism and pipelining, achieved instantiating several times basic computing units (DCT, quantization, etc.) and several local buffers, according to the architecture template of Fig. 2. As a consequence, our solution is 6× larger than the one of OpenCores. However, this area occupancy can be dramatically reduced when identifying hardware sharing possibilities between the luminance and chrominance pipelines (including DCT, quantization and run-length tasks). Indeed, these pipelines are never activated at

We have implemented a JPEG structure (i.e. an architecture with a parallelism and a data granularity) as represented in Fig. 6, on a FPGA Virtex-6 XC6VCX75T. The architecture has been synthesized, placed and routed with ISE Xilinx 13.2 with a speed grade of -2 and a user clock period of 20 ns. The computing units, realizing the DCT, the quantization, etc., are taken from the JPEG encoder of OpenCores [19]. As a comparing example, we have also implemented the OpenCores JPEG encoder on the same FPGA and with the same synthesis tool setting. The result of our method has a maximum frequency $f_{max}$ of 150 MHz with respect to the 97 MHz of the OpenCores solution. Our solution can process up to 162 frames of 640×480 pixels per second (Fps) with respect to 136 Fps of the OpenCores solution. The main reason of the throughput and maximum frequency increase for our solution is the usage of a higher degree of parallelism and pipelining, achieved instantiating several times basic computing units (DCT, quantization, etc.) and several local buffers, according to the architecture template of Fig. 2. As a consequence, our solution is 6× larger than the one of OpenCores. However, this area occupancy can be dramatically reduced when identifying hardware sharing possibilities between the luminance and chrominance pipelines (including DCT, quantization and run-length tasks). Indeed, these pipelines are never activated at

### Table I: Parameter setting for JPEG encoder exploration.

<table>
<thead>
<tr>
<th>GA</th>
<th>population size</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_2$</td>
<td>number of explored generations</td>
<td>100</td>
</tr>
<tr>
<td>$p_3$</td>
<td>number of parents per generation</td>
<td>25</td>
</tr>
<tr>
<td>$p_4$</td>
<td>number of children per generation</td>
<td>25</td>
</tr>
<tr>
<td>$p_5$</td>
<td>parent crossover rate</td>
<td>95%</td>
</tr>
</tbody>
</table>

Other Parameters

<table>
<thead>
<tr>
<th>Tiling factor</th>
<th>1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paving change factor</td>
<td>1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048</td>
</tr>
</tbody>
</table>

E. VHDL Model Customization and Simulation

We have implemented a JPEG structure (i.e. an architecture with a parallelism and a data granularity) as represented in Fig. 6, on a FPGA Virtex-6 XC6VCX75T. The architecture has been synthesized, placed and routed with ISE Xilinx 13.2 with a speed grade of -2 and a user clock period of 20 ns. The computing units, realizing the DCT, the quantization, etc., are taken from the JPEG encoder of OpenCores [19]. As a comparing example, we have also implemented the OpenCores JPEG encoder on the same FPGA and with the same synthesis tool setting. The result of our method has a maximum frequency $f_{max}$ of 150 MHz with respect to the 97 MHz of the OpenCores solution. Our solution can process up to 162 frames of 640×480 pixels per second (Fps) with respect to 136 Fps of the OpenCores solution. The main reason of the throughput and maximum frequency increase for our solution is the usage of a higher degree of parallelism and pipelining, achieved instantiating several times basic computing units (DCT, quantization, etc.) and several local buffers, according to the architecture template of Fig. 2. As a consequence, our solution is 6× larger than the one of OpenCores. However, this area occupancy can be dramatically reduced when identifying hardware sharing possibilities between the luminance and chrominance pipelines (including DCT, quantization and run-length tasks). Indeed, these pipelines are never activated at
the same time because they share the communication with the Global Memory. Therefore it is possible to merge their hardware datapaths and share the same hardware for their execution. Hardware sharing identification is one of our major future research direction.

IV. FURTHER EXPERIMENTS WITH THE PROPOSED METHOD

This section discusses the experimental results of our GA-based exploration method when applied to several applications having different characteristics. First, the experimental setup is briefly described. Then, the exploration results are reported and discussed.

A. Experimental Setup

In order to illustrate our method, several applications from different domains are processed and the obtained results are discussed. Table II lists the selected applications and provides a brief description of them. The list of selected applications includes applications (e.g. 3mm, 2mm, gemver, fddt-2d) from PolyBench 3.1 [20] and some other applications (e.g. LPSF, YJPEG) from image processing and multimedia domains.

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3mm</td>
<td>linear-algebra kernel</td>
</tr>
<tr>
<td>2mm</td>
<td>linear-algebra kernel</td>
</tr>
<tr>
<td>gemver</td>
<td>linear-algebra kernel</td>
</tr>
<tr>
<td>fddt-2d</td>
<td>stencil application</td>
</tr>
<tr>
<td>YJPEG</td>
<td>low pass spatial filter</td>
</tr>
<tr>
<td></td>
<td>gray-scale JPEG encoder</td>
</tr>
</tbody>
</table>

TABLE II: application descriptions

The explorations are performed on a Intel (R) Core (TM) i5 quad core (2.53GHz) processor running a GNU/Linux operating system with 3.5GB of RAM memory. This analysis compares the results of our GA-based method with those of an exhaustive search that we have developed for comparing reasons. Since the computation power of the host on which the explorations were run is limited with respect to the heap memory size, it is required to adapt the setting of the parameters of the exhaustive exploration to reduce its complexity (e.g. it is required to reduce the size of the explored population for complex problems). As a consequence of this limitation, and to keep valid the comparison between the exhaustive and GA-based explorations, we also limit the exploration capability of our GA-based method even though this is not required. Indeed, with the available computing resources, the computation complexity of our exploration allows performing very complex explorations within few seconds. Consequently, the parameters \((p_1, p_2, p_3, p_4, p_5)\) of the genetic algorithm are set to the values provided in Table III. Additionally, table III gives information on the loop transformation factors \((k)\) for each corresponding application.

B. Exploration Results

Table IV reports the complexity of the performed explorations by reporting for each analyzed application: the number of tasks in the analyzed application, the number of possible fusions, the total number of explored individuals, the number of possible individuals in the whole exploration space and the number of selected Pareto solutions. For example, 3mm linear-algebra application has 6 tasks and 15 possible task fusions. For each fusion, the initial population of the genetic algorithm includes 40 individuals and it evolves for 10 generations. At each evolution, 2 new individuals are created and explored. Considering the parameters defined in Tables III and IV, the expression \(((p_2 \ast p_1) + p_1) \ast c_2\) formulates the total number of explored individuals \((c_3)\). Hence, value of \(c_3\) for 3mm is \((10 \ast 2) + 40 \ast 10 = 900\). Furthermore, the formula \(k = c_2\),

\[
\text{TABLE III: Opt4J parameter setting: } p_1: \text{population size, } p_2: \text{number of explored generations, } p_3: \text{number of parents elected for reproduction (i.e. crossover), } p_4: \text{number of children elected to integrate the new generation and } p_5: \text{crossover rate}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Application Name} & \text{Description} & k \\
\hline
3mm, fddt-2d, gemver, LPSF & \{40, 10, 4, 95\%\} & \{1,2,4\} \\
YJPEG & \{40, 10, 4, 95\%\} & \{1,2\} \\
\hline
\end{array}
\]

\[
\text{TABLE IV: Exploration Complexity and Selectivity: } c_1: \text{number of tasks in the application, } c_2: \text{number of possible fusions, } c_3: \text{total number of explored individuals, } c_4: \text{number of possible individuals in the whole exploration space and } c_5: \text{number of Pareto solutions}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Application Name} & \text{Description} & c_1 & c_2 & c_3 \\
\hline
3mm & linear-algebra kernel & 6 & 15 & 900 & 900 \% & 2 \\
YJPEG & stencil application & 5 & 15 & 400 & 400 \% & 4 \\
LPSF & low pass spatial filter & 4 & 8 & 640 & 640 \% & 1 \\
\hline
2mm & linear-algebra kernel & 4 & 8 & 480 & 480 \% & 3 \\
fddt-2d & stencil application & 4 & 8 & 480 & 480 \% & 2 \\
gemver & gray-scale JPEG encoder & 4 & 8 & 480 & 480 \% & 3 \\
\hline
\end{array}
\]

In order to assess the quality of our method, the output of our method, which is a Pareto-front, is compared to the Pareto-front provided by an exhaustive search. The exhaustive search explores all the possible application transformations in

\[
\text{TABLE V: Information on the exploration run-time for the GA-based \((r_1)\) and exhaustive \((r_2)\) methods. Moreover, it gives the percentage of run-time for each exploration step of the GA-based exploration: fusion exploration \((r_2)\) and genetic algorithm in Opt4J \((r_3)\). As it can be seen from the table, most of the time for the GA-based method is spent by the the Opt4J framework to perform genetic algorithms. The }r_2\text{ values are relatively small compared to the }r_3\text{ values; however }r_2\text{ values trend to significantly increase in proportion with the increase of the application complexity (e.g. 3mm, YJPEG).}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{Application Name} & \text{Description} & % \\
\hline
3mm & linear-algebra kernel & 960 \% \\
YJPEG & stencil application & 960 \% \\
\hline
\end{array}
\]
the specified space. The comparison between our method and the exhaustive search is assessed through the computation of the $\epsilon$-indicator, as presented in [22].

Basically, the $\epsilon$-indicator checks the closeness of two Pareto-fronts, in our case, the reference Pareto-front is output of the exhaustive exploration. The closer the $\epsilon$-indicator is to 1, the better the solution. Table V also provides the $\epsilon$-indicators to measure the closeness of the Pareto fronts of the GA-based and exhaustive methods. This table demonstrates that our method is rapid and efficient in a sense that it explores the design space in a significantly less amount of time compared to the exhaustive exploration. Moreover, our method is effective because it still maintains the Pareto-front solutions, inferred from the fact that the $\epsilon$-indicator values are either 1 or close to 1.

V. CONCLUSION AND FUTURE WORKS

In this paper, we presented a method for the design of MPSoCs for data-intensive applications. Our method uses loop transformations to perform a blend optimization of the hardware communication structure, the memory hierarchy and the computing resource parallelism. A JPEG Encoder is studied as a representative case study in order to demonstrate all the steps of our method. Furthermore, several applications from different domains are tested in order to demonstrate the method quality and applicability. Experimental results show that the proposed method is rapid and efficient. The JPEG encoder implementation demonstrates that our method allows for implementations with a significant throughput increase, an increase of the supported $f_{\text{max}}$ of the FPGA implementation and a low area overhead due to the logic and an efficient exploitation of the FPGA integrated RAMs. To the best of our knowledge, this method is the first one using loop transformations to explore in combination the communication, the memory and the computing architectures. Several future research directions are possible, such as integrating into our method additional relevant loop transformations; enlarging the list of the possible mapping and scheduling rules; exploring hardware sharing and applying our method to the case of Application Specific Instruction Set Processors.

VI. ACKNOWLEDGEMENT

This paper has been partially supported by ASAM project of the ARTEMIS Research Program.

**TABLE V:** Exploration Run-Time: $r_1$: total run-time of the GA-based exploration in seconds, $r_2$: total run-time of the exhaustive exploration in seconds, $r_7$: percentage of the GA-based method run-time spent for fusion exploration, $r_{3}$: percentage of the GA-based method run-time spent for GA running.

<table>
<thead>
<tr>
<th></th>
<th>$r_1$ (sec.)</th>
<th>$r_2$ (sec.)</th>
<th>$r_7$ (%)</th>
<th>$r_3$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIPPEG</td>
<td>0.3</td>
<td>2.7</td>
<td>2.8</td>
<td>2.3</td>
</tr>
<tr>
<td>LPSF</td>
<td>2.9</td>
<td>2.8</td>
<td>1.11</td>
<td>1.26</td>
</tr>
<tr>
<td>gemver</td>
<td>4.4</td>
<td>22.2</td>
<td>4.7</td>
<td>3.1</td>
</tr>
</tbody>
</table>

REFERENCES


