System-Level Memory Optimization for High-Level Synthesis of Component-Based SoCs

Christian Pilato, Paolo Mantovani, Giuseppe Di Guglielmo and Luca P. Carloni

Department of Computer Science, Columbia University, New York, NY, USA
{pilato,paolo,giuseppe,luca}@cs.columbia.edu

ABSTRACT

The design of specialized accelerators is essential to the success of many modern Systems-on-Chip. Electronic system-level design methodologies and high-level synthesis tools are critical for the efficient design and optimization of an accelerator. Still, these methodologies and tools offer only limited support for the optimization of the memory structures, which are often responsible for most of the area occupied by an accelerator. To address these limitations, we present a novel methodology to automatically derive the memory subsystems of SoC accelerators. Our approach enables compositional design-space exploration and promotes design reuse of the accelerator specifications. We illustrate its effectiveness by presenting experimental results on the design of two accelerators for a high-performance embedded application.

Categories and Subject Descriptors
B.5 [**RTL Implementation**]: Design Aids

General Terms
Algorithms, Design, Experimentation

Keywords
High-Level Synthesis, Memory Optimization, System-on-Chip.

1. INTRODUCTION

System-on-chip (SoC) architectures are becoming increasingly heterogeneous as they combine multiple processor cores with a variety of accelerators. These are specialized hardware components that are dedicated to the execution of selected computational kernels. Since power dissipation is the biggest concern in SoC design and specialized hardware can offer 2 to 3 orders-of-magnitude higher efficiency than a corresponding software implementation [15], the number of accelerators in a given SoC is expected to continue to grow [25].

While beneficial in terms of energy-efficient performance, the presence of accelerators exacerbates the complexity of SoC design. To counterbalance this effect, designers will increasingly rely on Electronic System Level (ESL) methodologies that promote: the specification and validation of the design at level of abstraction higher than RTL [4], the use of high-level synthesis (HLS) tools [11, 14, 20], and the reuse of pre-designed and pre-validated components, also known as intellectual property (IP) blocks [23].

Recent works have proposed compositional design methodologies to derive optimal implementations of specialized hardware by combining components that are expressed in a high-level language, such as a set of SystemC processes, and synthesized with HLS tools [17, 19]. From a single high-level specification, HLS tools can be used to obtain a set of alternative Pareto-optimal implementations that offers many different choices in terms of performance versus cost (area, power) tradeoffs [19]. This process is called *accelerator characterization* and is the key to design reuse because it augments the applicability of a given accelerator design: the architect of a given SoC may prefer a faster implementation of the given accelerator while the architect of another SoC may choose a slower but smaller one. State-of-the-art HLS tools offer a rich set of *knobs* for *intra-process* optimization, e.g. for loop manipulation, state insertion, array implementation, and function inlining. On the other hand, they have limited capabilities for *inter-process* optimization. In particular, they lack proper support for the synthesis of communication channels [13] and the optimization of the memory hierarchy [9]. The impact of these limitations is destined to grow with the complexity and size of SoC accelerators. For instance, a complex accelerator for image processing or media applications typically consists of multiple processes that interact by producing and consuming large data structures such as image frames [13].

As storage elements may occupy more than 70% of a chip [16], it is critical to optimize their area. While the use of distributed registers is convenient to store small and frequently accessed data, large arrays and other complex data elements require the allocation of pre-defined memory IPs. These can be Static Random-Access Memories (SRAMs), in the case of SoC design based on standard cells, or Block Random-Access Memories (BRAMs), in the case of FPGA implementations. These memory IPs allow fast data accesses with reduced memory footprint, but their complexity and area grow quadratically with the number of their ports [24]. Hence, most target technologies usually offer only single- or dual-port memories, which can be instanced in different sizes using memory generators. The reduced number of ports, however, constrains the potential parallelism of memory operations, thereby limiting the overall accelerator performance. In particular, HLS tools allow the binding of a data structure, such an array, to a generic memory instance but introduce tight constraints on the use of these memories: e.g., for most tools the total number of process

CODES+ISSS'14, October 12-17 2014, New Delhi, India.
Copyright 2014 ACM 978-1-4503-3051-0/14/10...$15.00.
http://dx.doi.org/10.1145/2593069.2593071

interfaces that can access a data structure cannot exceed the number of ports of the corresponding memory instance.

To increase the number of parallel accesses that can be performed by the accelerator logic, designers combine many small SRAMs or BRAMs in multi-bank architectures, instead of using a single large memory. This, however, is a manual design effort that is not supported well by HLS tools. Techniques like *data duplication* [5] or *data distribution* [5, 7, 26] have been proposed for optimizing the memory accesses of a single process, but they remain to be extended to the case of accelerators composed of multiple processes. Furthermore, they are usually applied *before* the invocation of HLS tools [26, 27] as they require the rewriting of the synthesizable code in order to specify explicitly the accesses to the different banks. This limits design reuse because every time the memory sub-system is modified the accelerator characterization and validation must be repeated by rerunning the HLS tools (since the modified specification of its processes may result in different RTL micro-architectures).

To address these limitations we present a novel methodology for optimizing the memory subsystems of SoC accelerators. The goal is to bridge the gap between the data structures that are defined as part of the system-level specification of the accelerator processes and the physical memory IPs that are available for the synthesis with a given technology. Additionally, the methodology promotes design reuse because it does not require designers to make any modifications to the specification of the accelerator processes in terms of how they access their data structures. Indeed, our approach enables the pre-characterization and validation of each process through the automatic synthesis of multiple Pareto-optimal implementations with various HLS knobs configurations. Some of these implementations may be based on the assumption that a particular data structure is mapped on a memory instance which may have a number of ports higher than what is available for the actual memory IPs of the target technology. This, however, is not a problem for our methodology which is capable of building multi-bank architectures that can satisfy multi-port requirements through the automatic allocation of multiple memory IPs. These physical instances are encapsulated within a controller that can coordinate multiple processes in their accessing of the data stored in the various banks.

Specific contributions of this work include:

1. a flexible memory controller that can be configured to orchestrate the data accesses of all the processes insisting on the same underlying physical memories;

2. an algorithm to automatically determine an optimal architecture of the memory subsystem given a formal description of the data exchanges among the accelerator processes. If the description includes multiple accelerators that operate in time-multiplexing, our algorithm is capable of discovering opportunities to enforce the sharing of physical memories among the accelerator processes to reduce the overall memory footprint;

3. a prototype CAD tool that embodies the proposed methodology by combining the algorithm with the flexible memory controller.

To demonstrate the proposed methodology we applied it to the design of two accelerators for Wide Area Motion Imagery, a high-performance embedded application.

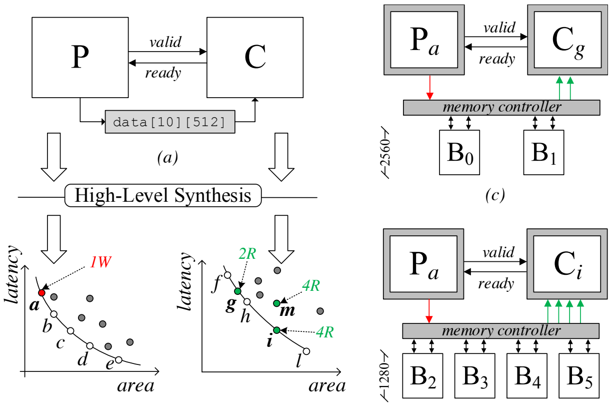
Listing 1: Synthesizable SystemC code of two simple processes P and C.

```
 1  #include <systemc.h>
 2  SC_MODULE(accelerator) {
 3    sc_in<bool> clk, rst;
 4    //...
 5    SC_CTOR(accelerator) {
 6      SC_CTHREAD(P, clk.pos());i
 7      reset_signal_is(rst, false);
 8      SC_CTHREAD(C, clk.pos());
 9      reset_signal_is(rst, false);
10      //...
11    }
12    void P(void) {
13      // reset ...
14      bool pp_flag = false;   // ping-pong flag
15      unsigned pp_offset = 0; // ping-pong offset
16      wait();
17      while(true) {
18         // input ...
19         // computation ...
20  LOOP1: for (int i=pp_offset; i<(pp_offset+5); i++)
21  LOOP2:   for (int j=0; j<512; j++)
22             { data[i][j] = f(...); }   //write to data
23         // output ...
24         // (wait for ready from C then notify as valid)
25         pp_flag != pp_flag;
26         pp_offset = pp_flag?5:0;
27       }
28    }
29    void C(void) {
30      // reset ...
31      bool pp_flag = false;   // ping-pong flag
32      unsigned pp_offset = 0; // ping-pong offset
33      wait();
34      while(true) {
35         // input ...
36         // (wait for valid from P then notify as ready)
37         // computation ...      //process one row
38  LOOP3: for (int i=pp_offset; i<(pp_offset+5); i++)
39  LOOP4:   for (int j=0; j<512; j++)
40             { g(data[i][j], ...); }  //read from data
41         // output ... (valid to P)
42         pp_flag != pp_flag;
43         pp_offset = pp_flag?5:0;
44      }
45    }
46    private:
47    sc_signal<bool> valid, ready;
48    int data[10][512];  // ping-pong buffer
49    int offset;
50  };
```

2. MOTIVATING EXAMPLE

Listing 1 reports a portion of synthesizable SystemC code for a simple accelerator, which is composed of two processes P and C. The same accelerator is graphically represented in Fig. 1(a). Process P produces a chunk of data (i.e. 5×512 integers that can represent 5 rows and 512 columns of an image) that is stored in array `data` (lines 20-22) and then consumed by process C (lines 38-40). The two processes synchronize their execution through explicit signals (`valid`, `ready`) such that C cannot start its computation before P has produced the required amount of data. The processes work in pipeline. For this reason array `data` (line 48) is implemented as a ping-pong buffer of size 10×512, which is two times the size of the produced/consumed chunk of data.

HLS tools can apply multiple "knobs" to generate alternative hardware implementations for each process in order to trade off performance metrics (e.g. latency) and area/power costs. The resulting implementations form a Pareto-optimal set of choices that the designer can compose to create the final system [17, 19], as shown in Fig. 1(b).



(a)

(b)

(c)

(d)

Fig. 1: A motivating example.

Example. Let us consider the implementations of process C. Implementation g is obtained by unrolling `LOOP4` for two iterations, which requires two concurrent memory-read operations. It also adopts resource sharing to limit the area. Implementation m is obtained by unrolling `LOOP4` for four iterations to maximize the performance at the cost of more area, but allowing no more than two concurrent memory-read operations. As a result, this imposes a bottleneck in the computation, because the four memory operations cannot be all scheduled in the same clock cycle. Finally, implementation i, which Pareto-dominates implementation m, is obtained by unrolling `LOOP4` for four iterations and allowing four concurrent memory-read operations. □

This design space exploration imposes additional challenges to the designer for the generation of the underlying memory subsystem since the technology libraries for HLS may only offer memories with a limited number of ports. For example, in case of ASIC, memory compilers (e.g. Embedded Memory IP [3]) can be adopted to create at most efficient dual-port SRAMs. Similarly, modern FPGAs feature configurable memories, called BRAMs [28], which still have at most two read/write ports. For this reason, the effective selection of implementation i requires to build a multi-bank memory architecture that increases the number of available physical ports and thus the bandwidth for the processes. The generation of such memory subsystems, however, is a laborious task, especially when multiple processes communicate through a shared memory. Indeed, in this case, the designer needs to combine the requirements of the different processes in terms of memory ports to determine the proper memory organization.

Example. A designer who has to select an implementation for each process must consider that each implementation may have different requirements in terms of memory operations. For example, implementations a of P and g of C require one memory-write operation and two concurrent memory-read operations, respectively. Since the two processes execute in pipeline and thus they need to independently access the memory, we need to implement the memory subsystem with two parallel banks and to partition the data between the two banks to allow the parallel accesses. To accomplish this, we apply a cyclic partitioning to the original data [26]: this affects both read and write operations because P has to write alternatively into the two banks. The resulting system is shown in Fig. 1(c), where banks B_0 and B_1 consist of 2560 words each. On the other hand, if the designer selects implementation i that requires four parallel interfaces, we need to include four banks of 1280 words, as shown in Fig. 1(d). □

It is worth noting that in both cases, process P is realized with the same implementation P_a, but, given the combined requirements of interfaces and the resulting memory subsystems, its memory operations have to be changed to write the values in the correct bank. Specifically, it is necessary to correctly remap the *logical addresses* provided by P to the *physical addresses* required to access the proper physical banks. For example, let's consider four write operations from process P (from logical addresses 0 to 3). In the former case, these need to be translated into four consecutive write operations to the physical addresses $\langle B_0,\texttt{0x000}\rangle$, $\langle B_1,\texttt{0x000}\rangle$, $\langle B_0,\texttt{0x001}\rangle$ and $\langle B_1,\texttt{0x001}\rangle$. In the latter case, the same logical addresses need to be translated into write operations to the physical addresses $\langle B_2,\texttt{0x000}\rangle$, $\langle B_3,\texttt{0x000}\rangle$, $\langle B_4,\texttt{0x000}\rangle$ and $\langle B_5,\texttt{0x000}\rangle$.

To effectively enable a compositional synthesis of various processes and the optimization of the resulting memory subsystem, we developed a flexible memory controller that can be instanced and configure to absorb these differences. This controller allows the reuse of the same physical banks to store different data structures by correctly managing the memory operations requested by each process. This possibility of reuse can be adopted to reduce the memory footprint of an accelerator. Consequently we developed a design methodology to allocate the data structures to the storage resources and determine the proper organization of the memory subsystem.

3. FLEXIBLE MEMORY CONTROLLER

Composing pre-characterized processes may require to generate different memory subsystems with multiple banks to provide enough bandwidth for read and write operations. As described in Section 2, processes perform the memory requests through *logical addresses* as they have no information about the organization of the memory subsystem. Then, the memory controller is in charge of translating these addresses into the corresponding *physical addresses* to access the banks where the data is effectively stored. This is performed in a transparent way with respect to the synthesis and execution of the processes and no modifications are required for their synthesis, which improves the reusability of the components.

Fig. 2 shows the organization of the memory subsystem for the example shown in Fig. 1(d). The memory controller features one write interface for process P and four read interfaces for process C. The memory subsystem contains four 1280x32 banks such that process C can perform four read operations in parallel while process P writes the subsequent chunk of data (i.e. ping-pong buffer to support pipelined execution). Both process and memory interfaces are designed to feature the following signals:

- *Chip Enable (CE)*: it represents when there is an active request on the corresponding interface. The request is considered active (and the corresponding address is valid) only when the corresponding CE is active.

- *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, since they have no information about the memory organization, the bitwidth of A corresponds to the size of the data to be accessed. In case of memories, the bitwidth of A corresponds to the actual size of the memory IP.

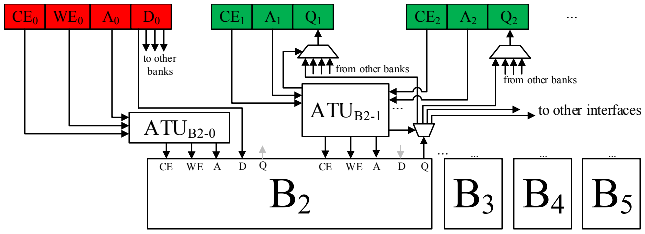


Fig. 2: The RTL architecture of the flexible memory controller.

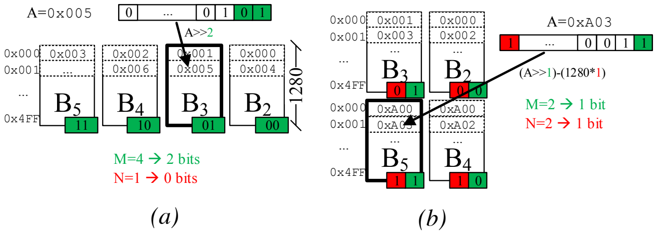
- *Input Data (D)*: it represents the data to be written into the memory and thus it is present only in write interfaces.

- *Output Data (Q)*: it represents the data read from the memory and thus it is present only in read interfaces.

- *Write Enable (WE)*: if active, it specifies that the request is effectively a write operation and the corresponding input data D is valid.

Each time the memory controller receives a request from an interface (i.e. the corresponding CE is active), it must analyze the provided address A and determine: (1) the bank to be effectively accessed (enabling the corresponding CE) and (2) the physical address inside this bank (providing the correct translation). This functionality is implemented by an *Address Translation Unit* (ATU) that is instantiated for each memory port. Each ATU collects the requests of all interfaces insisting on the corresponding memory port to determine when the bank is selected and provide the correct translation from *logical* to *physical* address. It is necessary to guarantee that the read operations from distinct interfaces never access the same bank at the same clock cycle. Indeed, in each bank, one port is reserved for write operations and, thus, only one read operation can be performed at each clock cycle. Supporting multiple requests on the same bank at the same clock cycle would require a proper circuitry to serialize the requests in a transparent way with respect to the computation of the process (as in [22]). However, this requires also to modify the specification of the process which must be *stallable* when performing a memory request because it has to be insensitive to the latency of memory operations [8]. This has been thus left as a future work. In this work, instead, we support a cyclic partitioning of the data to ensure independent accesses. This is especially valid for single accesses to arrays contained in loops that have been unrolled for a certain number of iterations, as the one in Listing 1. More complex patterns, like the ones in stencil codes, can be supported with more advanced algorithms for data partitioning, as the one presented in [26]. From our viewpoint, this only affects how the ATU translates the logical addresses into the corresponding physical addresses.

Within an instance of a memory controller we have 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 [1]; and it simplifies the logic required to translate the logical addresses into the physical ones. In fact, in order to reduce the memory footprint, the same memory banks can be also rearranged and reused to store a different



(a) *(b)*

Fig. 3: Logical-to-physical address translation.

data structure, as shown in Fig. 3. For instance, let's consider two processes that require only two parallel banks to satisfy their combined requirements in terms of ports and that are never executed at the same time with respect to P and C. Therefore, the two remaining banks can be used to virtually increase the capacity of each parallel bank up to 2560 elements through block partitioning [5, 26].

As a result, given this combined block-cyclic partitioning of the data, a generic logical address can be decomposed as shown in Fig. 3. In this example, we assumed that the number of parallel blocks is a power of two. A different number of parallel banks only affects the functions and the logic to identify the parallel block and its internal addresses. Specifically, let's consider a memory subsystem that is configured to store a data buffer of size B by using $M \times N$ banks of size S, where M represents the number of parallel banks and each of them is replicated N times to increase its capacity, as in Fig. 3(b). The logical address to access the buffer will need $\lceil log_2(B) \rceil$ bits to address the entire address space. The address is decomposed as follow:

$$\lceil log_2(B) \rceil = log_2(N) + \lceil log_2(S) \rceil + log_2(M) \qquad (1)$$

Due to the cyclic data partitioning, $log_2(M)$ less significant bits are used to determine which is the parallel bank to be accessed. Then, due to the block data partitioning, the $log_2(N)$ most significant bits are used to identify which of the N repetitions used to increase the bank capacity has to be effectively accessed. Finally, the remaining $\lceil log_2(S) \rceil$ bits are used to address the data inside the actual physical bank that has been identified. As a result, given a data layout, it is possible to associate a *tag* composed of $log_2(N)+log_2(M)$ bits with each bank. The tags resulting from two different organizations of the same memory banks are shown in Fig. 3.

Example. Let's assume that array `data` (5120 elements) is stored into four parallel banks (each of them has size of 1280) and process C requires to read the logical address $A = 5$ (13 bit). The 2 less significant bits (01) are adopted to identify bank B_3, where the data is effectively stored (see Fig. 3(a)). Then, the remaining 11 bits are used to read the data from the second location of B_3. Conversely, let's assume that the same data structure only requires two parallel banks (as in the solution in Fig. 1(c)), but the designer aims at reusing the same banks with size of 1280. They will be thus reorganized as shown in Fig 3(b). If C requires to read the logical address $A = 2563$, the combination of the most and less significant bits are used to identify bank B_5 where the data can be effectively accessed. □

Since each memory port only performs one operation per clock cycle, the CE signal of the corresponding interface, combined with bank tags, can be used also to correctly drive the data. As shown in Fig. 2, a multiplexer is used to drive the data to the proper bank in case of write operations. Similarly, a demultiplexer is used to determine which interface

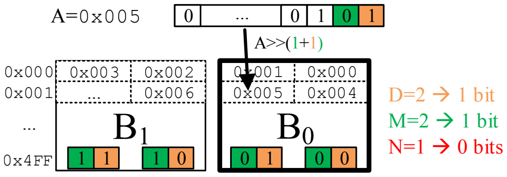


Fig. 4: Decoding of logical addresses in case of data merging.

must receive the value read from a bank. In case of read operations, the data will be provided after as many clock cycles as the latency of the read operation. For this reason, it is necessary to buffer the tag to ensure the correct identification of the bank that is serving the data.

Extension for Data Merging. The architecture of the proposed memory controller can be extended to support the merging of multiple write operations into a single one to reduce the number of ports. For example, let's consider a process that needs to write at the same clock cycle two different 16-bit values, always at consecutive addresses, while the rest of the data structures are 32-bit buffers. In this situation, it is possible to use a unique 32-bit memory bank, where the two 16-bit values are merged and written in parallel using only one physical port. Similarly to the case where multiple banks are instantiated in parallel, during a read operation the less significant bits will be used to determine which is the actual 16-bit data that needs to be accessed, as shown in Fig. 4. In particular, when D values are merged in parallel, the proper bits are adopted for selecting the memory location to be read, as explained above. The $log_2(D)$ less significant bits, combined with simple slice operations, are then used to address and extract the merged value to be effectively provided to the proper process interface.

In this section, we described the architecture of a flexible memory controller that can be configured to bridge the gap between the combined requirements of the processes and the available memory IPs. We also showed how the same physical banks can be efficiently used for multiple data structures.

4. SOC MEMORY ALLOCATION

Based on the memory controller described in Section 3, we developed a methodology to optimize the memory organization of the entire accelerator. In particular, after the designer has selected an implementation for each process, it is possible to determine the number of parallel memory interfaces that each data structure requires to ensure a sufficient amount of memory bandwidth to the processes. Then, it is possible to determine the requirements in terms of physical banks to meet the performance constraints of the memory accesses and how these banks can be reused across different data structures to minimize the memory footprint of the entire accelerator.

Specifically, given an accelerator to be implemented, it is necessary to perform the following steps to determine and optimize the corresponding memory subsystem:

1. **Identification of parallel banks.** The minimum number of memory banks is identified to satisfy the combined requirements of the processes in terms of bandwidth (i.e., number of parallel accesses). This requires to consider information about data access patterns to determine the technique for data mapping (distribution or duplication [5]).

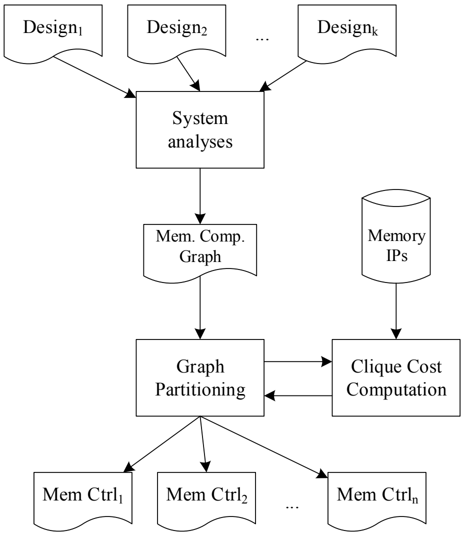


Fig. 5: Overview of the methodology for memory allocation and optimization at system level.

2. **Identification of physical banks.** The size of the actual banks is based on the size of the data structure, the amount of data that can be stored by each bank, and the area characterization stored in the library in order to minimize the memory footprint.
3. **Memory merging for reuse.** The same physical banks can be reused to store multiple data structures when they are not active at the same time. This aims at reducing the memory footprint of the entire system.
4. **Definition of data mapping.** Once the number of actual physical banks has been determined, the data mapping is performed (block and/or cyclic partitioning). The ATU functions (i.e. logical-to-physical address translation) are derived, along with the logic to control the banks.

These steps are highly interdependent. For example, the number of actual physical banks affects the possibilities for memory reuse, while sharing the same memory elements for different data structures may impose a different organization of the banks, as discussed in Section 2.

Fig. 5 shows our proposed methodology for system-level memory allocation and optimization. First, we analyze the system descriptions of the accelerators to be implemented as the methodology can easily support and optimize the memory subsystem for multiple accelerators to be executed in mutual exclusion in the SoC. This analysis aims at determining which data structures are compatible (i.e. their lifetimes are non-overlapping) and, therefore, can share the same storage resources. The result of this analysis is a *memory compatibility graph* [12].

Definition 1. A memory compatibility graph is a graph $G = (V, E)$ *where each node* $v \in V$ *represents a data structure to be stored in memory, while an edge* $e \in E$ *connects two nodes if and only if the corresponding data structures can share the same storage resources.*

Then, this graph is analyzed to determine the organization of the memory subsystem. Before defining the problem, let us recall the definition of *clique*.

Definition 2. 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 this context, each clique represents a set of compatible data structures that can share the same storage resources. For this reason, each clique will correspond to an instance of the memory controller described in Section 3.

Given a clique C_i, each data structure $v \in C_i$ is characterized by a combined requirement of read and write interfaces. This determines the minimum number of parallel banks for each data structure v that can satisfy these requirements. This information is used to compute the organization of the entire memory subsystem to minimize the cost of the clique, which corresponds to its memory footprint A_i. Finally, we can formulate system-level memory allocation as a *graph partitioning problem.*

Definition 3. The optimal memory allocation consists in finding a partition of V into disjoint cliques $C = (C_1, \ldots, C_n)$ of minimum cost. Here, the cost A of a partition is the sum of the costs of the cliques in the partition defined as follows:

$$A = \sum_{i=1}^{n} A_i \tag{2}$$

where n is the number of cliques and A corresponds to the footprint of the entire memory subsystem to be minimized.

Note that n corresponds also to the number of memory controllers that need to be generated (see Section 3).

The different steps to create and solve this problem are detailed in the following subsections.

4.1 Creation of Memory Compatibility Graph

The creation of the compatibility graph is crucial to determine the possibilities of reusing the storage resources. Hence, in most cases the designer has to provide insights on the application's behavior. Indeed, increasing the number of edges into the graph G corresponds to increasing the number of compatible data structures. This can potentially induce the creation of larger cliques at lower cost as the same banks can be reused across different data structures. Before describing the creation of the graph, we introduce the definition of *local*, *input* and *output* data structures.

Definition 4. Let P be a process to be implemented in the final system. A local data structure is a data structure that is read and written only by process P. An input (output) data structure is a data structure that is read (written) by P and shared with other processes.

We start building the graph by adding all nodes V, i.e., the data structures of the different accelerators to be implemented that need to be stored in memory, and no edges. We adopt a conservative approach: solving the formulation with no edges corresponds to implementing each data structure with its own memory subsystem. Then, any additional analysis on the processes or the system topologies can only increase the number of compatibility edges and, in turn, the possibilities of sharing.

It is possible to perform different analyses on the system topologies to determine which processes are never executed at the same time. For example, let's consider the two designs shown in Fig. 1(a) and assume that they are connected to the rest of the system, which repeatedly provides data to P and receives data from C. Processes P and C cannot use the same physical banks for their local data structures because P starts a new iteration and it is processing a new batch of data during the execution of C. Conversely, if additional signals are introduced such that P is forced to wait until the termination of C, P cannot start a new iteration. In this case, the *local* data structures of P and C are compatible and they can share the same memory banks.

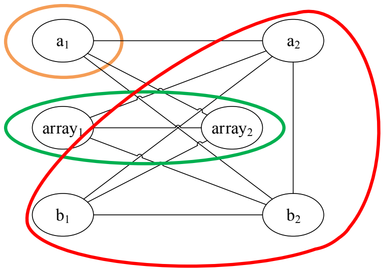


Fig. 6: Example of memory compatibility graph. Each clique represents which data structures can share the same memory banks.

Currently, we assume that all *local* data structures of the same process are alive from the activation of a process until its termination. So they cannot share the same storage resources. Moreover, we assume that all *input* data structures of a process are adopted to create all *output* data structures of the same process, and they cannot share the same resources as well. On the other hand, analyses of the code to be synthesized can 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 and, in this case, they can share the same storage resources because these are always accessed in mutual exclusion.

Finally, in case of multiple accelerators, since we assume that they are never executed simultaneously, all data structures belonging to different accelerators are compatible. This allows the minimization of the memory footprint across multiple accelerators.

Note that in some cases, it is also possible to increase the compatibility of some data structures by forcing the serialization of some parallel processes through explicit signals. This improves the possibilities for sharing, which can potentially reduce the memory footprint, but it requires to restructure the design of the accelerator with possible impact on the performance. The study of this transformation has been left for future work.

4.2 Computation of Clique Cost

As described above, a clique C_i of the nodes V corresponds to a memory subsystem that can be generated to reuse the same storage resources across multiple data structures. To efficiently implement the memory subsystem corresponding to a clique, we developed the algorithm shown in Listing 1, which determines both the number of banks and their size.

Algorithm Description. Given a clique C_i of data structures, the algorithm determines both the organization of the memory subsystem S_i (`DetermineBanks`) and the associated cost A_i (lines 3-5) for its implementation. First, for all data structures, we compute the number of interfaces

Algorithm 1: Algorithm to determine the memory subsystem associated with each clique and its cost.

```
1  Procedure CostClique($C_i$)
       Data: $C_i$ is the clique to be implemented
       Result: $S_i$ is the memory subsystem associated with $C_i$
       Result: $M_i$ is the cost of implementing $S_i$
2      $S_i \leftarrow$ DetermineBanks($C_i$)
3      $A_i \leftarrow 0$
4      foreach $bank \in$ GetListOfBanks($S_i$) do
5          $A_i \leftarrow A_i +$ GetArea($bank$)
6      return $\langle S_i, A_i \rangle$
7  Procedure DetermineBanks($C_i$)
       Data: $C_i$ is the clique to be implemented
       Result: $S_i$ is the memory subsystem associated with $C_i$
8      $R_i \leftarrow$ DetermineInterfaceReq($C_i$)
9      $P_i \leftarrow$ ComputeMinimumBanks($C_i$,$R_i$)
10     $v \leftarrow$ GetFirst(OrderByNumBank(Nodes($C_i$),$P_i$))
11     $B \leftarrow$ GetBanks($v$)   // $B$ is the current number of banks
12     $Size \leftarrow 0$  // $Size$ is the current capacity of each bank
13     if IsPartitioned($v$) then
14         $Size \leftarrow$ GetDataSize($v$) / $B$
15     else
16         $Size \leftarrow$ GetDataSize($v$)
17     foreach $v \in$ OrderByNumBank(Nodes($C_i$),$P_i$) do
18         $N \leftarrow$ Floor($B$/ GetBanks($v$))
19         if IsPartitioned($v$) then
20             if GetDataSize($v$) / GetBanks($v$) $> Size * N$ then
21                 $Size \leftarrow$ GetDataSize($v$) / (GetBanks($v$) $*N$)
22         else
23             if GetDataSize($v$) $> Size * N$ then
24                 $Size \leftarrow$ GetDataSize($v$) $/N$
25     $\langle B, Size \rangle \leftarrow$ SplitBanks($B$, $Size$)
26     $S_i \leftarrow$ GenerateMemoryController($C_i$, $B$, $Size$)
27     return $S_i$
```

required by the process implementations as well as the combined requirements in terms of concurrent read and write operations (`DetermineInterfaceReq`, line 8). Based on this information and the data access pattern, it is possible to determine the minimum number of parallel banks that is needed to satisfy the bandwidth requirements of each data structure (`ComputeMinimumBanks`, line 9), which corresponds to value M in Fig. 3. Then, we sort the data structures in a descending order (`OrderByNumbBank`), from the one that requires the maximum number of parallel banks to the one with the minimum number. This allows the definition of the maximum number of parallel blocks and, thus, the minimum number of banks that are required to provide this bandwidth (line 10-12). The goal is then to enforce the reuse of these banks as much as possible from parallel to serial as described in Section 2. We also determine an initial size for these banks based on the data mapping to be implemented (lines 13-16). Then, we then analyze all data structures following the same descending order and we aim at reorganizing the existing banks from parallel to serial (as shown in Fig. 3) to determine if the data structure can fit into this new configuration (line 18). If not, the size of the banks is updated accordingly to the data mapping solution.

Example. Let's assume that the current number of banks is four (each of them having size of 128) and we need to store a data structure which has size of 900, partitioned in three parallel banks. The existing four banks cannot be rearranged in multiple lines as done in Fig. 3 and for this reason $N = 1$. However, this organization is not sufficient to store then entire data structure (line 20) and for this reason, each of the four banks is now enlarged to store 300 elements (line 21). □

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, and we need to store a data structure of 512 elements, which requires two parallel banks ($M = 2$) but with data duplication because data partitioning cannot be adopted. In this case, the four banks can be rearranged as in the right-hand side of Fig. 3 ($N = 2$); the two serial banks provides a virtual capacity of 600 for each parallel block. It is thus possible to store the entire amount of data with a serial reorganization of the banks and, thus, no changes are applied to the size of the banks (line 23). □

Next, the banks are rearranged if the current size is greater than the largest one available into the library, by increasing the number of banks (`SplitBanks`, line 25). Finally, the actual instance of the memory controller is generated, including the logic for logical-to-physical address translation (line 26), as described in Section 3.

4.3 Memory Allocation and Optimization

To obtain an efficient system-level allocation of the memory elements, it is necessary to partition the memory compatibility graph in cliques such that the total cost is minimized. This problem is generally NP-hard because it contains the classic NP-hard problems *partition into cliques* (or *clique cover*) and *graph coloring* [10].

Given the organization of the memory controller described in Section 3, the most critical elements in terms of timing effects are the multiplexers and the demultiplexers for the correct routing of the data. Even if the processes are synthesized with a certain margin for memory operations, this logic grows with the number of the banks and the number of the interfaces and this may affect the critical path of the design. For this reason, the designer may impose a limit to the amount of memory sharing. This corresponds to imposing a maximum value B to the cardinality of the cliques, which needs to be empirically determined by the designer.

The resulting formulation is the *partition into cliques of bounded size* $PCliq(G, f, B)$ problem (where f is the function to determine the clique cost as described in Section 4.2). This is a well-known problem in combinatorial optimization that can be approached with the algorithm presented in [10]. The resulting cliques correspond to the memory subsystems to be generated and a memory controller is generated for each of them.

5. EXPERIMENTAL EVALUATION

We developed a prototype CAD tool in C++ based on the proposed methodology. This tool has been applied to two accelerators, which we designed in SystemC starting from the kernel implementations included in THE PERFECT BENCHMARK SUITE [6]. The benchmark chosen are *Debayer* and *Change Detection* from Wide Area Motion Imagery. Their system-level specifications are depicted in Fig. 7(a) and Fig. 8, respectively. Both benchmarks elaborate images of size 512x512 pixels. *Debayer* consists of three processes, which access two data structures a and b of size 12288∗`sizeof`($u16$) and $2 * 12264 * (3*$`sizeof`$(u16))$, respectively. We developed the synthesizable SystemC code in order to maximize the performance: we allowed pipelining of the processes by accessing a and b as circular buffers and we kept a limit to the total size of local variables to be allocated on memories. As shown in Fig. 7, process $D1$ needs to

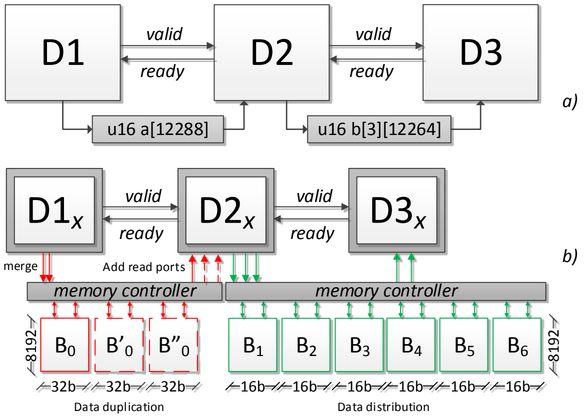


Fig. 7: *Debayer* system-level view (a) and with memory banks mapping (b).

perform two write operations on array a, while $D2$ could be concurrently reading one element. During the process specification we did not consider other feasibility constraints, such as the available memory sizes or maximum number of ports, because we rely on our tool to perform the appropriate mapping of the data structures. *Change Detection* is also composed of three processes, but it includes one array f of size $2 * 512 * \texttt{sizeof}(u16)$, one array g of size $2 * 512 * 1$ and six arrays u_in, s_in, w_in, u_out, s_out, w_out, of size $2 * 2560 * \texttt{sizeof}(u32)$. We allowed the pipelined execution of the processes, this time through a *ping-pong* mechanism that alternatively loads a portion of the data while it computes the results on the other portion. On the other hand, we also designed a version of this benchmark where we prevent the pipelined execution of the processes through explicit signals between $C3$ and $C1$. This solution was developed to evaluate the capabilities of our tool with respect to sharing the physical banks across the processes.

In our experiments, we used a commercial HLS tool to generate the different implementations of each process, aiming at targeting a frequency of 1GHz for an industrial 32nm CMOS and a frequency of 100MHz for a Xilinx Virtex-7 FPGA. We then adopted the available IPs provided by the two technology libraries to implement the memory banks. Specifically, we limited our analysis to SRAMs and BRAMs with two independent read-write ports, each capable of serving one read or one write request per clock cycle and with both synchronous write and synchronous read interfaces. Asynchronous RAMs were ignored because they are only available for FPGA and usually lead to very high (and inefficient) device utilization. On the other hand, *double-pumped* two-ports SRAMs are available for ASIC only. Two-port memories can accept one read and one write on the same cell in one clock cycle, but at the cost of a big increase in area occupation. Single-port SRAMs, instead, are very compact, but they limit the access to each bank to one single process at a time, which can potentially lead to wrong implementations when different processes have to read and write the same memory bank at the same time. For the 32nm CMOS technology, we adopted a memory generator to create SRAMs of alternative sizes, as those produced by the allocation algorithm proposed in Section 4. In the FPGA case, we used the available BRAMs on the target device.

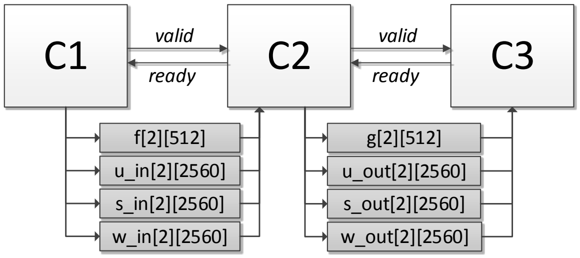


Fig. 8: *Change Detection* system-level view.

We designed a set of experiments to show that our tool can efficiently allocate memory banks based on the constraints derived from both the design and the configuration of HLS knobs, without the need to change the SystemC code.

For the *Debayer*, array a can be safely mapped to a bank with half of the words but a doubled bit-width thanks to the regular write pattern of $D1$. We indeed exploit the merge of data structures as described in Section 3. Thus the two logical write operations are performed by a single physical one and array a is mapped to the bank B_0. Banks B'_0 and B''_0 are not considered in this phase. The access to array b requires three write and two read operations. This time, we performed data distribution and split the bi-dimensional array into three uni-dimensional arrays. This allows an efficient mapping of the data structure to six banks (from B_1 to B_6) that satisfies processes' access requirements.

Next, we performed a simple design space exploration that takes advantage of our tool. Debayer's process $D2$ contains several loops which interleave computation to accesses to local data a. The latency of the accelerator is therefore severely affected by the number of possible memory accesses per clock cycle. Without changing the initial SystemC code we instrumented our tool to map array a with two and three read ports. At the same time, we set the HLS knobs to allow $D2$ to read respectively two and three elements in parallel. In this case, however, data duplication is the only viable solution to allow multiple read accesses in parallel. Data distribution is not applicable for this particular algorithm because it is impossible to guarantee that two concurrent read operations never insist on the same physical bank. At this point, we used our tool to analyze these constraints and generate the proper memory controllers for each of the target technologies. The resulting memory subsystems have been integrated with the rest of the system (i.e. RTL descriptions of the synthesized processes); logic synthesis and RTL simulation have been performed to obtain the area occupation of the memories and the performance of the systems, respectively. As reported in Table 1, when 2 parallel read operations can occur on array a ($2R$), the accelerator's total latency is reduced by approximately 40% on FPGA and by 45% on ASIC. Adding an additional read port ($3R$) brings latency down by 55% on FPGA and 60% on ASIC. The speedup introduces a cost in terms of memory resources that is at most 50% of the initial resources for both FPGA and ASIC, as reported in Table 1.

The two versions of *Change Detection* are identified as *pipe* and *share*, respectively. The former allows the pipelined execution of the processes through the *ping-pong* buffer, while the latter allows the sharing of the memory banks across the different processes since an iteration has to be completed

Table 1: Performance and memory occupation of the accelerators. In case of FPGA, the memory occupation corresponds to the number of BRAMs, while in case of ASIC, it corresponds to the memory footprint in μm^2.

	Debayer						*Change Detection*			
	FPGA Virtex7			**CMOS 32nm**			**FPGA Virtex7**		**CMOS 32nm**	
	1R	*2R*	*3R*	*1R*	*2R*	*3R*	*pipe*	*share*	*pipe*	*share*
# Cycles	13,826,802	8,275,396	6,210,884	11,888,292	6,727,012	4,662,500	8,279,465,640	9,093,107,980	7,618,917,820	8,432,508,900
Diff.	-	-40.15%	-55.08%	-	-43.41%	-60.78%	-	+9.83%	-	+10.68%
Memory	64	80	96	785,347	981,698	1,178,085	66	33	984,786	508,224
Diff.	-	+25.00%	+50.00%	-	+24.58%	+49.36%	-	-50.00%	-	-48.55%

before the subsequent one can start the computation. Similarly, we applied our tool to both system descriptions in order to determine the memory subsystems. The results in Table 1 show that preventing the pipelined execution of the processes degrades the performance by almost 10%, but allows a significant reduction in terms of memory resources (around 50%).

Finally, to verify the correctness of our designs, we integrated the two accelerators into a complete system that has been prototyped on a Xilinx Virtex-7 FPGA. It features a LEON3 processor [2] and an AMBA 2.0 bus as communication subsystem. The LEON3 processor provides the data to the accelerators, controls their execution, and retrieves the results when their computation is completed. All designs that have been evaluated in this section completed the execution without errors. This means that the memory controllers correctly perform the memory operations in all the cases.

The results show that the proposed approach can design efficient memory subsystems for component-based accelerators. In particular, it can deal with different requirements of the processes in terms of read and write interfaces and can update the memory subsystem accordingly. For doing this, it combines multi-bank memory architectures with a memory controller that manages the organization of the data in a transparent way with respect to the processes. As a result, the designer can focus on the development of the accelerators and easily evaluate the effects of applying HLS knobs (e.g. loop transformations) or design solutions (e.g. pipelined execution) in terms of performance and requirements of resources. He or she does not have to take care of the organization of the actual memory subsystem that is generated automatically by our tool.

6. RELATED WORK

The specialization of memory architecture has been widely studied in embedded systems due to its impact on area, power, and performance [21].

Panda *et al.* have proposed various solutions to create custom architectures and improve the system's performance, both in terms of memory organization and data layout [21]. Similarly, Benini *et al.* proposed a technique to customize the memory sub-system with respect to an application profiling while accounting for layout information to minimize power consumption [7]. The possibility of sharing and reusing memory elements have been explored by Desnos *et al.* [12]. They target MPSoCs and aim at minimizing the amount of memory to be allocated. As a consequence, multi-banks architectures and the constraints imposed by the limited number of the physical ports are not taken into account.

Customizing the storage structures in hardware accelerators has been explored by Baradaran and Diniz who propose *data duplication* and *data distribution* to improve performance while considering the capacity and bandwidth constraints of the storage resources [5]. They adopt a compiler-based approach to modify the source code. Similarly, Wang *et al.* [27] propose an automated methodology for data reuse, memory partitioning, and memory merging for FPGA. This approach has been also extended to consider the concurrent optimization of multiple processes [29], but only to optimize the fine-grained communication, not the storage elements. On the other hand, Wang *et al.* presented a complete theory for data partitioning which includes the support for more complex data access patterns [26]. This method can be added to our approach as a technique to determine how the logical addresses need to be translated into the corresponding physical addresses. All these solutions are applied to the behavioral specification, before high-level synthesis. This is efficient and elegant, but it imposes several limits to the reusability of the components as they need to be resynthesized each time the memory subsystem is modified.

Recently, different approaches have been proposed to effectively enable the reuse of pre-characterized components in SoCs. Liu *et al.* compose pre-characterized components to create a Pareto set of the entire system [19]. However, memory aspects are not taken into account. Li *et al.* adopt a similar approach to create systems by composing pre-characterized IPs through a pre-defined architectural template [17]. The approach has been further extended to determine the organization of the memory subsystem [18]. However, it is not clear how the authors can deal with memories having different latencies, especially considering the explicit synchronization of the entire system through a finite state machine. Moreover, it seems that parameters like the number of ports are not explored as this constraint is not considered when determining the memory elements. Conversely, our approach effectively enables a component-based design since it efficiently determines the organization of the memory subsystem. In addition, it also enables to share the same memory elements across multiple components, even if they belong to different designs that are deployed together onto the SoC.

7. CONCLUSIONS AND FUTURE WORK

We presented a solution to efficiently design the memory subsystems of one or more accelerators for heterogeneous SoCs. In particular, our approach accommodates the pre-characterization of all accelerator processes and handles the physical constraints (capacity and bandwidth) imposed by the physical memories available for a given target technology. To do so, we developed an algorithm that determines the proper organization of the memory subsystem in terms of a multi-bank architecture to satisfy these constraints while minimizing the memory footprint through the reuse of the banks. The algorithm leverages a configurable memory controller that is capable of dealing with the combined requirements of all the processes that need to access the same memory elements. Future work will involve the support for more complex data-access patterns and the identification of the proper system organization (i.e., trade-off between accelerator logic and memory elements) under strict area constraints.

Acknowledgments. This work is partially supported by C-FAR, one of the six centers of STARnet, the DARPA PERFECT program (Contract No: HR0011-13-C-0003), the NSF (A#:1219001), and by an ONR Young Investigator Award.

8. REFERENCES

[1] S. N. Adya and I. L. Markov. Consistent placement of macro-blocks using floorplanning and standard-cell placement. In *Proc. of ISPD*, pages 12–17, 2002.

[2] Aeroflex Gaisler. LEON3 Processor. Available at `http://www.gaisler.com`, 2005.

[3] ARM Ltd. Memory IP. `http://www.arm.com/`.

[4] B. Bailey and G. Martin. *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer-Verlag, 2006.

[5] 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*, 13(4):1–26, Oct. 2008.

[6] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. `http://hpc.pnnl.gov/projects/PERFECT/`.

[7] L. Benini, L. Macchiarulo, A. Macii, and M. Poncino. Layout-driven memory synthesis for embedded systems-on-chip. *IEEE Trans. on Very Large Scale Integration Systems*, 10(2):96–105, Apr. 2002.

[8] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(9):1059–1076, Sept. 2001.

[9] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Proc. of the Design Automatic Conf.*, pages 1233–1238, June 2012.

[10] J. Correa, N. Megow, R. Raman, and K. Suchan. Cardinality constrained graph partitioning into cliques with submodular costs. In *Proc. of CTW*, pages 347–350, 2009.

[11] P. Coussy and A. Morawiec. *High-level synthesis: from algorithm to digital circuit*. Springer, 2008.

[12] 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. (ESLsyn)*, pages 1–6, June 2013.

[13] G. Di Guglielmo, C. Pilato, and L. P. Carloni. A Design Methodology for Compositional High-Level Synthesis of Communication-Centric SoCs. In *Proc. of the Design Automatic Conf.*, pages 1–6, 2014.

[14] M. Fingeroff. *High-level Synthesis Blue Book*. Mentor Graphics Corporation, 2010.

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

[16] T. R. Kumar, R. Govindarajan, and C. Ravikumar. On-chip memory architecture exploration framework for DSP processor-based embedded system on chip. *ACM Trans. on Embedded Computing Systems*, 11(1):1–25, Apr. 2012.

[17] S. Li, N. Farahini, A. Hemani, K. Rosvall, and I. Sander. System level synthesis of hardware for DSP applications using pre-characterized function implementations. In *Proc. of the Int.l Conf. on Hardware/Software Codesign and System Synthesis*, pages 1–10, Oct. 2013.

[18] S. Li and A. Hemani. Memory allocation and optimization in system-level architectural synthesis. In *Proc. of ReCoSoC*, pages 1–7, July 2013.

[19] H.-Y. Liu, M. Petracca, and L. P. Carloni. Compositional system-level design exploration with planning of high-level synthesis. In *Proc. of DATE*, pages 641–646, Mar. 2012.

[20] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[21] P. R. Panda, N. D. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni, and E. de Greef. Data memory organization and optimizations in application-specific systems. *IEEE Design & Test of Computers*, 18(3):56–68, 2001.

[22] C. Pilato, V. G. Castellana, S. Lovergine, and F. Ferrandi. A runtime adaptive controller for supporting hardware components with variable latency. In *Proc. of AHS*, pages 153–160, June 2011.

[23] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-Chip: Reuse and Integration. *Proceedings of the IEEE*, 94(6):1050–1069, 2006.

[24] Y. Tatsumi and H.-J. Mattausch. Fast quadratic increase of multiport-storage-cell area with port number. *Electronics Letters*, 35(25):2185–2187, 1999.

[25] M. Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proc. of the Design Automatic Conf.*, pages 1131–1136, June 2012.

[26] Y. Wang, P. Li, and J. Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proc. of FPGA*, pages 199–208, Feb. 2014.

[27] Y. Wang, P. Zhang, X. Cheng, and J. Cong. An integrated and automated memory optimization flow for FPGA behavioral synthesis. In *Proc. of ASPDAC*, pages 257–262, Jan. 2012.

[28] Xilinx. Virtex-7 FPGA Family. `http://www.xilinx.com/`.

[29] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. Improving high-level synthesis optimization opportunity through polyhedral transformations. In *Proc. of FPGA*, pages 9–18, Jan. 2013.