System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip

Christian Pilato, *Member, IEEE,* Paolo Mantovani, *Student Member, IEEE,* Giuseppe Di Guglielmo, *Member, IEEE,* Luca P. Carloni, *Senior Member, IEEE*

***Abstract*—In modern System-on-Chip (SoC) architectures, specialized accelerators are increasingly used to improve performance and energy efficiency. The growing complexity of these systems requires the use of system-level design methodologies featuring high-level synthesis (HLS) for generating these components efficiently. Existing HLS tools, however, have limited support for the system-level optimization of memory elements, which typically occupy most of the accelerator area. We present a complete methodology for designing the private local memories (PLMs) of multiple accelerators. Based on the memory requirements of each accelerator, our methodology automatically determines an area-efficient architecture for the PLMs to guarantee performance and reduce the memory cost based on technology-related information. We implemented a prototype tool, called MNEMOSYNE, that embodies our methodology within a commercial HLS flow. We designed thirteen complex accelerators for selected applications from two recently-released benchmark suites (PERFECT and CORTEXSUITE). With our approach we are able to reduce the memory cost of single accelerators by up to 45%. Moreover, when reusing memory IPs across accelerators, we achieve area savings that range between 17% and 55% compared to the case where the PLMs are designed separately.**

***Index Terms*—Hardware Accelerator, High-Level Synthesis, Memory Design, Multi-bank Architecture.**

I. INTRODUCTION

SYSTEM-ON-CHIP (SoC) architectures increasingly feature *hardware accelerators* to achieve energy-efficient high performance [1]. Complex applications leverage these specialized components to improve the execution of selected computational kernels [2], [3]. For example, hardware accelerators for machine learning applications are increasingly used to identify underlying relations in massive unstructured data [4], [5], [6]. Many of these algorithms first build an internal model by analyzing very large data sets; then, they leverage this model to perform decisions (e.g. to give suggestions to the users). Thanks to the inherent parallelism of their kernels, they are good candidates for hardware specialization, especially with *loosely-coupled accelerators* (LCAs) [7], [8], [9].

Manuscript received February 12, 2015; revised June 11, 2016; accepted August 24, 2016. This paper was recommended by Associate Editor L. Benini. This work was supported in part by C-FAR, one of the six centers of STARnet, the DARPA PERFECT program (Contract No: HR0011-13-C-0003), and the NSF (A#: 1219001 and 1527821).

C. Pilato was with Columbia University, New York, NY 10027 USA, and he is now with the University of Lugano, Lugano 6900, Switzerland (email: christian.pilato@usi.ch).

P. Mantovani, G. Di Guglielmo and L.P. Carloni are with Columbia University, New York, NY 10027, USA (e-mail: {paolo,giuseppe,luca}@cs.columbia.edu).

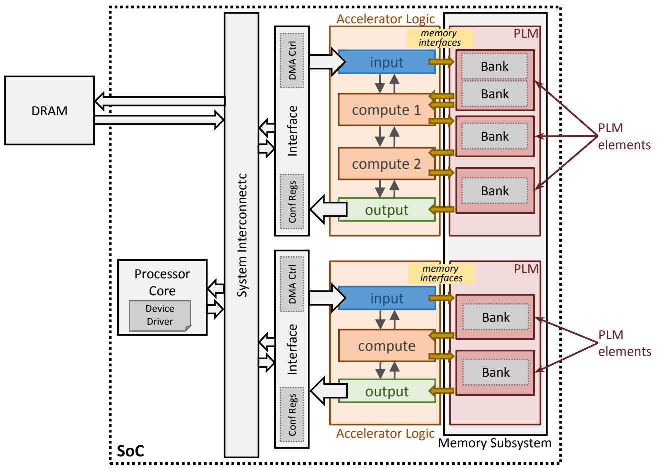

Fig. 1. Accelerator-based SoC. Memory banks can be reused across accelerators to reduce resource requirements.

The example in Fig. 1 shows a portion of an SoC, including two LCAs and a processor core, connected to an external memory (DRAM). Each LCA is composed of the *accelerator logic*, which implements the computation, and the *private local memory* (PLM), which stores data to be accessed with fixed latency [7], [10]. PLMs constitute the accelerator *memory subsystem* of the SoC and are composed of many units, called *PLM elements*. Each of these PLM elements is used to store a data structure of the algorithm. Although PLMs are known to be responsible for most of the accelerator area [10], at any given time they can contain only a portion of the entire working data set, which is entirely stored within the DRAM. The accelerator computation is thus organized in consecutive iterations, where data are progressively exchanged with DRAM through DMA transfers [7]. So, the accelerator logic is structured with multiple hardware blocks executing concurrently, in parallel or in pipeline (i.e. *input*, $compute_k$, and *output*). Hardware blocks $input$ and $output$ manage the data transfers, while hardware blocks $compute_k$ implement the functionality of the accelerator. The PLM management is thus totally transparent to the processor core, which is responsible for preparing the data in DRAM and controlling the accelerators' execution. The core runs an operating system and each accelerator is managed by a *device driver* [11].

LCAs can achieve better performance than processor cores thanks to specialized micro-architectures for both the accelerator logic and the PLM in order to execute the algorithm for which it has been designed. The accelerator logic can exploit spatial parallelism to execute multiple operations in parallel. The size of each PLM element is customized with respect to the amount of data to be stored. Additionally, while processor memories are designed for sequential access (even in case of memory sharing with the accelerator [12], [13]), PLMs require multiple ports to allow the accelerator logic to perform multiple memory operations within the same clock cycle and increase the hardware parallelism. There are different solutions to implement multi-port memories [14]. Distributed registers, which are completely contained into the accelerator logic, are used for small and frequently accessed data structures. However, the aggregated size of these registers is known to grow exponentially with the amount of data to be stored. Large and complex data structures require the allocation of dedicated *memory Intellectual Property (IP) blocks*, which are more resource efficient. However, since the size of these memory elements grows quadratically with the number of ports [15], only single- or dual-port memory IPs are usually offered by technology providers [16]. The available memory IPs compose the so-called *memory library*, where each of them is characterized in terms of height, width, and resource requirements. For example, a variable number of Static Random-Access Memories (SRAMs) are available in standard cell-based designs. Block Random-Access Memories (BRAMs) are used instead when targeting FPGA technologies, which have a certain number of such configurable blocks available in each device (e.g. between 1,500 and 4,000 16Kb BRAMs in modern Xilinx Virtex-7 FPGAs [17]). Each PLM element is then implemented with a multi-bank architecture, based on the combined requirements of each hardware block accessing the corresponding data structure. For example, in the first accelerator of Fig. 1, hardware blocks $input$ and $compute_1$ communicate through a data structure; at each clock cycle, $input$ updates one value with one memory-write interface, while $compute_1$ elaborates two values with two distinct memory-read interfaces. To manage these three concurrent memory operations, the corresponding PLM element must have two dual-port banks.

Due to the growing complexity of these SoCs, *system-level design* methodologies are used to increase design productivity by optimizing the components at a level of abstraction higher than RTL [18], [19]. There is also a trend to separate the *IP design*, where optimized components are created for specific purposes, from the *SoC integration* of these components. This reduces the design complexity, but may limit the optimization of the design of accelerators that are integrated on the same SoC, especially with respect to their memory elements. Fig. 2(a) shows how current methodologies work for the design of accelerator-based SoCs. First, each algorithm is specified in a high-level language (e.g. SystemC) to enable the use of *high-level synthesis* (HLS) [20], [21], [22]. Then, state-of-the-art HLS tools are used to derive multiple Pareto-optimal implementations for the accelerator logic [23], [24]. These are alternative trade-offs in terms of performance versus cost (area or power). They can be created by applying a rich set of "*knobs*" (e.g. activating loop transformations or varying the number of memory interfaces to access each data structure) to the same SystemC code [23], [25]. To avoid the manual design of the PLMs, multi-bank architectures can be specified with source code transformations before running HLS [26], [27]. However, the possibilities for memory-related optimizations are limited during the SoC integration. Moreover, it may be necessary to reiterate the IP design when changing the requirements for the memory subsystem (e.g. the available area). This is time consuming and error-prone. In contrast, designing the memory subsystem during SoC integration enables additional optimizations to reduce the memory cost.

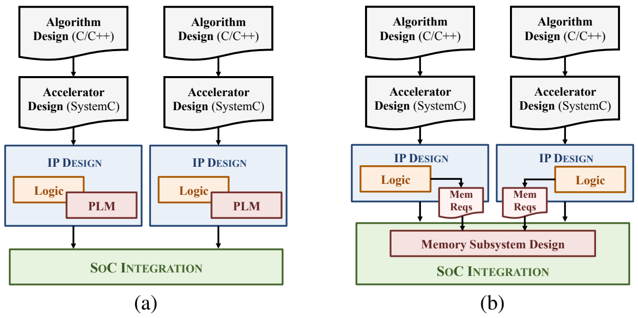

Fig. 2. Traditional (a) and proposed (b) design flow for heterogeneous SoCs.

We thus propose an alternative approach, which is shown in Fig. 2(b), to design and optimize the memory subsystem of multiple LCAs during their integration in the SoC. We first explore the micro-architectures of the accelerators with HLS tools and collect their memory requirements. Then, we generate optimized PLM micro-architectures by taking into account also the characteristics of the memory IPs of the given target technology. To reduce the memory cost, we enable the reuse of memory IPs across the accelerators that are not executed at the same time. This flexibility is achieved with a *memory controller* that encapsulates the actual memory IPs and coordinates the accesses from the different LCAs. With this methodology, we can design and optimize the entire memory subsystem for SoCs with different requirements and without necessarily modifying the accelerators.

Contributions. After introducing a paradigmatic example in Section II, we present our main contributions:

- a methodology to automatically derive the accelerator memory subsystem for LCAs (Section III);
- a set of technology-unaware and technology-aware optimizations to reduce the cost of the accelerator memory subsystem (Section IV and Section V, respectively).
- a flexible controller to manage the accesses to the generated PLM micro-architecture (Section VI);

We implemented these combined contributions in a prototype CAD tool, called MNEMOSYNE, which we used to optimize the memory subsystems of many complex accelerators that we designed for applications selected from two new benchmark suites, PERFECT [28] and CORTEXSUITE [29] (Section VII).

II. ACCELERATOR DESIGN

In this section, we introduce a relatively small example to illustrate the main issues that must be addressed when designing hardware accelerators with system-level methodologies. Listing 1 reports a portion of the synthesizable SystemC code of *Debayer*, an accelerator for image debayering [28]. This accelerator has been designed with three concurrent processes

Listing 1. Synthesizable SystemC code of *Debayer*, an accelerator for image debayering [28].

```
1  #include <systemc.h>
2  SC_MODULE(Debayer) {
3   sc_in<bool> clk, rst;
4  private:
5   sc_signal<bool> i_valid, i_ready, o_valid, o_ready;
6   int A0[6][2048];  // circular buffer
7   int B0[2048];
8   int B1[2048];
9  public:
10  //...
11  SC_CTOR(debayer) {
12   SC_CTHREAD(input, clk.pos());
13   reset_signal_is(rst, false);
14   SC_CTHREAD(compute, clk.pos());
15   reset_signal_is(rst, false);
16   SC_CTHREAD(output, clk.pos());
17   reset_signal_is(rst, false);
18   //...
19  }
20  void input(void) {
21   // reset ...
22   unsigned circ = 0; // circular buffer write pointer
23   wait();
24   while(true) {
25    L0: for (int r=0; r<2048; r++) {
26     // DMA request
27     // read input ...
28     L1:  for (int c=0; c<2048; c++)
29      { A0[circ][c] = f(...); } //write to A0
30     // output ...
31     if (r >=5){
32       // wait for ready from compute then notify as valid
33     }
34     circ++;
35     if (circ == 6)
36      circ = 0;
37    }
38   }
39  }
40  void compute(void) {
41   int PAD = 2; bool flag = true;
42   int r_r = 0; // central row of the mask
43   // reset ...
44   wait();
45   while(true) {
46    L2: for (int r=0; r<2048-PAD; r++) {
47     // (wait for valid from input then notify as ready)
48     r_r = circ_buffer_row(r + 2);
49     L3:  for (int j=PAD; j<2048-PAD; j++) {
50       if (flag) B0[j] = g(A0[r_r][j-2], A0[r_r][j-1],
51          A0[r_r][j], A0[r_r][j+1], A0[r_r][j+2], ...);
52       else B1[j] = g(A0[r_r][j-2], A0[r_r][j-1],
53          A0[r_r][j], A0[r_r][j+1], A0[r_r][j+2], ...);
54      }
55     // (valid to output, ready to compute)
56     flag = !flag;
57    }
58   }
59  }
60  void output(void) {
61   int PAD = 2; bool flag = true;
62   // reset ...
63   wait();
64   while(true) {
65    L4: for (int r=PAD; r<2048-PAD; r++) {
66     // (wait for valid from compute then notify as ready)
67     // prepare DMA request
68     // send data
69     L5:  for (int c=PAD; c<2048-PAD; c++) {
70      if (flag) h(B0[c], ...);  //read from array B0
71      else h(B1[c], ...);       //read from array B1
72     }
73     // (ready to compute)
74     flag = !flag;
75    }
76   }
77  }
78 };
```

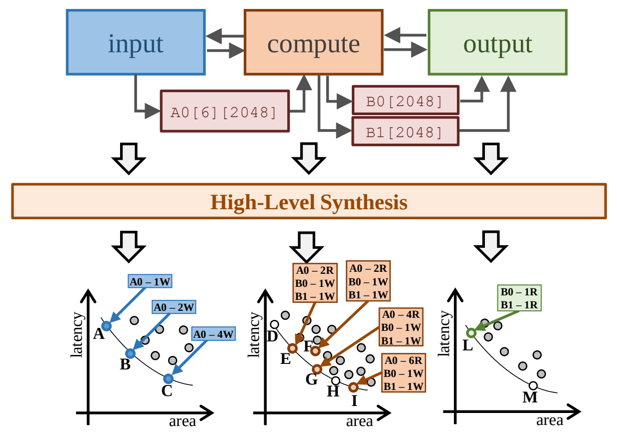

Fig. 3. Graphical representation of the *Debayer* accelerator presented in Listing 1, along with associated Pareto-set implementations.

(*input*, *compute*, and *output*) that are connected as shown in the upper part of Fig. 3. They execute in pipeline on a 2,048×2,048-pixel image, which is stored in DRAM, to produce the corresponding debayered version. The accelerator is connected to the rest of the system as shown in Fig. 1.

Example. Process *compute* elaborates each row of the input image (lines 46-54). It uses a mask around each pixel and thus requires four additional rows, two above and two below the current one. Three data structures are used to store the data: one for the input rows (i.e. array `A0`) and two for the results (i.e. arrays `B0` and `B1`). Array `A0` (line 6) is implemented as a *circular buffer* with the capacity of storing 6×2,048 pixels. In fact, an additional row is stored to overlap communication and computation. So, after reading the first five rows, process *input* fetches one new row at each iteration through the DMA controller (lines 26-30), discarding the oldest one. Arrays `B0` and `B1` (lines 7-8) are used, instead, to form a *ping-pong buffer*. Each array stores one row (i.e. 2,048 pixels). In this way, process *output* can send one row back to DRAM (lines 67-71), while process *compute* is producing the next one into the other array. The three processes work in pipeline and synchronize their execution through explicit protocol signals (`valid`, `ready`) such that one process cannot start its computation before the previous one has produced the required amount of data (lines 32 and 47, and lines 55 and 66). By using such latency-insensitive protocol [30], it is possible to vary the execution time of one process without affecting the execution of the others. □

The *circular buffer* and the *ping-pong buffer* are mechanisms widely used in high-throughput accelerators to optimize communication and computation at the cost of increasing the PLM size [7]. The former allows the reuse of local data, thus minimizing the amount of data transfers with DRAM. The latter allows the overlapping of computation and communication.

Design Space Exploration. HLS tools allow the design of accelerators at a higher level of abstraction. The designer can generate many alternative RTL implementations by applying multiple "knobs" to trade off performance metrics (e.g. latency) and area/power costs. In the set of resulting designs, it is possible to identify Pareto-optimal choices, as shown in the lower part of Fig. 3.

Example. Consider the implementations of process *compute*. Implementation E is obtained by unrolling `L3` for two iterations, which requires two concurrent memory-read operations. Implementation F is obtained by unrolling `L3` for four iterations to maximize performance at the cost of more area, but with only two memory-read interfaces; this creates a bottleneck because the four memory operations cannot be all scheduled in the same clock cycle. Implementation G, which Pareto-dominates implementation F, is obtained by unrolling `L3` for four iterations and having four memory-read interfaces to allow the four memory-read operations to execute concurrently. □

Accelerator Logic Design. Based on the overall requirements of the SoC architecture, the designer then selects an implementation for each process to create the final system (i.e. *compositional high-level synthesis* [23], [24]). Compositional HLS allows IP designers to optimize the different hardware blocks separately and more efficiently, but requires that selecting an implementation for one block does not imply any changes to the others. This is critical for shared resources, such as memory elements. In fact, changing the number of concurrent memory operations on a data structure shared between two components may impact the memory operations of the other components.

Example. Assume that implementations A and E are selected for processes *input* and *compute*, respectively; then, array `A0` must be stored in a PLM with one memory-write interface and two memory-read interfaces. Instead, if implementation G is selected for process *compute*, the PLM for storing the same array requires four memory-read interfaces. □

System-level Memory Optimization. We aim at generating an optimized memory subsystem for one or more accelerators. The designer provides information on the data structures to be stored in the PLMs, along with additional information on the number of memory interfaces for each accelerator and the *compatibilities* between the data structures. This information is used to share the memory IPs across accelerators whenever it is possible. Our approach is motivated by the following observations. First, when a data structure is not used, the associated PLM does not contain any useful data; the corresponding memory IPs can be reused for storing another data structure, thus reducing the total size of the memory subsystem [10]. Second, in some technologies, the area of a single memory IP is smaller than the aggregated area of smaller IPs. For example, in an industrial 32nm CMOS technology, we experimented that a 1,024×32 SRAM is almost 40% smaller than the area of two 512×32 SRAMs, due to the replicated logic for address decoding. In these cases, it is possible to store two data structures in the same memory IP provided that there are not conflicts on the memory interfaces, i.e. the data structures are never accessed at the same time with the same memory operation. Next, we formalize these situations.

To understand when two data structures can share the same memory IPs, we recall the definition of *data structure lifetime*.

Definition. The lifetime of a data structure b is the interval time between the first memory-write and the last memory-read operations to the data structure [31]. ■

Having two data structures with no overlapping lifetimes means that while operating on one data structure the other remains unused. Hence, we can use the same memory IPs to store both of them. On the other hand, even when two data structures have overlapping lifetimes, it is still possible to share memory interfaces to potentially reduce the accelerator area.

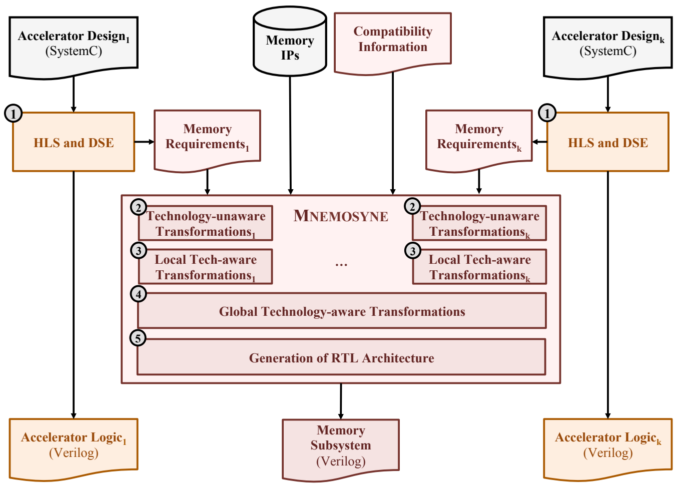
Accelerator Design$_1$ (SystemC)
Memory IPs
Compatibility Information
Accelerator Design$_k$ (SystemC)
1
HLS and DSE
Memory Requirements$_1$
Memory Requirements$_k$
1
HLS and DSE
MNEMOSYNE
2
Technology-unaware Transformations$_1$
2
Technology-unaware Transformations$_k$
3
Local Tech-aware Transformations$_1$
...
3
Local Tech-aware Transformations$_k$
4
Global Technology-aware Transformations
5
Generation of RTL Architecture
Accelerator Logic$_1$ (Verilog)
Memory Subsystem (Verilog)
Accelerator Logic$_k$ (Verilog)

Fig. 4. Methodology overview for accelerator memory design.

Definition. Two data structures b_i and b_j are address-space compatible when their lifetimes are not overlapping for the entire execution of the accelerator. They are memory-interface compatible when it is possible to define a *total temporal ordering* of the memory operations so that two read (resp. write) accesses to b_i and b_j never happen at the same time. ■

When two data structures are memory-interface compatible, memory-read and memory-write operations are never executed at the same time on the same data structure.

Example. Processes *compute* and *output* of the Debayer accelerator in Listing 1 use arrays `B0` and `B1` to exchange data. When process *compute* is writing into `B0`, process *output* is reading from `B1` and vice versa. Hence, the two arrays are never written (read) at the same clock cycle. □

III. Proposed Methodology

To assist the system-level optimization of the memory subsystem for K accelerators, we propose the methodology shown in Fig. 4. Our methodology takes as input the SystemC descriptions of the accelerators (*Accelerator Design*$_{1...k}$) and the information about compatibilities among their data structures (*Compatibility Information*). We first use a commercial HLS tool to perform design space exploration and generate many alternative micro-architectures of each accelerator logic in order to optimize the performance (*HLS and DSE*). Each implementation is characterized by a set of data structures to be stored in the PLM and the corresponding requirements in terms of memory interfaces (*Memory Requirements*$_{1...k}$). After selecting an implementation for each component, we determine the combined requirements in terms of memory interfaces to access each data structure in order to guarantee performance and functional correctness (*Technology-unaware Transformations*$_{1...k}$). We combine the information on all data structures (*Memory Requirements*$_{1...k}$), the information on compatibilities (*Compatibility Information*), and the characteristics of the memory IPs in the *Memory Library* to determine an optimized architecture for each PLM. First, we apply transformations for each accelerator (*Local Technology-aware Transformations*$_{1\ldots k}$). Then, we consider all accelerators at the same time and identify when the memory IPs can be reused across different data structures to minimize the cost of the entire memory subsystem (*Global Technology-aware Transformations*). As output, we produce the RTL description of the memory subsystem (*Generation of RTL Architecture*) that can be directly integrated with the RTL descriptions of the accelerator logic generated by the HLS tool.

We implemented the steps related to memory optimization (from ② to ⑤ of Fig. 4) in a prototype tool, called MNEMOSYNE. We interfaced MNEMOSYNE with a commercial HLS tool to automatically derive the memory requirements based on the knobs' configuration. In the following sections, we describe each of the memory-optimization steps.

IV. TECHNOLOGY-UNAWARE TRANSFORMATIONS

In the HLS phase, the designer applies a set of micro-architectural optimization knobs to trade-off performance and cost for the accelerator logic. The corresponding PLM architecture has then to be designed so that the accelerator behaves correctly and achieves the desired performance. Specifically, the PLM must provide each data to the accelerator logic in the number of cycles (usually one) assumed by the HLS scheduling phase. For functional correctness, no more than one operation must be executed on each port of the memory IP at the same time (i.e. *conflict-free accesses* to the banks). It is thus necessary to determine the number of concurrent memory operations required to access each data structure and the corresponding number of memory interfaces. These combined memory requirements determine the number of *parallel blocks* required by each data structure to avoid conflicts when accessing the banks. To reduce the cost of the entire memory subsystem, we identify data structures that can be assigned to the same PLM element and share the same memory IPs. So, this logical organization of the physical memory IPs into parallel blocks can vary from one data structure to the other of the same PLM element.

Example. Consider two address-space compatible data structures b_i and b_j. Each requires one memory-write interface, while they require four and two memory-read interfaces, respectively. The PLM element is organized in four parallel blocks for b_i; this requires at least four memory IPs, which can be logically reorganized in two parallel blocks for b_j. □

To identify the minimum number of memory interfaces that can access the data at the same time allows us to minimize the number of parallel blocks and, in turn, of memory IPs.

Example. Assume a 512×32 array to be stored in the PLM. Process $input$ produces the data. Processes $compute_1$ and $compute_2$ need access to the data with two memory-read interfaces each. Without any additional information, the array requires four parallel blocks, two for $compute_1$ and two for $compute_2$. The designer may specify that, by construction, processes $compute_1$ and $compute_2$ never access the data at the same time (e.g. they execute serially). If so, two parallel blocks are sufficient since the two available memory-read interfaces can be used alternatively by $compute_1$ and $compute_2$. □

TABLE I
SUMMARY OF THE NOTATION USED IN THIS WORK.

SYMBOL	DEFINITION
b	PLM data structure of size *Height*×*Width*
R^b	Read interfaces for the data structure b
W^b	Write interfaces for the data structure b (i.e. *write blocks*)
L^b	Number of read interfaces for accessing data structure b based on sharing information
P^b	Total number of parallel blocks for data structure b
C_{PB}	Capacity of each parallel block
$Size$	Capacity of the selected physical memory IP

We propose the following approach to identify the minimum number of parallel blocks. During HLS, the designer specifies the read and write interfaces (R^b and W^b, respectively) to access each data structure b. Multiple write operations from one process can be supported only if they write consecutive addresses. This defines the number W^b of *write blocks*. Write operations from different processes can be supported only if they can share the same interfaces (i.e. only one process writes the data at each time). To identify the minimum number of read interfaces, we apply *graph coloring* to a *conflict graph* of the read interfaces based on compatibility information provided by the designer. Each node represents a read interface $r^b \in R^b$, while an edge is added between two interfaces r_i^b and r_j^b if the designer specifies that they may access the data at the same time. Let $proc(\cdot)$ be a function that returns the process associated with the corresponding interface $r^b \in R^b$. A conflict edge is added when: 1) the two interfaces refer to the same process, i.e. $proc(r_i^b) = proc(r_j^b)$; 2) processes $proc(r_i^b)$ and $proc(r_j^b)$ execute concurrently. We use a greedy heuristic to assign a color to each node of a graph such that two connected nodes have different colors. The resulting number of colors corresponds to the number $|L^b|$ of memory-read interfaces needed to access the data structure.

To determine the final number and capacity of the parallel blocks P^b for a data structure b of size $Height$, we analyze its access patterns and determine how to allocate the data. If the read patterns are deterministic and can be statically analyzed, we can distribute the data structure across many blocks; this *cyclic partitioning* technique [26] assigns consecutive values of the data structure to different blocks, as shown in the upper part of Fig. 5 (here, the number of parallel blocks is the *least common multiple* between W^b and $|L^b|$ ($P^b = lcm(W^b, |L^b|)$) and each block has capacity $C_{PB} = \lceil Height/P^b \rceil$). Otherwise, we must create identical copies of the data in the *write blocks* ($P^b = W^b \times |L^b|$), each with capacity $C_{PB} = Height$, as shown in the lower part of Fig. 5; in this way, each memory-read interface is assigned to a distinct parallel block and is guaranteed to access the data without conflicts [9] as long as the corresponding memory-write operations create consistent copies of the data in each bank.

Example. Consider array A0 of the Debayer accelerator in Listing 1 (6×2,048 integer values). Assume that process $input$ produces the data with four memory-write interfaces (writing

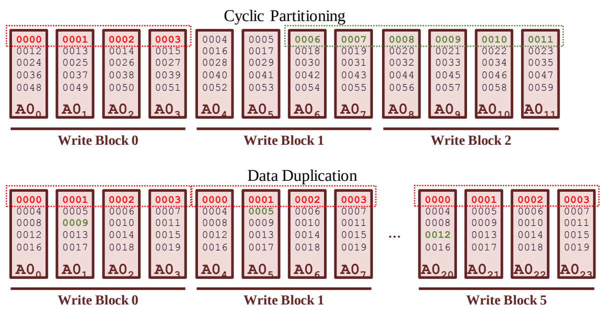
Cyclic Partitioning
A0_0 A0_1 A0_2 A0_3 A0_4 A0_5 A0_6 A0_7 A0_8 A0_9 A0_10 A0_11
Write Block 0
Write Block 1
Write Block 2
Data Duplication
A0_0 A0_1 A0_2 A0_3 A0_4 A0_5 A0_6 A0_7 ... A0_20 A0_21 A0_22 A0_23
Write Block 0
Write Block 1
Write Block 5

Fig. 5. Examples for the identification of parallel blocks in case of cyclic partitioning (top) and data duplication (bottom).

consecutive elements of the array), while process *compute* reads the data with six memory-ready interfaces ($|L^b| = 6$). The *write block* is composed of four parallel blocks (each of 3,072×32 bits) to allow the four parallel memory-write operations ($W^b = 4$). If the six memory-read operations also access consecutive addresses, array `A0` can be implemented with cyclic partitioning. In this case, only twelve banks are sufficient ($P^b = lcm(4, 6) = 12$) to store the data, and the data will be distributed over the entire set of banks, as shown in the upper part of Fig. 5. The size of each bank is 1,027×32 bits. On the contrary, if array `A0` is implemented with data duplication, the write block (and the corresponding data) must be replicated six times ($P^b = 4 \times 6 = 24$ parallel blocks, each of 3,072×32 bits) so that the six memory-read interfaces can access the data independently (see lower part of Fig. 5). □

V. Technology-aware Transformations

Based on the technology information in the memory library, we can determine the composition of each parallel block in terms of memory IPs. Specifically, the capacity of the parallel block C_{PB} may be larger than the capacity of the selected memory IP ($Size$). In this case, the data must be partitioned into multiple consecutive memory IPs. This technique is called *block partitioning* [26] and determines the number of memory IPs into each parallel block, which is equal to $\lceil C_{PB}/Size \rceil$.

Example. In the Debayer accelerator of Listing 1, the bi-dimensional array `A0` contains 6×2,048 integer elements (12,288 integer values) to be stored in the PLM. If only one memory-write and one memory-read interfaces are required to access `A0`, then a parallel block of capacity $C_{PB} = 12,288$ is sufficient. The array can be implemented with three 4,096×32 SRAMs in case of standard-cell technology and twenty-four 512×32 BRAMs in case of FPGA devices. □

A. Local Transformations

After defining the parallel blocks, we apply more optimizations on the bank architecture to obtain a data layout that maximizes the use of memories, while minimizing their cost.

Data merging. When a process has multiple memory-write interfaces that produce consecutive values, it is possible to write them in a single clock cycle. Additionally, if the aggregated bitwidth of these memory-write operations is supported by the memory IPs in the technology libraries, it is possible to write them with a single memory operation.

Example. Assume that the Debayer accelerator of Listing 1 is configured to process 128×128-pixel images. Array `A0` thus stores 768 values (6×128). Assume also that process *input* is implemented with implementation b (two memory-write operations) and connected to a 32-bit data bus and array `A0` can be represented with 16 bits. This implementation corresponds to two 16-bit write blocks (each of 384×16 bits) that need to be stored into two BRAMs. However, these parallel write blocks can be merged into a single 384×32-bit write block where the two values are concatenated and written in the same clock cycle. The read interface will simply select the proper part of data obtained from the block based on the given address (i.e. *signal slicing*). The resulting write block can be completely contained into a single BRAM, thus reducing the resource requirements for the implementation of the array. □

The input data to the memory IP is obtained by concatenating the values from the memory-write interfaces. For memory-read operations, the interface reads the entire memory line and provides only the part that is effectively requested.

Data splitting. This transformation can optimize the implementation of data structures whose bitwidth is different from the one of the memory IPs available in the library. Specifically, the data structure is split into smaller data tokens and written into different parallel blocks. When a memory-read operation is performed, all parts are read from the parallel blocks and concatenated to recompose the data.

Example. Consider the implementation of a 12,264×35 array on FPGA, where the available BRAMs have a maximum width of 32 bits. We thus need at least two parallel blocks to store this array. Using the 512×32 configuration of the BRAMs requires 24 BRAMs to store the array ($\lceil 12,264/512 \rceil$) with two parallel blocks. The resulting implementation consists of 48 BRAMs. Alternatively, we can use the 4,096×4 configuration, splitting each input value into nine parallel blocks. This corresponds to three parallel blocks ($\lceil 12,264/4,096 \rceil$) replicated nine times, for a total of 27 BRAMs. □

Each memory-write operation corresponds to multiple writes to each parallel block (at the same address). Similarly, a memory-read operation gets the different parts from the proper banks and recomposes the data by concatenating the values.

Optimization algorithm. To identify the proper combination of merge and split factors, we developed Algorithm 1. We first determine all candidates for block merging (`MergeCandidates`, line 3), which capture all the possibilities to combine the current number of parallel blocks P^b. For each of these candidates, we compare the resulting data structure with all memory IPs in the library (lines 4-11) in order to determine whether split (lines 6-8) or merge (lines 10-11) operations are possible. The resulting configuration of the blocks (obtained with the function `GetCfg`) is generated by considering the hypothetical implementation of the data structure with the current bank mem (lines 8 or 11). Finally, these configurations are sorted in ascending order, starting from the one with the minimal memory cost (line 12). This configuration determines which operation must be performed on the banks (split or merge); width and number of blocks are updated accordingly (lines 13 and 14, respectively).

Algorithm 1: Algorithm to determine merge/split operations to be performed on the parallel blocks.

```
1  Procedure DetermineBlockOptimization($b$, $P^b$, $width$)
       Data: $b$ is the buffer to be implemented
       Data: $w$ is the current width of the write block
       Data: $P^b$ is the current configuration of the write block
       Result: $\hat{w}$ is the updated width of the write block
       Result: $\hat{P}^b$ is the updated configuration of the write block
2      $L \leftarrow \emptyset$
3      foreach $m \in$ MergeCandidates($P^b$) do
4          foreach $mem \in MemoryLibrary$ do
5              if $m == 1$ then
6                  if $w >$ Width($mem$) then
7                      $split \leftarrow \lceil w/$Width($mem$)$\rceil$
8                      $L \leftarrow L \cup$ GetCfg($b, P^b, mem, w, 1, split$)
9              else
10                 if $w * m \geq$ Width($mem$) then
11                     $L \leftarrow L \cup$ GetCfg($b, P^b, mem, w, merge, 1$)
12     $cfg \leftarrow$ GetFirst(OrderByTotalArea($L$))
13     $\hat{w} \leftarrow w * cfg.merge \;/\; cfg.split$
14     $\hat{P}^b \leftarrow P^b * cfg.split \;/\; cfg.merge$
15     return $\{w, \hat{P}^b\}$
```

B. Analysis of Compatibility Information

The compatibility information provided by the designer is combined into a *Memory Compatibility Graph* (MCG), which captures the sharing opportunities among the data structures.

Definition. The Memory Compatibility Graph is a graph $MCG = (B, E)$ where each node $b \in B$ represents a data structure to be stored in the entire memory subsystem; an edge $e \in E$ connects two nodes when the corresponding data structures can be assigned to the same physical memory IPs. Each edge $e \in E$ is also annotated with the corresponding type of compatibility (e.g. address-space or memory-interface). ■

MCG is the dual of the *Memory Exclusion Graph* presented by Desnos *et al.* [32], which instead contains information on the data structures that cannot be allocated at the same address space in an MPSoC. A MCG with no compatibility edges corresponds to implementing each data structure in a dedicated PLM element. Increasing the number of edges into the MCG corresponds to increasing the number of compatible data structures. This can potentially increase the number of banks that can be reused across different data structures. An accurate compatibility graph is the key to optimize the memory subsystem of the accelerators. In most cases, the designer has to analyze the application's behavior or modify the interconnection topology of the accelerator to increase sharing possibilities.

Identification of Compatibilities. Control signals between the processes can be used to synchronize their execution and vary the lifetime of the data structures, thus increasing the situations in which it is possible to identify compatibilities.

Example. Assume two computational processes C_1 and C_2, each having a local 512×32 data structure to store temporary results. The standard implementation requires two memory blocks (e.g. two 512×32 BRAMs) for storing the two data structures because the two processes may access them at the same clock cycle. However, we can introduce additional signals between C_1 and C_2 to serialize the execution so that process C_2 can start only when process C_1 terminates and process C_1 can restart only after process C_2 ends. As a result, the two local data structures have non-overlapping lifetimes and can be stored in the same memory block (e.g. a single 512×32 BRAM). □

Two data structures can also share the memory banks when they are always accessed by mutually exclusive parts of the accelerator's code. The analyses of the code to be synthesized can also identify *local* data structures of a process that are never *active* at the same time, as well as the exact dependences between *input* and *output* ones. For example, when different computations are performed based on control conditions, different data structures may be read/written; in this case, they can share the same storage resources because these are always accessed in mutual exclusion. To identify such compatibilities, the designer has to perform an accurate dataflow analysis at different levels, i.e. both on the accelerator's interconnection topology and on the code of each process. On the other hand, it is also possible to use the following conservative assumptions:

- the lifetime of a data structure shared between two processes spans from the beginning of the producer execution to the end of the consumer execution; if there are multiple consumer processes, the termination of the last consumer process determines the end of the data structure lifetime.
- local data structures are alive from the beginning to the end of the process, when they store temporary local data, or from the beginning to the end of the entire accelerator execution, when they are used to maintain the state.

Moreover, when accelerators are never executed simultaneously, all data structures belonging to different accelerators are address-space compatible with each other. This allows the reuse of memory IPs across multiple accelerators.

Based on the characteristics of the available memory IPs, the designer can decide to implement the data structures in a larger memory IP.

Example. The two memory-interface compatible arrays `B0` and `B1` of the Debayer accelerator in Listing 1 (each having the size of 2,048×32 bits) can be implemented in standard-cell technology with a single 4,096×32 SRAM (with array `B1` starting in the second half of the memory block), instead of two 2,048×32 SRAMs. In our industrial 32nm CMOS technology, this reduces the memory area by almost 20%. Conversely, in FPGA technologies, BRAMs have a maximum capacity of 512×32 bits and we need multiple instances to virtually increase the size. Since both implementations require 10 BRAMs there is no difference in using one or the other. □

Memory-interface compatibilities can be also enabled by the different representations used by the designer who implements the communication mechanisms in the SystemC design. For example, the same ping-pong buffer can be represented as a single data structure with an offset or as two distinct arrays.

Example. Consider again the Debayer accelerator of Listing 1. The ping-pong buffer between processes *compute* and *output* is implemented with two distinct arrays `B0` and `B1` (each containing 2,048 integer values) to be accessed independently. However, it can be also represented as a single array, whose size is 2×2,048 integer values since it needs to contain both parts at different offsets. Even if the two solutions are functionally

Algorithm 2: Algorithm to determine the physical banks required to implement each clique.

```
Procedure DeterminePhysicalBanks($C_i$)
    Data: $C_i$ is the clique to be implemented
    Result: $N$ is number of memory IPs required to implement the clique
    Result: $Size$ is the size of each memory IP
    $P_i \leftarrow$ GetMinParallelBlocks($C_i$)
    $b \leftarrow$ GetFirst(OrderByNumBlocks(Nodes($C_i$),$P_i$))
    $N \leftarrow$ GetMinParallelBlocks($b$)  // current bank number
    $Size \leftarrow 0$        // current capacity of each bank
    if IsPartitioned($b$) then
        $Size \leftarrow$ GetDataSize($b$) / $N$
    else
        $Size \leftarrow$ GetDataSize($b$)
    foreach $b \in$ OrderByNumBlocks(Nodes($C_i$),$P_i$) do
        $S \leftarrow$ Floor($N$/ GetBanks($b$))
        if IsPartitioned($b$) then
            if GetDataSize($b$) / GetBanks($b$) $> Size * S$ then
                $Size \leftarrow$ GetDataSize($b$) / (GetBanks($b$) $*S$)
        else
            if GetDataSize($b$) $> Size * S$ then
                $Size \leftarrow$ GetDataSize($b$) /$S$
    $\langle N, Size\rangle \leftarrow$ SplitBanks($N$, $Size$)
    return $\langle N, Size\rangle$
```

equivalent and the total amount of data to be stored is the same, the first implementation requires two distinct PLM elements. □

With our approach, we use technology-aware information to determine the best implementation for the data structures rather than being limited by the way in which the designer defines them before HLS.

C. Global Transformations

Definition of Memory Subsystem. We target the problem of optimizing the memory subsystem at the system level by identifying possibilities for reusing banks across different data structures, even from different accelerators. For doing this, let us first recall the definition of *clique*.

Definition. A clique C_i of a graph $G = (V, E)$ is a non-empty subset of nodes (i.e. $C_i \subseteq V$) inducing a complete subgraph (not necessarily maximal) of G. ■

In our context, each clique C_i of the MCG represents a set of data structures that can share the same physical banks (i.e. a PLM element). Given a clique C_i, each data structure $b \in C_i$ is characterized by the minimum number of parallel blocks P^b needed to satisfy its combined requirements of read and write interfaces, as described in Section IV. This information is used to compute the organization of the memory architecture (in terms of number and characteristics of the physical banks) for the clique C_i and its cost A_i. We can thus formulate the system-level memory allocation problem as a *graph partitioning problem*.

Definition. Let $MCG = (B, E)$ be the MCG associated with the set of data structures B. The optimal memory allocation consists in finding a partition of B into n disjoint cliques $C = (C_1, \ldots, C_n)$ of minimum cost. The cost A of a partition is:

$$A = \sum_{i=1}^{n} A_i \qquad (1)$$

where A_i is the cost of clique C_i. The cost A of the entire memory subsystem is the value to be minimized. ■

Algorithm 2 determines the number of banks and their size to efficiently implement each clique C_i. In our implementation, each clique has a homogeneous organization, where all physical banks have the same size. This has multiple advantages: it eases the reorganization of the banks to store different data structures; it benefits the floorplanning of the modules by enforcing a regular design [33]; and it simplifies the logic to create the associated memory controller. Specifically, for each data structure contained into the clique, we compute its minimum number of parallel blocks (GetMinParallelBlocks) with the approach discussed in Section IV. We sort the data structures in a descending order (OrderByNumBlock), from the one that requires the maximum number of parallel blocks to the one with the minimum number. This determines the maximum number of parallel blocks and, thus, the minimum number of banks that are required to provide this bandwidth (lines 3-5). We also determine an initial size for these banks based on the data allocation strategy to be implemented (lines 6-9).

Then, we analyze all data structures following the same descending order and we seek for opportunities to reuse the banks. In particular, when a data structure requires a lower number of parallel blocks, it can reuse the exceeding ones in series to virtually increase the capacity of the parallel blocks (lines 11-17). We also check if the data structure can fit into this new configuration (line 12). If not, the size of the banks is updated accordingly to the data allocation strategy.

Example. Let's assume that the current number of banks is four (each having size of 128 words) and that we need to store a data structure which has size of 900 integer values, partitioned in three parallel blocks. The existing four banks cannot be redistributed into the parallel blocks and for this reason $S = 1$ (line 11). Since this organization is not sufficient to store the entire data structure (line 13), each of the four banks is expanded to store 300 integer values (line 14). □

On the other hand, if the banks can be rearranged and reused, it is not necessary to change their size.

Example. Let's assume that we now have four banks, which have a size of 300 words, and we need to store a data structure of 512 integer values, which requires two parallel banks ($P = 2$) but with data duplication. The four banks can be rearranged in two blocks of two banks each ($S = 2$); the two serial banks provides a virtual capacity of 600 words for each parallel block. Here we can store the data with a serial reorganization of the banks without any changes to their size (line 16). □

Finally, if the current size is greater than the largest memory IP of the library, the banks are implemented with the necessary number of serial memory IPs (SplitBanks, line 18).

Memory Footprint Minimization. To obtain an efficient system-level allocation of the memory elements, we partition the MCG in cliques such that the total cost is minimized (see Equation 1). The cost of each clique is computed as the aggregated requirement of resources (either silicon area or number of BRAMs) for its implementation. Specifically, we enumerate all admissible cliques and compute the bank organization for each of them with the above procedure. Based on the information in the technology library, we associate the resulting cost to each clique, which is expressed as μm^2 for standard-cell technologies and as a number of BRAMs for FPGA technologies.

Selecting the best MCG partition can be formulated as a *clique partitioning* problem, which is NP-hard. The goal is to minimize the memory cost of the system, defined as

$$MEM = \sum_{i=1}^{N} A_i * c_i \tag{2}$$

where N is the total number of admissible cliques, A_i is the resource requirement of the clique C_i, while c_i is a Boolean variable that specifies whether the clique is included in the final solution or not.

Given each data structure b_i and assuming that C^i represents the set of cliques that contains b_i, we need to ensure that the data structure is contained into only one clique. Hence, for each data structure b_i, we impose the following constraint:

$$\forall b_i : \sum_{n \in C^i} c_n = 1 \tag{3}$$

Each of the resulting cliques requires the generation of the logic to convert the requests from the memory interfaces into the proper requests to the actual physical banks.

VI. Generation of the PLM Controller

This section describes the architecture and the design of the *flexible memory controller* that we use to coordinate accesses between the accelerator logic and the PLM element architectures that we generate. From the accelerator logic's viewpoint, each data structure b has a certain number of memory interfaces, based on the requirements of the processes that need to access it. An active request performed by the accelerator logic on one of these interfaces corresponds to a memory operation on the data structure b. The physical implementation of the PLM is transparent to the accelerator logic, which simply specifies an offset (*logical address*) with respect to the beginning of the data structure.

Based on the organization of the PLM element where the data structure is stored, two steps must be performed for each memory operation: 1) identify which physical bank effectively contains the requested data and activate the memory operation only on that one; and 2) translate the logical address into the corresponding *physical address* of the bank.

Example. Let us consider array `A0` of the Debayer accelerator of Listing 1. Let us assume that process *compute* needs to read the 6^{th} element of this array. The corresponding read interface will have an active request with logical address set to 5 (i.e. `A0[5]`). If array `A0` is implemented with cyclic partitioning over two banks (e.g. implementation E with two parallel reads), this corresponds to reading the third element of the second bank. If cyclic partitioning is performed instead over four banks (e.g. implementation G with four parallel reads), this corresponds to reading the third element of the second bank. Conversely, if array `A0` is implemented with data duplication, the access will refer to the sixth element of any of the banks. □

Each memory interface provides five main signals:

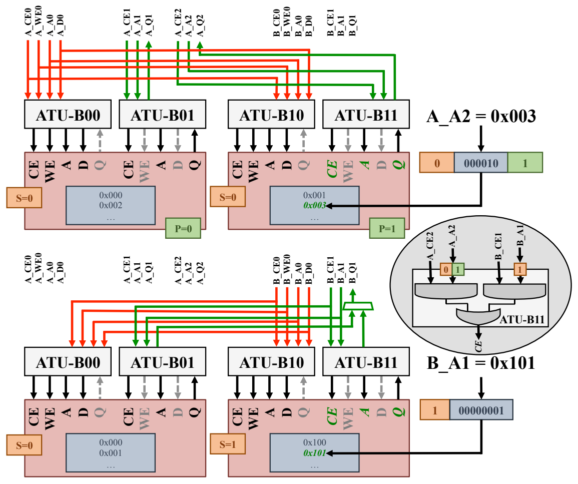

Fig. 6. Architecture of the *Address Translation Unit* (ATU) proposed in this work and address decomposition.

- *Chip Enable* (CE): it indicates the presence of an active request on the associated interface.
- *Address* (A): for processes, it represents the logical address to be accessed, while, for memories, it corresponds to the physical address to be accessed. In case of processes, the bitwidth of A corresponds to the size of the data to be accessed, since they have no information about the memory organization. In case of memories, the bitwidth of A corresponds to the size of the memory IP.
- *Input Data* (D): it represents the data to be written into the memory and thus it is present only in write interfaces.
- *Output Data* (Q): it represents the data read from the memory and thus it is present only in read interfaces.
- *Write Enable* (WE): when active, the request is a write operation and the corresponding input data D is valid.

In case of cyclic partitioning over P parallel blocks, the function to translate a logical address into the corresponding *block address* is: $block(i) = logical(i)/P$. The logic to implement the translation is greatly simplified if P is a power of two because the operation can be transformed into a hardware-friendly shift operation. In case of data duplication, all banks contain the same copy of the data structure and the corresponding function is: $block(i) = logical(i)$. When the address $block(i)$ is larger than the bank capacity $Size$, the physical address of the actual bank is determined as: $physical(i) = block(i) \bmod Size$. Again, this operation is greatly simplified if the bank capacity is a power of two.

With this approach, we need to synthesize the accelerators only once and we can combine them in multiple scenarios without any changes. For this, we use a flexible memory controller with an *Address Translation Unit* (ATU) that we generate directly in RTL from a high-level template. The ATU is generated for each port of the memory banks and is composed of two parts: 1) the *activation unit* determines if the memory interface is accessing a value that is effectively stored in the bank. 2) the *translation unit* translates the logical address from the accelerator logic into the physical address of the bank. Specifically, each bank has tags assigned during the controller generation, based on the configuration of the data structures. The activation unit then analyzes the memory request corresponding to a data structure (i.e. *Chip Enable* and *Address*) to determine whether it matches with the related tags. By construction, in each clique only one request is active during the same clock cycle time on any given bank port. Since no more than one activation unit is active at each clock cycle, the *CE* signal of one port is the output of an OR gate with the results of all corresponding activation units as input (Fig. 6). This signal is also used to control the multiplexing of the translation units and the values connected to the *Input Data* port of the memory when writing.

Example. The PLM controller of Fig. 6 accesses two banks for two different data structures (A and B) that are never accessed at the same time. In the upper part, array A is accessed with two memory-read interfaces and it is thus allocated with cyclic partitioning over the two banks (tag P). The less significant bit of the address is used to identify which bank is accessed by each operation. In the lower part, array B is larger than the single bank and it is thus stored with block partitioning over the two banks (tag S). The most significant bit of the address identifies which bank is accessed by the memory-read operation. □

The ATU is a specialized component similar to the Memory Management Unit (MMU) for processor cores. Differently from the MMU that uses a TLB to convert the addresses, the ATU design is customized for the specific data structure [34] and to guarantee that the translation does not introduce any additional cycle. The ATU architecture is greatly simplified if both the number of parallel blocks P^b and the size of each physical bank are a power of two. In this case, the logical address to access a data structure b of size H^b is composed of $\lceil log_2(H^b)\rceil$ bits that can be decomposed as follows:

$$\lceil log_2(H^b)\rceil = \{\lceil log_2(S)\rceil, log_2(Size), log_2(P^b)\}$$

Hence, the translation unit simply implements signal slicing.

VII. Experimental Results

We implemented our methodology in Mnemosyne[1], a C++ prototype tool where the problem described in Section V-C has been formulated as an Integer Linear Programming (ILP) problem and solved with Coin-Or [35].

A. Experimental Setup

We selected and analyzed several computational kernels from two recently-released benchmark suites, i.e. Perfect [28] and CortexSuite [29]. These suites contain kernels of various domains of interest, ranging from computer vision to machine learning. The selected benchmarks, shown in Table II, represent a variety of memory-access patterns and are suitable for memory-related optimizations. We designed synthesizable SystemC descriptions of these accelerators starting from the C-based implementations provided in the benchmark suites. The structure of all accelerators follows the template described in Section II, with multiple communicating processes.

In our experiments, we targeted two different technologies:

[1]*Mnemosyne* was the personification of memory in Greek mythology.

TABLE II
Details of the applications considered in this work.

Suite	Benchmark	Details	Data Size (MB)
Perfect [28]	Sort	Quicksort of 1,024 vectors of 1,024 fixed-point elements each	4.00
	FFT-1D	One dimensional FFT on 2^{16} fixed-point elements	0.25
	FFT-2D	Two dimensional FFT on 4,096×4,096 matrix of fixed-point values	64.00
	Debayer	Debayering of a 2,048×2,048-pixel image	16.00
	Lucas Kanade	Registration algorithm for a 2,048×2,048-pixel image	32.00
	Change Detection	Detecting regions of change in five 2,048×2,048-pixel images	320.00
	Interpolation 1	Polar format algorithm (kernel 1, 2,048×2,048-pixel image)	32.04
	Interpolation 2	Polar format algorithm (kernel 2, 2,048×2,048-pixel image)	64.01
	Backprojection	Back projection algorithm (2,048×2,048-pixel image)	256.04
CortexSuite [29]	Disparity	Computing pixel distance between two 1,920×1,080-pixel images	15.82
	Principal Component Analysis (PCA)	Feature extraction from a 5000×1059 matrix of integer values	20.19
	Restricted Boltzmann Machine (RBM)	Model training and prediction on 100 movies for 1,000 users	3.81
	Superresolution Reconstruction (SRR)	Creation of a high-resolution image from 16 low-resolution images	4.76

- **CMOS:** an industrial 32nm CMOS process with the corresponding memory generator to create SRAMs of different sizes. For this, we generated a library of 18 SRAMs, ranging from 128×16 to 2,048×64. Synopsys Design Compiler J-2014.09-SP2 is used for logic synthesis, with a target frequency of 1 GHz.
- **FPGA:** a Xilinx Virtex-7 FPGA device. For this, we used dual-port 16 Kb BRAMs as memory blocks (in the six available configurations that have different port aspect ratios [17]). Xilinx Vivado 2015.2 is used for logic synthesis, with a target frequency of 100 MHz.

We used Cadence C-to-Silicon 14.2 to generate implementations for the accelerator logic. Table III reports the number of data structures to be stored in the PLM of each accelerator and their total size. It also reports the resource requirements for the *Baseline* versions of the accelerators (i.e. without any proposed optimizations). We report information for the accelerator logic (Logic) and the memory (PLM) for both technologies (CMOS and FPGA). We then used Mnemosyne to design the memory subsystem for these accelerators in different experiments. As part of each experiment, we performed RTL simulation with Mentor ModelSim 10.1 to evaluate the functional correctness and performance of the resulting accelerators. This analysis confirmed that all accelerator designs work correctly without any possible performance overhead due to the PLM controller. In case of CMOS technology, we used these simulations also to collect the switching activity of the generated netlists in order to perform power analysis with SAIF back-annotations. We also tested our accelerators in an FPGA-based full-system prototype, where the processor core

TABLE III
DETAILS OF ACCELERATORS' IMPLEMENTATIONS.

BENCHMARK	DATA STRUCTURES		CMOS		FPGA RESOURCES			
			LOGIC	PLM	LOGIC			PLM
	(#)	(KB)	(μm^2)	(μm^2)	LUTs	FFs	DSPs	BRAMs
Sort	6	24.00	54,408	210,640	33,312	23,240	0	12
FFT-1D	10	40.00	93,558	299,605	9,357	4,295	144	20
FFT-2D	4	128.00	44,785	785,210	7,195	2,109	59	64
Debayer	4	95.86	9,436	684,355	5,184	1,888	5	48
Lucas Kanade	11	20.28	42,488	319,629	6,831	3,471	48	18
Change Det.	10	62.13	90,420	1,305,130	20,692	6,582	121	70
Interpolation 1	6	48.05	136,689	343,144	25,346	6,517	84	25
Interpolation 2	7	64.05	123,599	441,296	25,649	6,476	55	33
Backprojection	8	99.00	129,153	639,922	19,169	5,031	126	50
Disparity	11	145.56	24,082	2,196,741	14,225	4,033	15	153
PCA	3	117.19	20,539	1,140,592	6,971	2,410	21	80
RBM	8	65.27	24,804	2,888,981	11,544	8,853	6	67
SRR	32	76.20	36,631	1,222,102	11,059	4,264	36	71

running a complete Linux OS executes software applications that invoke the accelerators through device drivers [7]. The accelerators that share memory IPs are serialized by their device driver so that they never execute at the same time. They also share the same interface with the rest of the system (i.e. DMA controller and configuration registers in Fig. 1).

B. Single-Accelerator Optimization

In the first set of experiments, we used MNEMOSYNE to analyze the impact of the proposed optimizations on the accelerator PLMs. We performed four experiments for each accelerator: *Baseline* with no optimizations; *Compatibility* where we use compatibility information to share memory IPs and the computation of the parallel blocks is performed with a conservative approach; *Coloring* where the computation of parallel blocks is performed with our graph coloring-based approach (Section IV); and *Final* with all our optimizations active at the same time. Fig. 7 and Fig. 8 show the results for the two target technologies, respectively. Each bar represents the cost of the corresponding memory subsystem (either in terms of μm^2 for CMOS or number of BRAMs for FPGA), normalized with respect to the *Baseline* one.

Baseline results show that our methodology can fully automate the design of accelerators' memory subsystems. In these cases, the designer does not provide any information about compatibilities and MNEMOSYNE generates a conservative memory subsystem, at the cost of more area.

Compatibility results show that compatibility information about data structures can be used to reduce the area of the memory subsystem. For example, we obtain almost a 20% area reduction in CMOS technology for *Sort*, *FFT-1D*, and *Lucas Kanade*. However, there is no area reduction when targeting FPGA for the same benchmarks because the data structures are larger than the capacity of the BRAMs. Note that these accelerators were designed for high-throughput processing. Since all data structures are used for the entire execution of the accelerator, there is no potential for address-space compatibility. Hence, all area savings are obtained by exploiting memory-interface compatibilities on ping-pong buffers.

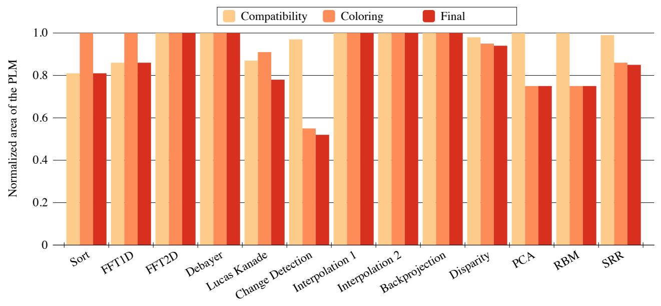

Fig. 7. Normalized area (with respect to *Baseline*) for each memory subsystem when the accelerators are designed separately (CMOS technology).

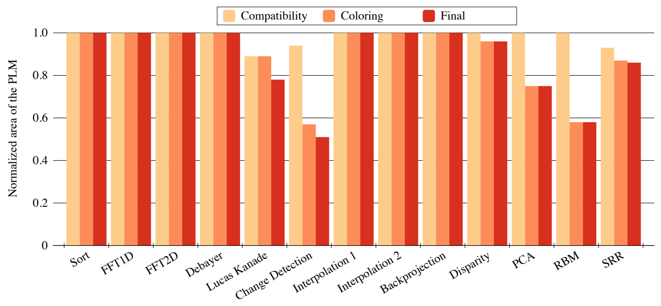

Fig. 8. Normalized area (with respect to *Baseline*) for each memory subsystem when the accelerators are designed separately (FPGA technology).

Coloring results show that the proper identification of parallel blocks effectively reduces the area of the memory IPs by 20% on average, especially when the same data structure is accessed by multiple processes. This optimization is particularly efficient for *Change Detection* (up to 45% of area saving) and most of CORTEXSUITE accelerators. Indeed, in these accelerators, we avoid unnecessary duplication of the parallel blocks (and data) by properly sharing memory interfaces between the different processes.

Final results show that the combined optimizations can reduce the memory area by around 20% and in some cases up to 50% (e.g. *Change Detection*). Similar results are obtained in terms of power consumption, which is proportional to the amount of memory required to implement the PLM.

C. Multi-Accelerator Optimization

A Case Study: the RBM application. The implementation of the Restricted Boltzmann Machine algorithm in the CORTEXSUITE is used for predicting movie ratings based on a data set of previous users. To analyze the possibilities of sharing memory IPs among accelerators, we redesigned the RBM accelerator previously used by splitting it into two distinct ones to be optimized separately. The TRAIN accelerator analyzes the training data to build the underlying model (i.e. a bipartite neural network), while the PREDICT accelerator uses this model to make predictions on new users. For each accelerator, we created three different versions, each capable of locally storing a variable number of movie ratings and the correponding part of the RBM model (from 10 to 100 movies). The memory footprint of these data structures ranges from 64 to 256 KB. This affects the size of the PLMs and changes the number of DMA data transfers. The resulting

TABLE IV
AREA SAVINGS FOR THE RBM CASE STUDY.

Tech.		TRAIN V0	TRAIN V1	TRAIN V2
CMOS	PREDICT V0	-37.15%	-29.84%	-24.33%
	PREDICT V1	-39.25%	-37.29%	-29.25%
	PREDICT V2	-32.21%	-38.20%	-37.39%
FPGA	PREDICT V0	-20.90%	-28.28%	-17.39%
	PREDICT V1	-25.27%	-42.28%	-28.11%
	PREDICT V2	-45.39%	-30.06%	-43.40%

speed-up varies between 10× and 20×. Since, however, the same computational kernel is repeated over the entire set of data just with a different number of iterations proportional to the PLM size, to vary the PLM size has almost no impact on the area of the accelerator logic.

First, we used MNEMOSYNE to generate distinct PLMs for each accelerator. This is the baseline for the following experiments. Then, we combined the different versions of the two accelerators and we used MNEMOSYNE to generate the memory subsystem of each of these combinations, allowing the possibility to share memory IPs. As reported in Table IV, the sharing of the memory IPs yields area savings ranging from 18% to more than 45%. Better results are obtained when the data structures of the two accelerators have similar amount of data to be stored. In the other cases, the accelerator with the largest data structures dominates the cost of the memory subsystem. Results show that MNEMOSYNE can derive an optimal PLM organization for the given target technology and memory library. For CMOS technology, the configuration with TRAIN V0 and PREDICT V1 is the one with the biggest area improvement (-39.25%), while, for FPGA technology, we achieve the best results (-45.39%) with TRAIN V0 and PREDICT V2. This area saving can be used to implement bigger PLMs for each accelerator (improving the overall performance of the RBM application) at the same total memory cost. Note that the performance of the RBM application is not affected because the two phases are always executed serially even without the reuse of memory IPs.

Resource-Oriented SoCs. To further evaluate the impact of memory sharing across accelerators for reducing the resource requirements, we designed four additional systems: *Required*, *WAMI*, *SAR*, and *Cortex*. In each system, multiple accelerators are combined as follows: *Required* contains accelerators *Sort*, *FFT-1D*, and *FFT-2D*. *WAMI* contains accelerators *Debayer*, *Lucas Kanade*, and *Change Detection*. *SAR* contains accelerators *Interpolation 1*, *Interpolation 2*, and *Backprojection*. Finally, *Cortex* contains the four CORTEXSUITE accelerators.

For each of these scenarios, we reused the accelerator logic that we synthesized for the previous experiments with no modifications. The memory subsystem is generated with MNEMOSYNE both with and without activating the sharing of the memory IPs across accelerators (SHARING and NO SHARING, respectively). Our flexible memory controller is used to coordinate the memory requests between the accelerator logic and the different PLM elements. Specifically, to synthesize the accelerators independently (NO SHARING), we used MNEMOSYNE with no address-space compatibilities between data structures from different accelerators. For each accelerator of each scenario, this generates the same PLMs obtained in the single-accelerator scenarios. To share memory IPs between accelerators, we apply all proposed optimizations and we specified address-space compatibilities between data structures of different accelerators. In this case, we create a single memory subsystem for each scenario, where MNEMOSYNE identifies the best configuration of the banks. The experiments are replicated for the two target technologies, e.g. CMOS and FPGA. Table V shows the results for these experiments. We report the total number of data structures and the corresponding memory footprint in KB. Then, for each scenario, we report the number of PLM elements (i.e cliques) that have been generated (*#Ctrl*), along with the size (in KB) and the cost of the entire memory subsystem. Clearly, when no sharing is activated, each data structure is implemented with its own PLM element. Hence, the number of resulting PLM elements corresponds to the number of initial data structures. Activating sharing across accelerators allows us to reduce the number of PLM elements and their total size by implementing more data structures with the same physical banks. Moreover, the total area and power are generally reduced by more than 30% and 20%, respectively. Best results are obtained for applications that have similar data structures in terms of width and height (e.g. *SAR* and *WAMI*). In these cases, the same configuration of the banks can be instantiated only once and reused with almost no modifications (i.e. no area overhead).

VIII. RELATED WORK

The specialization of the memory subsystem has been widely studied since it critical to improve performance, while reducing both area and power [27], [36]. Recently, many approaches have been proposed to promote the use of HLS in the design of specialized accelerators, but memory aspects are often ignored. Liu *et al.* compose pre-characterized components to create a Pareto set of the entire system [23]. Li *et al.* extend a method to compose pre-characterized IPs through a pre-defined architectural template [24] to the design of the memory subsystem, but without considering design parameters like the number of memory interfaces [37]. Panda *et al.* study how to create custom architectures and improve the system's performance, both in terms of memory organization and data layout [36]. Benini *et al.* propose a technique to customize the memory subsystem given an application profiling while accounting for layout information to minimize power consumption [38]. These approaches can be extended to hardware accelerators. Baradaran and Diniz propose *data duplication* and *data distribution* to improve performance while considering the capacity and bandwidth constraints of the storage resources of the accelerators [39]. Their compiler-based approach has been extended with a theory for data partitioning to support more complex data access patterns [26], [27]. All these approaches are complementary to our work and can be used to improve our ATU design. However, in these cases, varying the PLM micro-architecture to share the memory IPs across many accelerators would require multiple iterations through the HLS steps. In our methodology,

TABLE V
SCENARIOS WITH MULTIPLE ACCELERATORS SHARING THE MEMORY IPS.

Bench.	Data structures		Industrial 32nm CMOS								Xilinx Virtex-7 FPGA					
			NO SHARING		SHARING						NO SHARING		SHARING			
	(#)	(KB)	PLM (KB)	Area (μm^2)	#Ctrl.	PLM (KB)	Area (μm^2)	Diff (%)	Power (mW)	Diff (%)	PLM (KB)	BRAMs	#Ctrl.	PLM (KB)	BRAMs	Diff (%)
Required	13	192.00	192.00	1,295,454	4	140.00	880,237	-32.05	102.86	-22.72	192.00	96	4	136.00	68	-29.17
WAMI	25	178.27	221.00	1,692,080	8	131.00	948,812	-43.93	213.19	-20.85	212.00	106	8	90.00	45	-57.55
SAR	21	211.10	213.00	1,424,362	8	100.00	639,922	-55.07	150.81	-32.10	216.00	108	8	100.00	50	-53.07
Cortex	54	404.23	1,1018.00	7,053,948	32	653.50	4,484,684	-36.42	1,060.95	-20.10	690.00	345	32	368.00	184	-46.67

instead, the global optimization of the memory subsystem is independent from the optimization of each accelerator (see Fig. 2). In fact, as shown in Section VII, the logic of each accelerator is reused without any modifications when creating multi-accelerator scenarios.

Abdelhadi and Lemieux analyze various techniques to perform multiple memory operations in the same clock cycle [14]. Among these, raising the memory frequency is usually limited by the technology, while bank arbitration affects the accelerator performance. Hence, we focus on register-based RAM (created by HLS tools) and conflict-free banking (created by MNEMOSYNE). Similar architectures are created by combining data reuse, memory partitioning, and memory merging for FPGA [40]. Zuo *et al.* extend this approach to the concurrent optimization of multiple processes but only to optimize fine-grained communication, not the memory elements [41]. Desnos *et al.* explore the sharing and reusing of memory elements in MPSoCs to minimize memory allocation [32], but do not consider multi-bank architectures and the constraints imposed by the limited number of the physical ports. Vasiljevic and Chow explore the possibilities to store multiple data structures in the same BRAM through *buffer packing* [42]. All these solutions are applied before HLS. This is efficient and elegant, but it has limitations in case of multiple accelerators to be jointly designed and optimized. Our approach, instead, decouples the design of the components from their PLMs. Hence, it enables the system-level optimization of the memory subsystem with multiple memory IPs, eventually shared among different data structures.

Some architectures are aimed at sharing memory IPs across many accelerators. Lyons *et al.* propose the *Accelerator Store*, where a predefined set of memory IPs are dynamically assigned to the accelerators [10]. This requires the memories to be latency insensitive [30] since their controller may introduce an overhead. Cong *et al.* propose a NoC-based architecture where each tile contains small blocks that are dynamically composed to create larger accelerators and memory blocks are shared in each tile [43]. Instead, we generate the memory subsystem for large accelerators with multiple memory interfaces. Our work is more similar to the work by Cong and Xiao [44], who design a crossbar to connect the accelerators to a set of memory banks so that each accelerator can access multiple ports. However, our specialized PLM micro-architecture guarantees no performance overhead. It also allows designers to tailor the memory IPs to the data structures and to apply technology-related optimizations to this micro-architecture.

IX. CONCLUSIONS

We presented a methodology for the system-level design of accelerator local memory and a supporting CAD flow which combines a new tool, MNEMOSYNE, with commercial high-level synthesis tools. With our approach we can design and optimize the local memory of multiple accelerators at the system level, by identifying possibilities to share physical memory IPs across many data structures. This allows us to achieve area savings up to 55% compared to the case where the accelerators are designed separately.

REFERENCES

[1] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC Digest of Technical Papers*, Feb. 2014, pp. 10–14.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Communication of the ACM*, vol. 54, pp. 67–77, May 2011.

[3] M. Taylor, "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse," in *Proc. of the Design Automation Conf.*, Jun. 2012, pp. 1131–1136.

[4] T. Chen *et al.*, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *Proc. of Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 269–284.

[5] Z. Wang, K. H. Lee, and N. Verma, "Hardware specialization in low-power sensing applications to address energy and resilience," *Journal of Signal Processing Systems*, vol. 78, no. 1, pp. 49–62, 2014.

[6] C. Zhang *et al.*, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proc. of the Int. Symp. on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[7] E. Cota *et al.*, "An analysis of accelerator coupling in heterogeneous architectures," in *Proc. of the Design Automation Conf.*, Jun. 2015, pp. 1–6.

[8] J. Cong *et al.*, "Architecture support for accelerator-rich CMPs," in *Proc. of the Design Automation Conf.*, 2012, pp. 843–849.

[9] B. Li, Z. Fang, and R. Iyer, "Template-based memory access engine for accelerators in SoCs," in *Proc. of the Asian and South-Pacific Design Automation Conf.*, Jan 2011, pp. 147–153.

[10] M. Lyons *et al.*, "The accelerator store: A shared memory framework for accelerator-based systems," *ACM Trans. on Architecture and Code Optimization*, vol. 8, no. 4, pp. 48:1–48:22, Jan. 2012.

[11] R. Iyer, "Accelerator-rich architectures: Implications, opportunities and challenges," in *Proc. of the Asian and South-Pacific Design Automation Conf.*, 2012, pp. 106–107.

[12] P. H. Wang *et al.*, "EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System," *Proc. of the Conf. on Programming Language Design and Implementation*, pp. 156–166, Jun. 2007.

[13] N. Goulding-Hotta *et al.*, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *IEEE Micro*, vol. 31, no. 2, pp. 86–95, Mar. 2011.

[14] A. M. Abdelhadi and G. G. Lemieux, "Modular multi-ported SRAM-based memories," in *Proc. of the Int. Symp. on Field-Programmable Gate Arrays*, Feb. 2014, pp. 35–44.

[15] Y. Tatsumi and H.-J. Mattausch, "Fast quadratic increase of multiport-storage-cell area with port number," *Electronics Letters*, vol. 35, no. 25, pp. 2185–2187, 1999.

[16] C. Pilato *et al.*, "System-level memory optimization for high-level synthesis of component-based SoCs," in *Proc. of the Int. Conf. on Hardware/Software Codesign and System Synthesis*, Oct. 2014, pp. 1–10.
[17] Xilinx, "7 Series FPGAs Overview (DS180)," http://www.xilinx.com/.
[18] B. Bailey and G. Martin, *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer, 2006.
[19] L. P. Carloni, "The case for embedded scalable platforms," in *Proc. of the Design Automation Conf.*, Jun. 2016, pp. 17:1–17:6.
[20] P. Coussy and A. Morawiec, *High-level synthesis: from algorithm to digital circuit*. Springer, 2008.
[21] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, 2009.
[22] M. Fingeroff, *High-level Synthesis Blue Book*. Xlibris Corp., 2010.
[23] H.-Y. Liu, M. Petracca, and L. P. Carloni, "Compositional system-level design exploration with planning of high-level synthesis," in *Proc. of Design, Automation and Test in Europe Conf.*, Mar. 2012, pp. 641–646.
[24] S. Li *et al.*, "System level synthesis of hardware for DSP applications using pre-characterized function implementations," in *Proc. of the Int. Conf. on Hardware/Software Codesign and System Synthesis*, Oct. 2013, pp. 1–10.
[25] B. Carrion Schafer, "Probabilistic multi-knob high-level synthesis design space exploration acceleration," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2015.
[26] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proc. of the Int. Symp. on Field-Programmable Gate Arrays*, Feb. 2014, pp. 199–208.
[27] J. Cong *et al.*, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Trans. on Design Autom. of Electronic Systems*, vol. 16, no. 2, pp. 15:1–15:25, Apr. 2011.
[28] K. Barker *et al.*, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*, Pacific Northwest National Laboratory and Georgia Tech Research Institute, Dec. 2013, hpc.pnnl.gov/projects/PERFECT/.
[29] S. Thomas *et al.*, "CortexSuite: A synthetic brain benchmark suite," in *Proc. of Int. Symp. on Workload Characterization*, Oct 2014, pp. 76–79.
[30] L. Carloni, "From Latency-Insensitive Design to Communication-Based System-Level Design," *Proc. of the IEEE*, vol. 103, no. 11, pp. 2133–2151, 2015.
[31] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
[32] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Pre- and post-scheduling memory allocation strategies on MPSoCs," in *Proc. of the Electronic System Level Synthesis Conf.*, Jun. 2013, pp. 1–6.
[33] S. N. Adya and I. L. Markov, "Consistent placement of macro-blocks using floorplanning and standard-cell placement," in *Proc. of the Int. Symp. on Physical Design*, 2002, pp. 12–17.
[34] F. Conti *et al.*, "He-P2012: Performance and Energy Exploration of Architecturally Heterogeneous Many-Cores," *Journal of Signal Processing Systems*, pp. 1–16, 2015.
[35] "Common infrastructure for operations research," www.coin-or.org.
[36] P. R. Panda *et al.*, "Data memory organization and optimizations in application-specific systems," *IEEE Design & Test of Computers*, vol. 18, no. 3, pp. 56–68, 2001.
[37] S. Li and A. Hemani, "Memory allocation and optimization in system-level architectural synthesis," in *Proc. of ReCoSoC*, Jul. 2013, pp. 1–7.
[38] L. Benini *et al.*, "Layout-driven memory synthesis for embedded systems-on-chip," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 10, no. 2, pp. 96–105, Apr. 2002.
[39] N. Baradaran and P. C. Diniz, "A compiler approach to managing storage and memory bandwidth in configurable architectures," *ACM Trans. on Design Autom. of Electronic Systems*, vol. 13, no. 4, pp. 1–26, Oct. 2008.
[40] Y. Wang *et al.*, "An integrated and automated memory optimization flow for FPGA behavioral synthesis," in *Proc. of the Asian and South-Pacific Design Automation Conf.*, Jan. 2012, pp. 257–262.
[41] W. Zuo *et al.*, "Improving high-level synthesis optimization opportunity through polyhedral transformations," in *Proc. of the Int. Symp. on Field-Programmable Gate Arrays*, Jan. 2013, pp. 9–18.
[42] J. Vasiljevic and P. Chow, "Using buffer-to-BRAM mapping approaches to trade-off throughput vs. memory use," in *Proc. of the Int. Conf. on Field Programmable Logic and Applications*, Sept 2014, pp. 1–8.
[43] J. Cong *et al.*, "CHARM: A Composable Heterogeneous Accelerator-rich Microprocessor," in *Proc. of the Int. Symp. on Low Power Electronics and Design*, 2012, pp. 379–384.
[44] J. Cong and B. Xiao, "Optimization of interconnects between accelerators and shared memories in dark silicon," in *Proc. of the Int. Conf. on Computer-Aided Design*, Nov. 2013, pp. 630–637.

Christian Pilato received the Laurea degree in Computer Engineering and the Ph.D. degree in Information Technology from the Politecnico di Milano, Italy, in 2007 and 2011, respectively. From 2013 to 2016, he was a Post-doc Research Scientist with the Department of Computer Science, Columbia University, New York, NY, USA. He is currently a Post-doc at the University of Lugano, Switzerland. His research interests include high-level synthesis, reconfigurable systems and system-on-chip architectures, with emphasis on memory aspects. In 2014 Dr. Pilato served as program chair of the Embedded and Ubiquitous Conference (EUC) and he is currently involved in the program committees of many conferences on embedded systems, CAD, and reconfigurable architectures (e.g., FPL, DATE, CASES). He is a member of the Association for Computing Machinery.

Paolo Mantovani received the BS degree (*Summa Cum Laude*) in Electronic Engineering from the Università di Bologna, Italy, in 2008, and the MS degree (*Summa Cum Laude*) in Electronic Engineering from the Politecnico di Torino, Italy, in 2010. He is currently a Ph.D. candidate of the Department of Computer Science at Columbia University, New York. His research focuses on system-level design of heterogeneous system-on-chip platforms and computer architectures for embedded systems.

Giuseppe Di Guglielmo received the Laurea degree (*Summa Cum Laude*) and the Ph.D. in Computer Science from the Università di Verona, Italy, in 2005 and 2009, respectively. He is currently an Associate Research Scientist with the Department of Computer Science, Columbia University, New York, NY, USA. He has authored over 40 publications. His current research interests include system-level design and validation of system-on-chip platforms. In this context, he collaborated in several US, Japanese and Italian projects.

Luca P. Carloni received the Laurea degree (*Summa Cum Laude*) in electrical engineering from the Università di Bologna, Bologna, Italy, in 1995, and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, USA, in 1997 and 2004, respectively. He is currently an Associate Professor with the Department of Computer Science, Columbia University, New York, NY, USA. He has authored over 130 publications and holds two patents. His current research interests include system-on-chip platforms, distributed embedded systems, and high-performance computer systems. Dr. Carloni was a recipient of the Demetri Angelakos Memorial Achievement Award in 2002, the Faculty Early Career Development (CAREER) Award from the National Science Foundation in 2006, the ONR Young Investigator Award in 2010, and the IEEE CEDA Early Career Award in 2012. He was selected as an Alfred P. Sloan Research fellow in 2008. His 1999 paper on the latency-insensitive design methodology was selected for the Best of ICCAD, a collection of the best papers published in the first 20 years of the IEEE International Conference on Computer-Aided Design. In 2013 Dr. Carloni served as general chair of Embedded Systems Week (ESWEEK), the premier event covering all aspects of embedded systems and software. He is a Senior Member of the Association for Computing Machinery.