Code Manipulation for Virtual Platform Integration

Sara Vinco[1], Valerio Guarnieri[2], Franco Fummi[2]
[1] Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy
[2] Department of Computer Science, University of Verona, Verona, Italy

***Abstract*—Simulation speed is crucial in virtual platforms, in order to enhance the design flow with early validation and design space exploration. This work tackles this challenge by focusing on two main techniques for speeding up virtual platform simulation, namely efficient data types implementation and a novel scheduling technique. Both the optimizations are obtained through code manipulation. The target language is C++ and its extensions (*i.e.*, SystemC), that are the most widespread languages for virtual platform modeling and simulation. The optimization techniques are considered orthogonal, as they target different aspects of the simulated code. Experimental results prove the effectiveness of both the single techniques and of their combined application on complex case studies, with the result of reaching a maximum speedup of 70x in the simulation of a virtual platform.**

***Keywords*—Virtual platform, data types, SystemC, C++ code generation, scheduling, simulation, SystemC optimization.**

I. Introduction

Virtual platforms are a powerful solution in embedded system design, as they allow to explore alternative design solutions, to achieve early validation of the overall system and to develop and validate applications on top of the HW components. In this context, simulation speed is a pressing concern, as a slow simulation would slow down the entire design and validation process.

Most of the currently available virtual platform environments target simulation speedup by adopting C++ and SystemC TLM [5], [15], [17], [23]. On one hand, C++ native constructs and types allow a lightweight and fast execution. On the other, SystemC, particularly in its TLM extension, eases integration and reuse of existing IPs, while still preserving good simulation performance and adherence *w.r.t.* the IP functionality.

The choice of C++ and SystemC as target languages reduces the type of IPs and models that can be reused, as models implemented in other HDLs can not be straightforwardly included in the virtual platform. Co-simulation would overcome this limitation [1], [16], but it would decrease simulation performance due to the frequent synchronization between different simulation environments. At the same time, many state-of-the-art methodologies and tools allow to automatically convert HDLs to C++ and SystemC [3], [14], [22]. However, such tools focus on the reproduction of the IP functionality, with no concern for simulation performance and optimization. On the contrary, code conversion for inclusion in virtual platforms requires a further effort to achieve not only correct simulation but also good simulation speed, to allow multiple executions and to enhance the development of the virtual platform.

This work was supported by the EC co-funded SMAC (SMArt Systems Co-Design) project Grant Agreement FP7–ICT–288827.

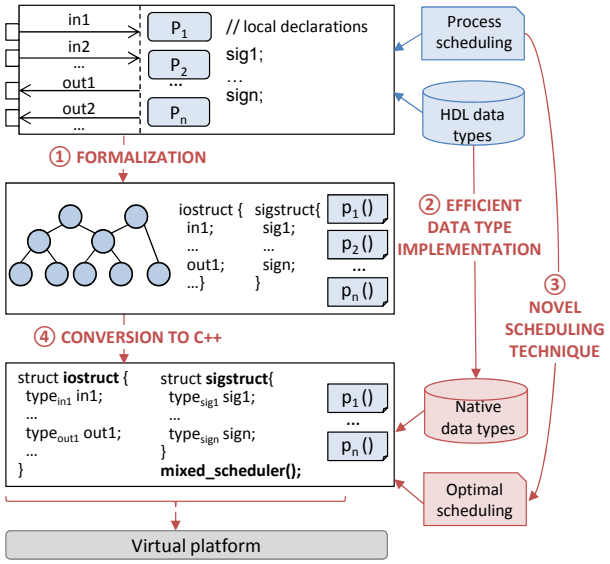

Fig. 1. Overview of the proposed approach. The starting IP is reduced to an intermediate HDL-independent format (1), and it then optimized via efficient data type implementation (2) and a novel scheduling technique (3). The code is then converted to C++ for inclusion in virtual platforms (4).

In this scenario, this article aims at enhancing virtual platforms by easing the integration of existing IPs, and by speeding up code execution through the adoption of novel optimization techniques. The methodology goes beyond standard C++ code generation solutions to reduce the impact of the two major bottlenecks: data type implementation and scheduling of the HDL processes [7], [8]. The optimizations are considered orthogonal, as they target different aspects of the simulated code and they require specific solutions.

Figure 1 outlines the resulting approach. The starting point is a digital IP, in any HDL, that must be inserted into a C++/SystemC virtual platform. Information extracted from the IP (*e.g.*, inter-process dependencies and data declarations) is formalized to ease code manipulation (1). HDL data types are converted to native C++ data types to speed up simulation (2). HDL scheduling is then replaced by a novel scheduling technique, mixing static and dynamic scheduling to execute each subset of processes of the original IP with the most suitable scheduling strategy (3). The resulting description is finally converted to C++ for integration into virtual platforms (4).

The main contributions of the current work are:

- *enhancement of data types* through a novel methodology that converts HDL data types into C++ native data types. Type conversion is supported by the modification of the

IP code and the implementation of an efficient library of operators, supporting typical HDL constructs;

- an *analysis of traditional static and dynamic scheduling approaches*, to define their limitations and to determine what conditions make the one more efficient than the other;
- construction of a *new scheduling technique*, that combines the advantages of both approaches by executing each subset of processes of the IP with the most suitable scheduling strategy, still preserving inter-process dependencies;
- *implementation of automatic tools* that apply all code manipulations to existing IPs, *i.e.*, DDT for data type-based optimizations and TANGLE for automating the adoption of the novel scheduling strategy. The tools are based on an existing framework for code manipulation, HIFSuite [3]. However, all manipulation and formalization steps related to the proposed methodology have been implemented for the first time in TANGLE and DDT.

The result is an optimal configuration, that allows to reach a speedup of up to 474x when applied to single IPs, and a speedup of up to 70x when applied to a complex cryptographic platform. The approach proves thus to be especially suited for repeated simulation sessions, such as the case of virtual platform simulation during design exploration and validation.

The article is organized as follows. Section II provides the necessary background. Section III formalizes information extracted from the IP. Section IV focuses on data types. Section V outlines the novel scheduling strategy, while Section VI focuses on C++ code generation. Sections VII and VIII conclude the paper with experiments and with our remarks.

II. Background

A. Efficient data type implementation

The popularity of C++ and SystemC for virtual platform environments is due to the lightweight and fast execution offered by C++, and by the enhanced support for integration and reuse offered by the SystemC extensions. SystemC data types support all types and operators offered by the main HDLs, thus recreating the correct behavior of the physical circuit in a C++ environment. Unfortunately, SystemC types decrement simulation performance due to an inefficient implementation [8]. Different works have been proposed in the literature with the aim of defining a trade-off between accuracy and simulation speed of SystemC data types. [27] exploits native types to increase performance, but it limits the support to concatenation, bit selection and range operators. Performance close to native C/C++ data types is reached in [8], [24], providing arbitrary-length integer, fixed-point and complex data types. However, logic data types are not handled and the level of abstraction is too low, as the aim is to ease synthesis. The result is a limited speedup (about 2x). [13] supports only the unsigned integer data type `uc_uint`, for substituting the `sc_uint` SystemC type. The type `uc_uint` is provided only with a very limited range of operators (*i.e.*, assignment, bit selection and read operations), thus limiting the support for HDL types and constructs.

[2] proposed an efficient bit-accurate data type library, called HDTLib, that complies with the SystemC standard. Optimized execution is gained by avoiding dynamic memory allocation and by reducing class hierarchy to the minimum. Furthermore, unsigned integers are used as underlying data structure to perform operations on words, rather than on single bits. However, HDTLib is still affected by performance penalties due to its implementation as C++ template classes, *e.g.*, overheads associated with function calls and class hierarchy.

B. Static and dynamic scheduling

Dynamic scheduling is the reference scheduling policy for HDL simulators. It is strictly event-based, as each process is executed only when fresh data are available. The major bottlenecks are the management effort to determine the queue of runnable processes, and potentially repeated executions of processes in a single simulation cycle in response to multiple activation events.

Static scheduling has been adopted in literature in various scenarios, *e.g.*, for execution on massively parallel architectures [28] or in case of real-time systems [4]. It determines a static execution order between processes based on the intra-process dependencies. Most of the approaches in the literature assume that the dependency graph is acyclic, claiming that cycles can not be written in synthesizable code [4], [21]. Unfortunately, cyclic dependencies are not as rare as one may think (projects *systemcaes*, *a_vhd_16550_uart* and *plasma* from [19]). Even if some attempts to remove cycles have been made [12], [20], cycle removal mostly relies on the designer's knowledge and its automation is thus far from trivial.

To the authors' knowledge, no work in the literature targeted the creation of a scheduling approach mixing static and dynamic. All available scheduling strategies either try to make simulation very efficient and easy to control (*i.e.*, dynamic scheduling) or meet the requirements of a specific architecture (*e.g.*, static scheduling for GP-GPUs [28]).

III. Formal representation of the starting IP

HDL languages (*e.g.*, VHDL and Verilog) have major differences in terms of both syntax and semantics. It is thus necessary to represent the main characteristics and information of IPs in a common language-independent format.

The application of the proposed methodology requires to extract the following information from any given IP: (1) list of the ports constituting the IP interface, (2) list of signal declarations, for allowing inter-process communication, (3) processes of the IP, modeled as functions, and (4) a dependency graph, for modeling inter-process dependencies.

An example of IP and corresponding extracted information is provided in Figure 2. The following sections deepen the information extraction process. The intermediate format is represented in a tree-structured XML-like language including all objects and statements of a typical HDL language, that allows to apply the same methodology independently from the language and the characteristics of the starting IP.

1) Hierarchy removal: The proposed methodology forces hierarchy flattening as hierarchy smoothes the dependencies between processes, that are not instantaneously evident. This step does not constitute a novel contribution, as it relies on

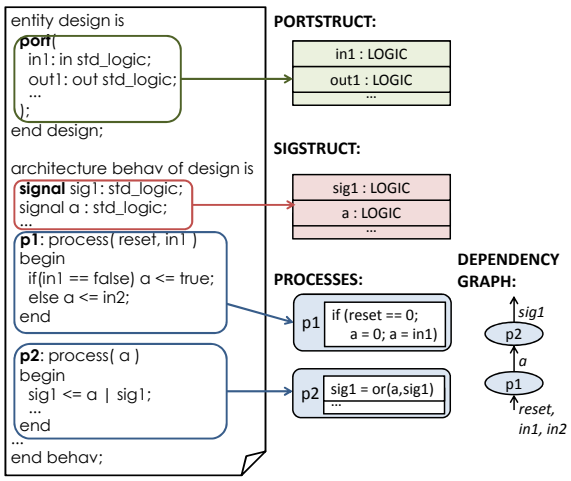

Fig. 2. Adopted intermediate format. Ports and signals are grouped into two lists (`iostruct` and `sigstruct`, respectively). Processes are modeled as functions, and their dependencies are formalized in a dependency graph.

the HR tool, part of HIFSuite [3], which manipulates the starting code to remove the hierarchy and returns a new non-hierarchical IP equivalent to the original one. However, its presentation was necessary to better explain the characteristics of the adopted IP description.

The hierarchy of any IP is removed by starting from the non-hierarchical instances, and moving up across the hierarchy levels. Each sub-component is flattened in its parent component by moving all its internal declarations and processes to the parent. To avoid conflicts, all declarations and processes are given fresh names. Then, operations on ports of the sub-component are replaced with operations on the signals of the parent that were originally bound to each sub-component port.

2) Data declarations: The HDL-independent format lists all declarations as objects (*e.g.*, port objects, signal objects, variable objects). All such objects are provided with a name (*i.e.*, the name in the starting HDL). As shown in Figure 2, the type of each object is obtained by mapping the original HDL type (*e.g.*, `std_logic`) to a HDL-independent type (*e.g.*, `LOGIC`). Finally, port objects are annotated with their direction.

To ease code generation and encapsulation, data declarations of the starting IP are grouped into two lists, outlined in Figure 2. *Ports* are collected in a list called `iostruct`, storing the direction, the HDL-independent type and the name of the port. *Signals* are collected in a list called `sigstruct`, whose entries store the name of the signal and its HDL-independent type. In order to preserve system consistency all along simulation, signals are duplicated to preserve both the current value of the signal at any simulation time, and its value after the execution of the previous simulation cycle. This reproduces the semantics of most HDLs (*e.g.*, VHDL, Verilog and SystemC).

3) Processes: Aside from the difference in terms of syntax, all HDLs model HW processes as functions executed in response to system events, such as changes in the input ports or in the value of signals. In the intermediate format, processes are converted to functions, named $p_i()$, where i is an index. The body of each function is the list of actions of the original process (*e.g.*, assignments, conditions and loops). Each function accesses data declarations through pointers to the `iostruct` and `sigstruct` lists, so that changes to the system state are visible from all functions.

4) Process management: Moving the IP to C++ implies that all scheduling and event management routines are not preserved. On the other hand, *inter-process dependencies*, that constrain both the activation (and the ordering) of processes and the data propagation flow across the IP, must be preserved to ensure the correctness of any scheduling routine.

To fill this gap, the intermediate format is enriched with a management function, called `mixed_scheduler()`, in charge of executing the HDL processes in an order that respects all dependencies (*i.e.*, the sensitivity list of each process) and that allows to detect all changes in the system status. The behavior of the `mixed_scheduler()` function is determined at later stages of the proposed approach (step 3, Section V). However, its construction requires a preliminary effort to formalize read and write dependencies between processes.

The intermediate format represents inter-process dependencies with a specific type of graph, called *dependency graph* (Definition 1). In a dependency graph, each vertex represents a process and points to the corresponding function $p_i()$. Vertexes are divided between synchronous (*i.e.*, processes that react to changes of the clock signal) and asynchronous (*i.e.*, processes that are activated by other signals). An edge connects two vertexes if the source vertex writes on one or more signals read by the destination vertex. Given edge $e = (v, v')$, signal s_e is the signal that determines the dependency of v' from v.

Definition 1 (Dependency graph)**.** *A dependency graph* $\mathsf{DG} = (\mathsf{V}, \mathsf{E})$ *is a direct graph built as follows:*

- V *is the set of vertexes of the graph. Each vertex corresponds to one of the design processes and it is provided with a pointer* F *to the corresponding function* $p_i()$*. A flag* sync *is used to state whether the process is sensitive to changes of the clock signal,* i.e.*, synchronous (true), or asynchronous (false).*
- $\mathsf{E} \subseteq (\mathsf{V} \times \mathsf{V})$ *is the set of edges between vertexes. An edge* $e = (v, v') \in \mathsf{E}$ *exists if process* v' *reads a signal* s *written by process* v *and* v'.sync *is false.*

  The condition on v' *implies that the dependencies of synchronous processes from asynchronous processes are not represented, since they create a dependency between present-state values and next-state values through time.*

Figure 3 shows an example of dependency graph. Colored vertexes are asynchronous nodes (*i.e.*, sync is false), while synchronous nodes (*i.e.*, sync is true) are dashed.

Note that, depending on design quality and programming style, the signals included in any process sensitivity list may be only a subset of the read signals. Whenever the sensitivity list of a process does not include all read signals, this may lead to losing some read-write dependencies, and thus to scheduling errors due to an incorrect execution order of processes. This explains why the dependency graph is built depending on signals read by each process, rather than on the process sensitivity list.

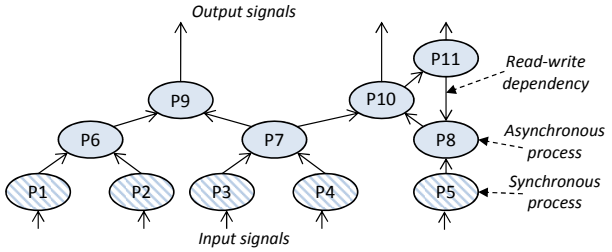

Fig. 3. Example of dependency graph. Vertexes represent processes and edges represent read-write dependencies between processes. Colored vertexes are asynchronous nodes, while synchronous vertexes are dashed. Each vertex is labeled with the name of the corresponding function $p_i()$.

TABLE I. MAPPING OF LOGIC AND BIT VECTORS INTO C++ FIXED-WIDTH UNSIGNED INTEGER TYPES.

Bitwidth range	Fixed-width integer type
2–8	`uint8_t`
9–16	`uint16_t`
17–32	`uint32_t`
33–64	`uint64_t`
> 64	Multiple `uint64_t` chunks

IV. EFFICIENT DATA TYPE IMPLEMENTATION

Data types have a heavy impact on both the functionality and the performance of the generated code [7], [9]. In the context of the development of C++/SystemC based virtual platforms, a straightforward solution is to map HDL data types to SystemC data types. Nevertheless, SystemC proved to be up to 10x slower than the other HDLs, mainly because of the heavy data types class hierarchy [9]. This Section aims at overcoming such limitations by proposing a methodology to convert HDL data types into C++ native data types.

A. CONVERSION OF HDL DATA TYPES TO C++ NATIVE DATA TYPES

This section proposes a comprehensive methodology to convert HDL data types to C++ native data types, made of three main steps: (1) retrieving data declarations, (2) converting types of data declarations to the most suitable C++ native types, (3) converting operations on data types.

1) Retrieving data declarations: the first step consists of retrieving all the data declarations (*i.e.*, ports, signals and variables) in the design description. Signal and port declarations are available in the `sigstruct` and `iostruct` lists, built at formalization time. Then, variable declarations internal to processes are collected in lists, to be easily accessed and to preserve the scope of each variable.

2) Converting data types: the type of each data declaration is then examined and converted to a corresponding C++ native data type. Single bit or logic types are replaced with the native **`bool`** type. Bit and logic vectors are replaced with corresponding fixed-width unsigned integer types, as defined in the `<cstdint>` header of the C++ standard library. Given a bit or logic vector, the corresponding fixed-width unsigned integer type is selected as the smallest that can safely contain the bitwidth of the vector, as shown in Table I. Note that this may lead to map logic and bit vectors to an unsigned integer type larger than their bitwidth. For example, a 24-bits vector is mapped to `uint32_t`, a 32-bits unsigned integer.

In case the bitwidth is larger than 64 bits, the vector is split into multiple unsigned integer chunks. These chunks are all of type `uint64_t`, except for the last chunk, containing the most significant bits, that may fit into a smaller unsigned integer. For example, a 160-bits logic vector `v` is converted into the following chunks:

```
uint64_t v_63_0;       // 64 least significant bits
uint64_t v_127_64;     // bits from 64 to 127
uint32_t v_159_128;    // 32 most significant bits
```

HDL logic types typically extend two-valued logic (*i.e.*, `'0'` and `'1'`) with a set of *metavalues* (*e.g.*, `'U'`, `'X'`, `'Z'`). Metavalues are used for debugging purposes and for simulating HW-specific behaviours, such as uninitialized, unknown, or high-impedance values. Fully reproducing this accuracy would introduce a performance overhead, as each logic value would be mapped to a number of bits (4 to reproduce the 9-valued logic of VHDL). For this reason, the data type mapping proposed in this work abstracts the multi-value logic and replaces all metavalues with `'0'`, similarly to the data abstraction methodology in [2]. This allows to map logic values to **`bool`** and logic vectors to unsigned integers, as done for bits and bit vectors. This transformation can be deemed as a reasonable, as metavalues model a low-level behavior no longer needed during virtual platform simulation. The loss of accuracy is thus considered an acceptable trade-off in order to significantly speed up simulation.

3) Converting operations on data types: operations on data types must be changed accordingly to take into account the conversion of data declarations. To this extent, typical HDL operators are implemented with corresponding bitwise operations or function calls, as detailed in Table II, where column *HDL syntax* provides a representation of the operation in Verilog syntax. This shows that adopting C++ native types for efficient simulation requires both the definition of a library of operators, reproducing the typical HDL operators, and heavy code manipulations, to replace HDL types and operators with the efficient ones.

In case logic and bit vectors are mapped to an unsigned integer type larger than their bitwidth, bitwise masking operations are added to the right-hand side of assignments on those vectors, in order to preserve bitwise accuracy. For example, suppose `a` is a 24-bits logic vector (mapped to the `uint32_t` type). `a` is assigned the sum `b + c`. The corresponding assignment is implemented as follows:

```
a = (b + c) & 0xffffffU;
```

where the bit mask `0xffffffU` ensures that `a` is not assigned values larger than the maximum allowed by its bitwidth.

If a logic or bit vector is split into multiple unsigned integer chunks, operations on the vector are decomposed into the involved chunks. For example, suppose the slice `v[95:32]` of the 160-bits logic vector `v` is accessed. This slice access is converted to the following expression:

```
((v_127_64 & 0xffffffffULL) << 32) |
((v_63_0 & 0xffffffff00000000ULL) >> 32)
```

TABLE II. IMPLEMENTATION OF OPERATIONS ON HDL DATA TYPES WITH CORRESPONDING BITWISE OPERATIONS ON UNSIGNED INTEGERS.

Operation	HDL syntax	Implementation	Notes
Slice access	`a[hi:lo]`	`(a & mask)>> lo`	`mask` = $(2^{(hi-lo+1)}-1)$ `)<< lo`
Slice assignment	`a[hi:lo] = b`	`a = (a & mask)| (a << lo)`	`mask` = `(~0)-((`$2^{(hi-lo+1)}-1$`) << lo)`
Single bit access	`a[i]`	`(a & mask)== mask`	`mask` = 2^i

Single bit assignment	`a[i] = b`	`a = singleAssign(a, i, b)`
		`uint64_t singleAssign(uint64_t l, int i, bool b) {`
		`  uint64_t mask = 1ULL << i;`
		`  if (b) return (l | mask);`
		`  else return (l & (~mask));`
		`}`	

Concatenation	`{a, b}`	`(a <<` w_b`) | b`	w_b = bitwidth of operator `b`
Bitwise and	`a & b`	`a & b`	
Bitwise or	`a | b`	`a | b`	
Bitwise xor	`a ^ b`	`a ^ b`	
And reduction	`&a`	`(a == mask)`	`mask` = $2^{w_a}-1$, w_a = bitwidth of operator `a`
Or reduction	`|a`	`(a > 0)`	

Xor reduction	`^a`	`xorReduce(a,` w_a`)`
		`bool xorReduce(uint64_t v, int w) {`
		`  bool result = (v & 1ULL) == 1ULL;`
		`  uint64_t mask = 2ULL;`
		`  for (int i = 1; i < w; ++i) {`
		`  result ^= ((v & mask) == mask);`
		`  mask <<= 1;`
		`} }`
			w_a = bitwidth of operator `a`

Now suppose that `v` is assigned the concatenation of 64-bits vector `a` and 48-bits vectors `b` and `c`:

```
v = {a, b, c};
```

This assignment is decomposed into three separate assignments, one for each chunk of `v`:

```
v_63_0 = ((b & 0xffffULL) << 48) | c;
v_127_64 = ((a & 0xffffffffULL) << 32) |
    ((b & 0xffffffff0000ULL) >> 16);
v_159_128 = (a & 0xffffffff00000000ULL) >> 32;
```

Finally, the most widespread arithmetic libraries (*e.g.*, `ieee.numeric_std` of VHDL) are supported by implementing their operations on C++ native data types. Additional arithmetic libraries can be easily supported by mapping their operations on corresponding expressions and function calls operating on C++ native data types.

V. A NOVEL SCHEDULING TECHNIQUE

Scheduling is usually considered an underlying dimension of IP simulation in the context of embedded system design. Scheduling algorithms are not considered as a target for optimization, and the state-of-the-art versions of HDL scheduling algorithms are considered as standard-de-facto. To overcome their heavy impact on IP simulation [7], many approaches have been proposed in the literature targeting parallel simulation of HDLs, with a focus on SystemC simulation [6], [10], [18], [29]. Such solutions, despite being effective at reducing simulation speed, are not suitable for standard virtual platforms, as they require highly (or massively) parallel architectures.

This work proposes to overcome such limitations by defining a novel scheduling approach that goes beyond standard static and dynamic scheduling approaches.

A. ANALYSIS OF STATIC AND DYNAMIC SCHEDULING

Dynamic scheduling execution time is made of two major contributions: process execution and scheduling management. The term $\#iter_{j,i}$ represents the number of executions of process p_j at simulation cycle i, while the scheduling management cost at cycle i is represented by the term $mgmt_i$. Given t_j average execution time of process p_j, the resulting execution time is thus:

$$T_{dynamic} = \sum_{0 \le i < \#cycles} (mgmt_i + \sum_{\substack{0 \le j < \\ \#process}} \#iter_{j,i} \times t_j) =$$

$$(\#cycles \times \sum_{\substack{0 \le j < \\ \#synch \\ process}} t_j) + \sum_{\substack{0 \le i < \\ \#cycles}} (mgmt_i + \sum_{\substack{0 \le j < \\ \#async \\ process}} (\#iter_{j,i} \times t_j))$$

where the latter formula further breaks down execution contribution between synchronous processes (executed exactly once at any simulation cycle) and asynchronous processes (whose execution is less predictable).

Overall execution time of *static scheduling* is the sum of the execution time of all processes, accumulated for all clock cycles. Given t_j average execution time of process p_j:

$$T_{static} = \#cycles \times \sum_{0 \le j < \#process} t_j.$$

The presented formulas are very hard to apply to real case studies without any profiling information. On the other hand, the formulas are useful for providing some guidelines for the application of static or dynamic scheduling. In both the versions, synchronous processes are executed once at any simulation cycle. Dynamic scheduling outperforms static scheduling if asynchronous processes execute sporadically, as management time $mgmt_i$ is compensated by the time saved because of executing processes only on fresh data. On the contrary, if asynchronous processes execute often, then the $mgmt_i$ overhead outweighs the benefit of restricting execution to ready processes, thus making static scheduling more efficient.

An experimental evidence of this analysis is applied in the following, by profiling and comparing the execution of two designs with complementary characteristics. The CAMELLIA

design is a Verilog cryptographic core [26], while the ECC is a VHDL error correction code module provided by an industrial partner. Table III compares their characteristics in terms of execution times in their purely dynamic and static scheduling versions, and in terms of profiling.

TABLE III. ANALYSIS OF SIMULATION PERFORMANCE AND PROFILING OF THE DES56 AND ECC DESIGNS IN THEIR STATIC AND DYNAMIC SCHEDULING VERSIONS.

		CAMELLIA	ECC
Processes	Synchronous	6	4
	Asynchronous	71	7
Dynamic	Execution time (s)	129.4	99.8
	Processes per cycle (avg. #)	114.6	8.5
	Mgmt. overhead (%)	0.9	24.2
Static	Execution time (s)	64.9	145.9
	Processes per cycle (avg. #)	77.0	11.0

The CAMELLIA design is better in its static scheduling version (speedup 2.0x), as it executes on average 77.0 processes per simulation cycle, versus the 114.6 of the dynamic version. This means that in the dynamic version some of the processes are executed more than once (even more than 3 times) per simulation cycle. This slows down execution, even if the management overhead has a very low impact on execution (0.9%). On the contrary, the ECC design is better in its dynamic version (speedup 1.5x), which executes on average 8.5 processes per simulation cycle, instead of 11.0 as for the static version. Indeed, not all processes are activated at any simulation cycle, and only in few cases the same process is executed more than once. This balances the management overhead, that accounts for slightly more than 24.0% of execution time. Such considerations explain the different performance of the two designs when adopting static and dynamic scheduling, and prove that no scheduling strategy is optimal for any design.

B. EXTENDING STATIC SCHEDULING WITH CYCLE SUPPORT

Static scheduling is based on the assumption that there are no cyclic dependencies between processes. Unfortunately, cyclic dependencies are not as rare as one may intuitively think: examples of cyclic designs are available at [19] (projects *systemcaes*, *a_vhd_16550_uart* and *plasma*). Cyclic dependencies may be difficult to detect, *e.g.*, they may occur between processes at different hierarchy levels. Furthermore, cyclic dependencies may not raise any error during the synthesis phase whenever the dependencies are between different cones of logic contained in the processes, rather than the entire processes. As a result, the restriction to non-cyclic designs severely limits the applicability of static scheduling. One of the goals of this paper is to go beyond this limitation, to allow applicability of static scheduling to any kind of designs.

1) Cycle identification: The identification of cycles in a dependency graph can be easily accomplished by applying Tarjan's algorithm for enumerating elementary circuits [25]. If the algorithm identifies any cycle in the dependency graph (*e.g.*, the graph between $P2$, $P3$ and $P4$ in Figure 4.a), it is then necessary to break the cyclic dependencies to make the dependency graph acyclic. This allows to apply the topological order and to create the static scheduling for the current design.

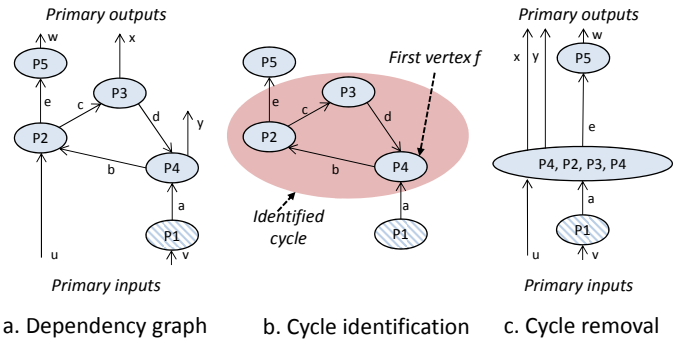

Fig. 4. Example of cyclic dependency graph (a) and of application of the cycle management approach (b-c).

To ease the presentation of the cycle management strategy, it is now necessary to define the *first vertex of a cycle*.

Definition 2 (First vertex of a cycle)**.** *Given a dependency graph* DG = (V, E) *and a cycle contained in* DG*, the first node f is a vertex of the cycle such that:*

- $\exists e = (v, f) \in$ E *s.t.* v.sync *is true,* i.e.*, the node receives inputs from a synchronous process;*
- $\forall\ e = (f, v)$ *belonging to to the cycle,* $!\exists e' = (v', f)$ *such that e' belongs to the cycle and the value of s_e is computed from $s_{e'}$.*

The first condition implies that f is the first function to execute, as it gets fresh data from a synchronous process. The second condition states that any output signal s_e of the first vertex must obey this constraint:

- either edge e does not belong to the cycle, and thus any new value produced for s_e does not provide fresh data to processes of the cycle;
- or edge e belongs to the cycle, but s_e is not function of any signal $s_{e'}$ belonging to the cycle.

This guarantees that there is no cyclic dependency on signals, thus making the cycle synthesizable. Indeed, all outputs produced by f and belonging to the cycle are not influenced by the execution of the other processes of the cycle, thus guaranteeing that the cycle will never be executed an infinite number of times. Note that the synthesizability of the IP guarantees the existence of at least one first vertex per cycle. If no vertex in the cycle can be first vertex, than the cycle may be activated an infinite number of times also in its dynamic scheduling version. If more than one vertex qualifies as first vertex of a cycle, the first vertex is the process with the lightest computational effort.

An example of identification of the first vertex of a cycle is provided in Figure 4.b. Given the cycle made of $P2$, $P3$ and $P4$, vertex $P4$ has an input edge coming from the synchronous vertex $P1$, thus satisfying the first condition. $P4$ can be identified as first vertex if and only it satisfies the second condition, even if it has an output edge (b) belonging to the cycle. The condition is thus satisfied only if the value of s_b (*i.e.*, of the signal associated to the output edge b) does not depend on the value of s_d (*i.e.*, of the signal associated to the

input edge d, that is part of the cycle).

2) Cycle removal: Given a cycle in a dependency graph, the cycle is broken by replacing it with a new vertex, v_new. The function pointer of the new node (*i.e.*, v_new.F) reproduces the functionality of the whole cycle, as determined by visiting the cycle starting from the first vertex f and following the cycle edges until f is reached again (lines 7-10 of Algorithm 1). In this way, f is executed as first function (in response to its synchronous dependencies) and as last function of the cycle (in response to system changes due to the execution of the cycle).[1] The edges forming the cycle are then removed from the dependency graph (lines 11-13). The input and output edges of the vertexes forming the cycle are then moved to the new vertex (lines 14-17 for input edges and 18-21 for output ones). This allows to preserve the dependencies *w.r.t.* the remainder of the design.

As an example, Figure 4.c shows that the previously identified cycle (made of $P2$, $P3$ and $P4$) is replaced by a new vertex associated with the functionality of all processes listed as: $P4$ (*i.e.*, the first vertex), $P2$, $P3$ and $P4$. Input edges of the original vertexes (*i.e.*, u and a) constitute the input edges of the new vertex, while e, x and y are its output edges.

```
1  break_cycle(cycle, DG) begin
2        f = cycle.find_first_vertex();
3        v_new = new vertex();
4        V.add(v_new);
5        V.remove(cycle.get_vertexes());
6        v_new.F = v_new.F ∪ f.F;
7        foreach v ∈ cycle s.t. v ≠ f by topological sort do
8            v_new.F = v_new.F ∪ v.F;
9        end
10       v_new.F = v_new.F ∪ f.F;
11       foreach (v, v') ∈ E s.t. v,v' ∈ cycle do
12           E = E \ {(v, v')};
13       end
14       foreach v ∈ cycle ∀ (v', v) ∈ E s.t. v' ∉ cycle do
15           E = E \ {(v', v)} ∪ {(v', v_new)};
16           s_(v',v_new) = s_(v',v);
17       end
18       foreach v ∈ cycle ∀ (v, v') ∈ E s.t. v' ∉ cycle do
19           E = E \ {(v, v')} ∪ {(v_new, v')};
20           s_(v_new,v') = s_(v,v');
21       end
22 end
```

Algorithm 1: Algorithm for breaking a cyclic dependency

a) Discussion on correctness: Let us consider a cycle made of a number of vertexes v_i, such as the one formed by processes $P2$, $P3$ and $P4$ in Figure 4. Section V-B1 showed that, if the IP is synthesizable, the cycle must contain at least one node f that is a first vertex node (in this case, $P4$).

By definition, f receives fresh data both from synchronous processes (*i.e.*, $P1$) or external inputs, and from inside the cycle (*i.e.*, from $P3$). The synchronous dependency implies that f is the first node of the cycle to be executed with any scheduling strategy. At this point, inputs coming from inside the cycle (*i.e.*, d) are not stable yet, as the other processes have not executed yet. However, synchronous inputs are stable (*i.e.*, a) and it is thus possible to compute the updated value for the outputs involved into the cycle (*i.e.*, b).

Then, all cycle vertexes are executed one at a time, by following the order implied by the cycle (*i.e.*, $P2$ and then $P3$). This implies that each process executes as soon as all its input data are available, by respecting the dependency constraints.

Finally, f gets fresh data from vertexes inside the cycle (*i.e.*, d from $P3$). The constraints posed in Definition 2 imply that such fresh data are not involved in the computation of the value of output signals belonging to the cycle (*i.e.*, b is not function of the value of d). Thus, outputs for the cycle are not updated and it is not necessary to reactivate the other cycle processes. This is consistent with dynamic scheduling, where the other cycle processes would not be executed anyhow, as no event occurred on their inputs.

Finally, fresh data are used to update the value of the process outputs outside the cycle (*i.e.*, y), that are propagated in the following of the execution. This confirms that the algorithm for cycle removal preserves both the correctness of execution and all dependency constraints.

3) Cycle management approach: Our experimental analysis proves that designs tend to contain a number of nested cycles, rather than a single cycle (*e.g.*, project *plasma* from [19] contains 194 cycles). Given a generic dependency graph, cycles are broken one after another by starting from cycles that do not contain nested cycles, and in order of size (*i.e.*, the cycle containing the fewest vertexes is broken first). The cycle removal algorithm modifies the graph with new vertexes and edges. Thus, it is necessary to reorder the cycles after any execution of the cycle removal algorithm. Once all cycles have been eliminated, it is possible to apply the standard topological sort algorithm, and to build the static scheduling for the design.

C. Beyond static and dynamic scheduling

Section V-A highlighted that no scheduling approach is the best solution overall, as static and dynamic scheduling approaches are suited to different types of execution profiles. As such, it is worth trying to mix static and dynamic scheduling, thus building a novel solution targeting frequent simulations in virtual platforms. The proposed solution adopts each scheduling approach only in the most suitable conditions, combining the advantages and reducing the drawbacks. Note that the extension of static scheduling to support a wider range of designs (achieved in the previous section) further augments the chances for evaluating alternative scheduling associations.

The resulting flow is depicted in Figure 5:

a. construction of the *dependency graph* of the IP;
b. *partitioning of the dependency graph* into sets of vertexes, executed with the same scheduling approach;
c. *association of scheduling approaches to sets*, depending on heuristic policies or on designer's configurations;
d. sets are executed sequentially by a *mixed scheduler* by respecting dependency constraints.

[1] Note that removing a cycle results in replicating the first vertex f. This explains why, whenever two or more vertexes are eligible for being the first vertex, the first vertex is the process with the lighter computational effort.

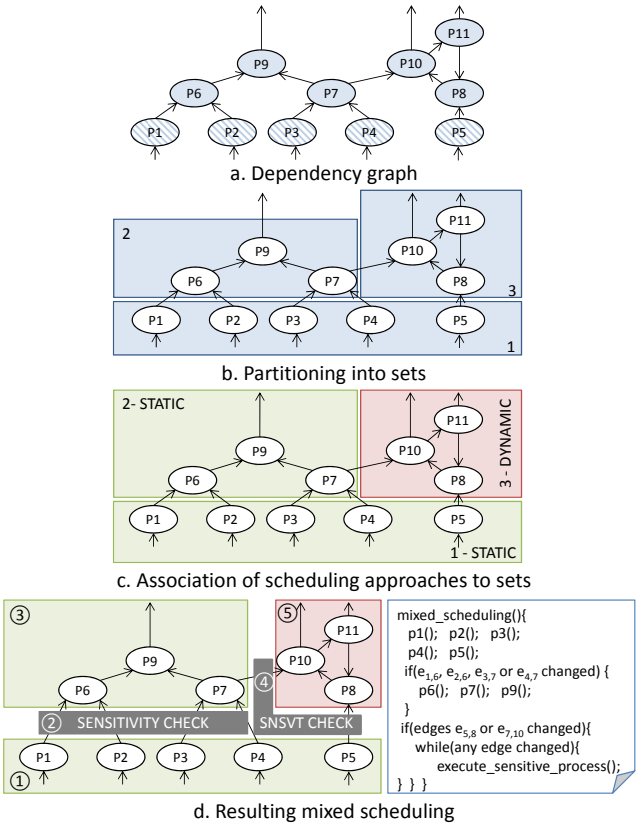

Fig. 5. Proposed approach. The dependency graph (a) is partitioned into sets (b). Each set is associated with the most suitable scheduling policy (c). The generated mixed scheduler sequentially executes the sets if any incoming edge changed value (d).

1) Partitioning into sets: The dependency graph is partitioned into subgraphs, called *sets* (the solid rectangles in Figure 5.b). Vertexes in the same set represent processes that are executed at the same time with the same scheduling approach. Some constraints are defined to preserve the inter-process dependencies:

- *Synchronous processes:* all synchronous processes must belong to the same set (*e.g.*, set 1 in Figure 5.b). In HDL simulation, synchronous processes are the first ones to execute at the beginning of a new simulation cycle. To reflect this behavior, they are collected in a single set, called *synchronous set*, that is executed at the beginning of any simulation cycle.
- *Ancestors and descendants:* a set can not contain both ancestors and descendants of a vertex, unless the vertex itself is contained in the set. This condition guarantees that sets can be ordered by respecting the dependencies outlined in the dependency graph. An example of application of this constraint is outlined in Figure 6. In Figure 6.a, set 2 contains both an ancestor (*i.e.*, $P4$) and a descendant (*i.e.*, $P10$) of $P7$, that is contained in set 1. This implies that set 2 should be executed both before ($P4$-$P7$ dependency) and after set 1 ($P7$-$P10$ dependency). A possible correct partitioning is achieved by partitioning set 2 into two separate sets (Figure 6.b).

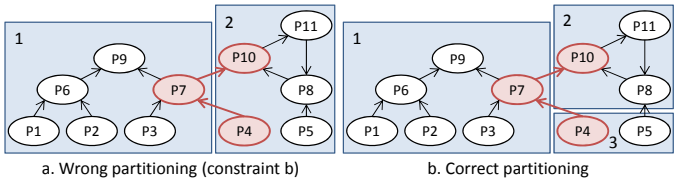

Fig. 6. Effect of the ancestors and descendants constraint on set partitioning.

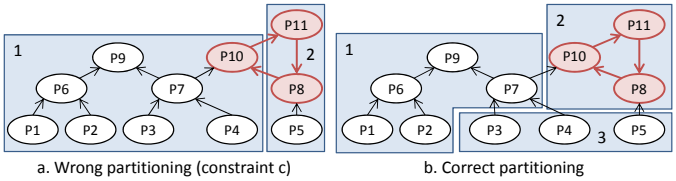

Fig. 7. Effect of the cycle management constraint on set partitioning.

- *Cycle management:* all vertexes of a cycle must belong to the same set, to allow a correct management of the inter-process dependencies. Separating vertexes belonging to the same cycle (as done in Figure 7.a with vertexes $P10$, $P11$ and $P8$) would create a cyclic dependency between the sets. An example of correct partitioning is provided in Figure 7.b, where the cycle is mapped to set 2 and sets can finally be executed in the order 3 - 1 - 2.

2) Association of scheduling approaches to sets: Association of scheduling approaches to sets (Figure 5.c) is mainly based on heuristic policies and on the considerations about the optimal conditions for static and dynamic scheduling reported in Section V-A. This step leaves space to design space exploration and to the evaluation of alternative associations to determine an optimal solution. This may require some effort from the designer, but support for automatic code generation eases and speeds up the process. Once an optimal result is found, the generated code can be used for multiple executions of the virtual platform, thus balancing the effort.

a) Synchronous processes: The synchronous set is always associated with static scheduling (set 1 of Figure 5.d). Indeed, all synchronous processes are executed at the beginning of a simulation cycle, with no need to check activation conditions.

b) Cycle management: The extension to static scheduling presented in Section V-B allows to handle cycles both with static and dynamic scheduling. Applying cycle removal to cycles does not result in a huge computational effort, as the proposed solution executes all processes of the cycle and replicates only one of them. However, our experimental analysis proves that designs tend to contain a number of nested cycles, rather than a single cycle (as the experimental section will show). This leads to an explosion of the computational overhead, that cancels the benefit of reducing event management costs. Thus, dynamic scheduling is the optimal approach for cycle scheduling. This guarantees optimal performance, since cycle execution is suspended whenever no fresh data are available.

c) Heuristic policy: Finding an optimal balance between static and dynamic scheduling for a given design is a matter

of exploration of possible alternatives. However, it is possible to define policies to guide the choice of the designer.

- *Time-based policy:* this accurate policy is derived by the considerations in Section V-A. The idea is to determine the ratio (at any simulation time) between the computational cost of all process of a set *w.r.t.* the management cost derived from the system update and event evaluation routine. When considering set i (set_i in the following), the proposed heuristic is:

$$\frac{\sum_{sim.cyclej} \sum_{p_k \in set_i} t(p_k, j)}{\sum_{cyclej} \sum_{p_k \in set_i} mgmt(p_k, j)} \geq^{?} 1$$

  where $t(p_k, j)$ and $mgmt(p_k, j)$ are the execution time and the management cost of process p_k at simulation cycle j, respectively. This estimation is accurate and effective. However, it requires very detailed profiling information (*i.e.*, the execution time of any given process and its management cost at any simulation cycle). Unfortunately, properly recording such information typically demands a much finer time granularity than the time resolution employed by profilers, thus making the application of this heuristic very difficult.

- *Invocation-based policy:* a more effective heuristic considers the average number of invocations of each process per cycle. When considering set i, the heuristic is calculated as:

$$\frac{\sum_{p_j \in set_i} invocations(p_j)}{(|set_i|(\#cycles))} \geq^{?} 1$$

  where $|set_i|$ is the number of processes in set i. This metric determines whether, on average, processes are executed at least once per simulation cycle. If so, static scheduling is a winning solution, as it would limit the number of executions per simulation cycle to 1. Otherwise, the set should adopt dynamic scheduling. This heuristic is far easier to apply, as any profiler would estimate the number of process invocations. Still, it is really effective, as it relates the adopted scheduler to the execution profile. Adding execution weight of each process invocation would make the heuristic more detailed, but once again this information would be really difficult to profile.

3) Set scheduling: Final code execution implies to execute the identified sets one after another, by following a topological order. Sets (except for the synchronous one) are executed dynamically, *i.e.*, they are predicated by a check on their incoming signals (Figure 5.e). This means that each set is executed only if fresh data are available for its processes, no matter the chosen scheduling approach. The result is that, if no new data are available, a lot of computational effort is avoided. This allows to balance the presence of static scheduling sets. As a result, the final generated code interleaves set execution with sensitivity list checks, as shown on the left hand side of Figure 5.d (where circled numbers show the execution order).

4) Discussion on correctness: The correctness of the partitioning into sets is guaranteed by the defined constraints, that avoid cyclic dependencies between sets and that at the same time guarantee that cyclic dependencies between processes are preserved. Once the sets have been identified, their scheduling is entirely based on a topological sort of the dependency graph, that respects by definition inter-set and inter-process dependencies.

The proposed approach guarantees that processes do not miss any event or fresh data. In case of dynamic scheduling, processes are executed as soon as any event is available on their inputs. At the same time, in case of static scheduling, processes are executed after all the processes that may produce fresh inputs have finished. This guarantees that no event is missed, despite the adopted scheduling.

Finally, correctness of inter-set scheduling relies on the correctness of standard dynamic and static scheduling.

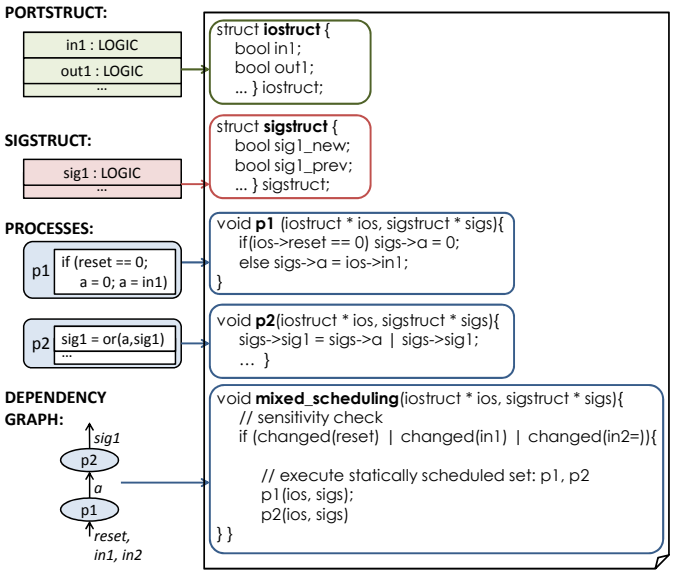

Fig. 8. C++ code generation applied Figure 2. The lists of ports and signals are implemented as structs, and processes are converted to functions. The `mixed_scheduling()` function reproduces the scheduling routine.

VI. C++ CODE GENERATION

This section focuses at first on C++ code generation for a single IP (Section VI-A) and then on integration issues (Section VI-B).

A. C++ CODE GENERATION FOR A SINGLE IP

The main steps of C++ code generation are: (i) data structure declaration, (ii) processes and (iii) implementation of the scheduling function (Figure 8).

The first conversion step is applied to *data structures*. The `iostruct` and `sigstruct` lists are implemented as C++ **`struct`** declarations. The fields of each struct are declared by replacing the HDL-independent types of each entry with the corresponding C++ native data types (by following the conversion proposed in Section IV). Each IP instantiates one instance of `iostruct` and one of `sigstruct`.

The functions $p_i()$ associated with the starting HDL processes are converted by translating each statement into a corresponding C++ statement. Each function receives as parameters

pointers to the `iostruct` and `sigstruct` instances. Operations applied to ports or signals are converted to operations on the fields of the `iostruct` and `sigstruct` structs.

The `mixed_scheduler()` function is finally implemented by alternating process invocations and checks on signals, as determined by set scheduling and by the scheduling approach adopted in each set. As a result, the `mixed_scheduler()` is a mix of static and dynamic scheduling, reflecting the result of the strategy in Section V. Each invocation of the `mixed_scheduler()` function corresponds to the execution of one clock cycle in the starting HDL IP.

The HDL design can now be integrated in virtual platforms as C++ code. In order to ease integration, the generated code can be wrapped with SystemC TLM modules or with interfaces specific of the adopted integration environment. It is important to note that this would affect only the interface of the generated code, with no decrement of simulation performance.

B. Integration issues and solutions

The methodology proposed so far focuses on a single IP, where any level of hierarchy has been flattened to expose all inter-process dependencies (as explained in Section III). However, virtual platforms are adopted for the simulation of complex systems. Flattening the overall platform may be extremely complex, and the resulting description may explode in the attempt of flattening a high number of levels of hierarchy. Moreover, the generated code may have to interact with existing IPs, designed with different approaches and wrapped in C++-based languages, *e.g.*, SystemC and SystemC TLM. It is thus necessary to define how code generated with the proposed approach can interact with the surrounding system.

1) Orchestration of different IPs: Whenever two or more IPs have to be integrated, a global `mixed_scheduler()` function is generated for orchestrating overall execution. Section VII-F shows an example of integration on a cryptographic virtual platform.

The global `mixed_scheduler()` function reproduces overall system evolution by invoking all IPs `mixed_scheduler()` functions one after the other. Since the HDL signal semantics is preserved (*i.e.*, signals are duplicated to separate current and future value), invocation order does not affect system evolution. The underlying assumption is that all data exchanges between IPs happen synchronously[2].

In case IPs have different clock periods, the system clock period is the least common divider of all IPs clock periods. Each invocation of the global scheduler function reproduces one system clock period, and each IP is activated whenever necessary to respect its starting clock period.

2) Integration in existing platforms: Whenever necessary, *e.g.*, for enhancing reuse or reducing design costs, it is possible to mix code generated with the proposed approach with existing IPs and platforms.

If an IP must be integrated with C++/SystemC/SystemC-TLM IPs and platforms, its `mixed_scheduler()` function must be wrapped to allow seamless integration. In case of C++, a global scheduling routine must be generated, to activate all C++ IPs consistently *w.r.t.* their temporal relationship (*e.g.*, in terms of clock cycle). In case of SystemC, the generated `mixed_scheduler()` function is declared as a synchronous process and it is wrapped by a SystemC module, having as interface the ports listed in the `iostruct` structure. Finally, in case of SystemC-TLM, the `mixed_scheduler()` function is encapsulated by a TLM wrapper, following either a blocking or non-blocking interface. Each transaction required from the TLM module corresponds to one invocation of the `mixed_scheduler()` function.

If existing IPs are otherwise implemented in non-C++-based languages (*e.g.*, VHDL and Verilog), they must be converted to C++ by using any technique available at state of the art (including the approach proposed in this work). The integration process follows the previously outlined strategies.

[2]In case of asynchronous communication, the IPs must be treated as a single IP and flattened, not to miss events and synchronization points.

TABLE IV. Benchmark characteristics.

Designs	PI (#)	PO (#)	Processes (#)		HDL language	Lines of code
			Sync.	Async.		
AES	260	129	5	26	Verilog	1,776
CAMELLIA	262	131	6	71	Verilog	887
DES56	132	67	3	23	VHDL	1,054
ECC	25	46	4	7	VHDL	180
MLITE	36	100	7	46	VHDL	2,315
PAR_FIB	6	256	1	13	VHDL	148
RAM_BIST	48	47	2	15	VHDL	451
XTEA	195	64	3	12	VHDL	278

VII. Experimental results

The benchmarks used for experimental analysis are:

- four cryptographic cores: AES, DES56 and XTEA from [19] (projects *systemcaes*, *BasicDES* and *xteacore*) and CAMELLIA from [26];
- an industrial error correction code module (ECC);
- a MIPS processor (MLITE) from [19] (project *plasma*);
- a parallel Fibonacci numbers calculator (PAR_FIB);
- an industrial BIST module for a simple-port synchronous RAM (RAM_BIST).

Table IV reports the main characteristics of the designs, in terms of primary inputs and outputs (columns *PI (#)* and *PO (#)*, respectively), number of synchronous and asynchronous processes (column *Processes*), IP language and lines of code.

All experiments have been carried out on a 64-bit Linux server with 6 2.53 GHz CPU cores and 16 GB RAM memory. All the alternative implementations of the same design are executed with the same testbench, which consists of the translation to SystemC and C++ of the original testbench for the design. Functional equivalence between the different versions of each design has been dynamically verified by comparing resulting waveforms. Simulation times are calculated as an average over a number of executions.

A. Methodology automation

Manually applying code manipulations to complex designs may result in a time-consuming and error-prone process, which may reduce by far the effectiveness and usefulness of the

TABLE V. IMPACT OF DATA TYPES ON SIMULATION PERFORMANCE.

Designs	SystemC	HDTLib		DDT	
	Time (s)	Time (s)	Speedup (x)	Time (s)	Speedup (x)
AES	8,484.4	4,436.6	1.9	132.5	64.0
CAMELLIA	25,641.4	19,394.5	1.3	423.9	60.5
DES56	19,534.2	14,903.3	1.3	293.9	66.5
ECC	993.4	290.1	3.4	120.3	8.3
MLITE	3,374.7	2,605.1	1.3	232.4	14.5
PAR_FIB	454.8	336.6	1.4	82.8	5.5
RAM_BIST	1,422.8	403.3	3.5	157.4	9.0
XTEA	2,935.6	826.6	3.6	79.7	36.8

achieved speedups. Thus, all the proposed transformations have been automated and manual configurations and interventions have been reduced to the bare minimum.

The methodology presented in Section IV-A has been implemented in DDT, which automatically converts HDL data types to C++ native data types and maps operations on HDL data types to corresponding C++ operations and function calls.

Application of the mixed scheduling approach has been automated in the novel tool TANGLE, which uses the GNU profiling tool gprof [11] to gather profiling information for the heuristic policies. Note that TANGLE is capable of generating alternative versions of the same design with different scheduling policies, as specified by user-defined configurations.

TANGLE and DDT are based on the commercial suite HIFSuite [3]. In detail, HIFSuite front-end tools are used to convert the starting IPs to the HIF intermediate language. This description is then manipulated by TANGLE and DDT through the HIF APIs to apply all the presented code transformations. Note that all transformations, including the extraction of the formal representation, are based on HIF APIs, but they are by no means natively included in HIF, as they rather have been implemented for the sake of the proposed methodology. Finally, HIFSuite back-end tools are used to convert the resulting description to C++. Thus, even if the newly developed tools make an extensive use of HIFSuite interfaces and tools, all manipulation steps related to the proposed methodology have been implemented ex novo in TANGLE and DDT.

B. DATA TYPE OPTIMIZATION

To determine the sole impact of data types on simulation performance, it is necessary to compare versions of the same design having a single language and scheduling approach, but different data type implementations. To this extent, all benchmarks have been converted to C++ by adopting standard HDL scheduling for process management.

As reference implementation, we adopted SystemC data types to emulate the conversion of each design for insertion into a C++-based virtual platform. This version is then compared against the adoption of the HDTLib data type library [2] and of the C++ native data types proposed in this work. Table V shows the corresponding simulation times and the speedup of the latter data types *w.r.t.* SystemC types. HDTLib achieves a fairly limited speedup (up to 3.6x), while still retaining accuracy *w.r.t.* multi-value logic. Conversely, C++ native types greatly enhance simulation performance of all designs (up to 66.5x), at the expense of accuracy *w.r.t.* multi-value logic.

TABLE VI. DEPENDENCY GRAPHS AND EXECUTION TIMES WITH THE DYNAMIC AND STATIC SCHEDULING APPROACHES.

Designs	DG			Simulation time (s)	
	|V|	|E|	Cycles	Dynamic	Static
AES	27	74	y	106.1	228.9
CAMELLIA	46	69	n	129.4	64.9
DES56	18	25	n	115.3	100.2
ECC	11	8	n	99.8	145.9
MLITE	51	121	y	67.6	1,092.7
PAR_FIB	13	30	n	123.2	53.8
RAM_BIST	17	42	n	141.9	345.7
XTEA	11	10	n	74.2	87.3

C. SCHEDULING OPTIMIZATION

To determine the sole impact of the scheduling policy on simulation performance, it is necessary to compare versions of the same design adopting a single data type library (*i.e.*, SystemC types), but different scheduling approaches. The dynamic and static version of each design have been generated through HIFSuite. The dynamic version of each design features the event-based process scheduler that is typically adopted in HDL simulations (*i.e.*, a lightweight implementation in C++ of the SystemC simulation kernel). The static version of each design implements the static scheduling approach that can be found in the literature [12].

Table VI shows the main characteristics of the generated dependency graphs, in terms of number of vertexes |V| and edges |E|, and of presence of cycles. The reported simulation times of the static and dynamic versions for each design show that none of the scheduling approaches wins on all designs. Static scheduling in faster on the CAMELLIA, DES56, and PAR_FIB designs, while dynamic scheduling is faster on the ECC, RAM_BIST and XTEA designs. Dynamic scheduling also performs better on cyclic designs AES and MLITE, since the process replication in presence of multiple nested cycles significantly raises simulation times. This proves that performance strictly depends on the design characteristics, rather than solely on the adopted scheduling approach.

a) Partitioning into sets: Table VII shows the result of applying set partitioning to the benchmarks, defined as number of processes per set (column *Proc. (#)*). The table shows also whether each set contains synchronous processes (*Synch. (y/n)*) and cycles (*Cycle (y/n)*). Such characteristics straightforwardly determine the type of scheduling to be adopted, *i.e.*, static if the set is synchronous, dynamic if the set contains cycles. If such conditions do not hold, column *Heuristic ratio* reports the heuristic policy ratio. This allows to determine the ideal scheduling policy for each set (*Sched. policy*).

b) Association of scheduling approaches to sets: Table VIII shows code performance when applying the scheduling policies determined in Table VII (row *Heuristic application* of each design). This is compared with alternative scheduling policies, as specified by user-defined configurations. For each configuration, the table reports execution time and differences in terms of scheduling choices.

c) Performance of the generated code: The comparison between the execution times of all the code versions is depicted, for the sake of clarity, in Figure 9. *Config 1* always represents the version following the proposed heuristic policy.

TABLE VII. SET PARTITIONING AND HEURISTIC APPLICATION.

Design	Set ID	Proc. (#)	Sync (y/n)	Cycle (y/n)	Heuristic ratio	Sched. policy
AES	1	5	y	n	-	Static
	2	2	n	n	0.439	Dynamic
	3	2	n	n	1.710	Static
	4	1	n	n	1.498	Static
	5	21	n	y	-	Dynamic
CAMELLIA	1	6	y	n	-	Static
	2	17	n	n	0.434	Dynamic
	3	8	n	n	1.020	Static
	4	5	n	n	1.011	Static
	5	11	n	n	1.613	Static
	6	12	n	n	1.991	Static
	7	4	n	n	2.435	Static
	8	10	n	n	2.473	Static
	9	3	n	n	2.957	Static
	10	1	n	n	3.064	Static
DES56	1	3	y	n	-	Static
	2	3	n	n	0.916	Static
	3	1	n	n	0.903	Static
	4	1	n	n	1.847	Static
	5	8	n	n	1.814	Static
	6	1	n	n	2.790	Static
ECC	1	4	y	n	-	Static
	2	4	n	n	0.251	Dynamic
	3	1	n	n	1.000	Static
	4	1	n	n	0.506	Dynamic
MLITE	1	7	y	n	-	Static
	2	3	n	n	0.105	Dynamic
	3	43	n	y	-	Dynamic
PAR_ FIB	1	1	y	n	-	Static
	2	2	n	n	1.000	Static
	3	2	n	n	1.500	Static
	4	2	n	n	2.000	Static
	5	2	n	n	2.500	Static
	6	2	n	n	3.000	Static
	7	2	n	n	3.500	Static
	8	1	n	n	4.000	Static
RAM_ BIST	1	2	y	n	-	Static
	2	10	n	n	0.063	Static
	3	4	n	n	0.095	Dynamic
XTEA	1	3	y	n	-	Static
	2	9	n	n	0.251	Dynamic
	3	1	n	n	0.500	Dynamic
	4	1	n	n	1.000	Static
	5	1	n	n	1.440	Static

The code generated with the mixed scheduling approach is always at least as fast as the best between static and dynamic scheduling, as it executes all processes with the most suitable approach. The mixed scheduling version is up to 2.3x faster than the dynamic scheduling version and up to 17.4x faster than the static one. These considerations highlight that adopting the mixed scheduling approach allows to exploit design characteristics to the full.

The effectiveness of the proposed heuristic approach is also proved by a comparison with alternative scheduling choices. The mixed scheduling version is always the fastest:

- if the heuristic ratio of a set is lower than 0.9, static execution leads to running processes exactly once per simulation cycle, thus slowing down simulation. This explains the slowdown of configuration 2 of MLITE, and configurations 2 and 3 of AES, ECC, RAM_BIST and XTEA;
- if the heuristic ratio of a set is higher than 0.9, dynamic execution of the set leads to executing asynchronous processes, on average, at least once per simulation cycle, and to paying the management overhead. This explains the slowdown of configurations 2 and 3 of CAMELLIA, DES56, PAR_FIB and XTEA and of configuration 3 of AES and ECC.

The 0.9 threshold to determine scheduling policies has been empirically chosen after experiments. It is slightly less than the expected value of 1.0 (*i.e.*, exactly once per simulation cycle) since it takes into account the scheduling management cost

TABLE VIII. MIXED SCHEDULING WITH INVOCATION BASED HEURISTIC APPLICATION AND DESIGN SPACE EXPLORATION.

Designs	Configuration (#)	Configuration Characteristics	Simulation time (s)
AES	1	Heuristic application	83.4
	2	2 static	84.3
	3	2 static, 3 and 4 dynamic	84.8
CAMELLIA	1	Heuristic application	63.1
	2	3 dynamic	67.1
	3	3 and 4 dynamic	69.5
DES56	1	Heuristic application	62.2
	2	6 and 7 dynamic	124.5
	3	1 and 7 dynamic	94.4
ECC	1	Heuristic application	90.8
	2	2 static	155.4
	3	3 dynamic, 4 static	91.6
MLITE	1	Heuristic application	62.8
	2	2 static	63.1
PAR_FIB	1	Heuristic application	53.2
	2	2 dynamic	55.8
	3	2, 3, and 4 dynamic	66.1
RAM_BIST	1	Heuristic application	74.9
	2	2 static	75.1
	3	3 static	75.7
XTEA	1	Heuristic application	66.6
	2	2 and 3 static, 4 and 5 dynamic	75.6
	3	2 static, 4 and 5 dynamic	79.2

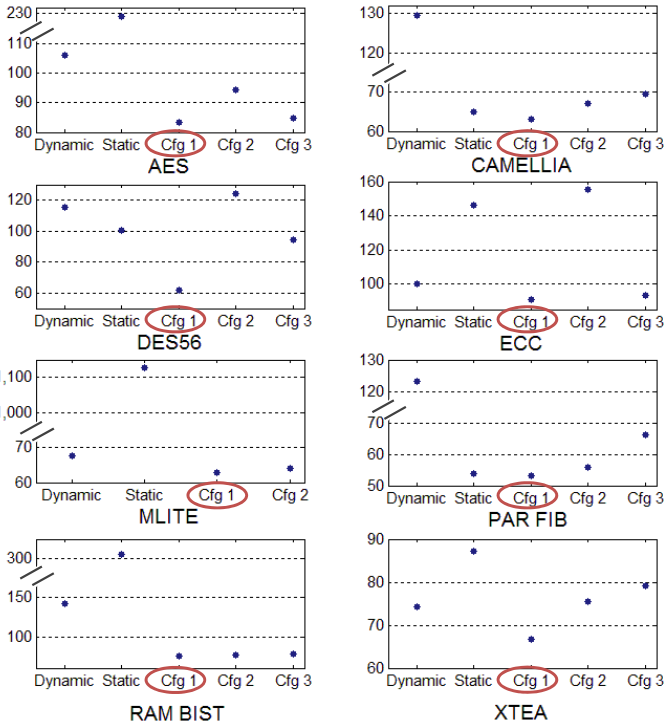

Fig. 9. Execution time (in seconds) of the different configurations provided for each design, including dynamic and static scheduling and mixed scheduling. Execution time of the proposed approach is labelled as *Config 1* (circled).

associated with dynamic scheduling. In fact, it is more efficient to statically execute processes for which the heuristic ratio is slightly less than 1.0 (*i.e.*, belonging to the range $[0.9, 1]$), because the performance penalty due to extra executions turns out to be less than the scheduling management cost.

The speedups prove that performance relies on both an optimal partitioning and on an effective heuristic, that exploits the features of each set to get the best performance.

D. Discussion on static management of cycles

Table IX summarizes the characteristics of the cyclic designs adopted in this experimental evaluation, *i.e.*, AES and MLITE. Both designs are hierarchical, with a high number of cycles hidden from the designer by the presence of multiple levels of hierarchy. To make things worse, these cycles are nested, which has a heavy impact on the cycle removal algorithm, as processes duplicated to remove inner cycles may be replicated a number of times when removing outer cycles. This leads to an explosion of the number of replicated processes.

TABLE IX. Effect of static cycle management on hierarchical cyclic designs.

Designs	Hierarchy (y/n)	Cycles (#)	Simulation time (s)	
			Dynamic	Static
AES	y	32	106.1	228.9
MLITE	y	194	62.8	1,092.7

Columns *Dynamic* and *Static* report simulation times by adopting the standard HDL scheduling and static scheduling, respectively. The table shows that the application of static scheduling is not convenient, as this approach pays too hefty a price to remove nested cycles from the process graph. The extremely high number of cycles in both designs renders the process replication strategy highly inefficient from a simulation performance standpoint. This is particularly evident in the MLITE design, for which the static version is more than one order of magnitude slower than the dynamic version.

Despite the impact on performance, it is important to note that static scheduling management of cycles constitutes an important contribution to state of art. Indeed, it allows to apply techniques restricted to non-cyclic designs, thus leaving space for further optimization and for the application of highly efficient parallelization approaches [18], [28].

E. Effectiveness of the proposed approach

The previous sections presented the results of the proposed approach by focusing on the single optimizations. Table X combines the effects of both data type optimization and of scheduling optimization, to highlight the effectiveness of the overall proposed approach.

The table adopts as reference simulation time the performance obtained by simulating each design with SystemC data types and with standard HDL scheduling (Column *HDL scheduler + SystemC types (s)*). This emulates the conversion of each design to SystemC, for insertion into a C++-based virtual platform. Column *Full optimization* displays the effect of combining both the optimizations by applying data type optimization to the best scheduling configuration. Simulation results prove that the proposed techniques are orthogonal and mutually influence each other. Data types have a stronger impact on simulation time, as they reduce time spent both in process execution and in event management for the dynamic scheduler versions. At the same time, scheduling optimization allows to further improve simulation performance, by executing each set with the most suitable scheduling approach. The achieved speedups greatly vary from design to design according to the following three design characteristics: (1) the subdivision of the functionality of the design between synchronous and asynchronous processes (as synchronous computation is left unchanged by the proposed approach), (2) the presence of asynchronous processes executing on average multiple times per clock cycles, (3) the amount and the computational weight of operations performed on data types. The result is a maximum speedup of 441x, which is higher than the speedups gained with the single optimization techniques. A speedup of two orders of magnitude has a significant impact on simulation, and can be considered an important improvement of simulation performance.

TABLE X. Simulation speedup obtained with the application of the proposed approach.

Designs	HDL scheduler + SystemC types (s)	Full optimization	
		Time (s)	Speedup (x)
AES	8,484.4	106.7	79.5
CAMELLIA	25,641.4	58.1	441.3
DES56	19,534.2	70.8	275.9
ECC	993.4	106.8	9.3
MLITE	1,342.9	67.6	19.9
PAR_FIB	454.8	58.8	7.7
RAM_BIST	1,422.8	100.4	14.2
XTEA	2,935.6	58.6	50.1

F. Application to a cryptographic platform

The proposed experimental results show performance on relatively short executions. However, the achieved speedup scales also on longer executions, and becomes even more significant in virtual platforms, where simulations would be frequently repeated to evaluate alternative configurations.

For this reason, we applied the overall optimization methodology to the cryptographic platform in Figure 10, composed of a subset of the analyzed benchmarks. It includes the cryptographic cores (AES, DES56 and CAMELLIA) and the MLITE processor. A simple memory is used by all the other modules to communicate and share data.

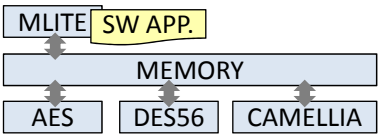

Fig. 10. The cryptographic platform.

Table XI shows simulation times of the cryptographic platform. Column *HDL scheduler + SystemC types* reports the reference simulation time of the description featuring SystemC data types and standard HDL scheduling. Column *Full optimization* displays the simulation time after applying data type and scheduling optimizations. Integration of the components

was achieved by applying the technique outlined in Section VI-B1. Column *Speedup* shows the achieved speedup, which is definitely significant, as simulation goes from about 334 minutes to less than 5 minutes. This is especially important in the context of virtual platforms usage, as simulations have to be repeated very frequently.

TABLE XI. SIMULATION TIMES OF THE CRYPTOGRAPHIC PLATFORM.

HDL scheduler + SystemC types (s)	Full optimization	
	Time (s)	Speedup (x)
20,048.4	287.1	69.8

VIII. CONCLUDING REMARKS

The paper proposed two code manipulation techniques for optimizing simulation of IPs in the context of virtual platforms, *i.e.*, data types implementation and process scheduling. Both techniques proved to achieve a good speedup, and their combined application allowed to achieve a maximum speedup of 441x on single IPs, and of 70x on a cryptographic virtual platform. Future work will extend the proposed approach with an evaluation of alternative mechanisms for set invocation and with the generation of wrappers for integration in commercial virtual platform environments.

REFERENCES

[1] M. Bombana and F. Bruschi. SystemC-VHDL co-simulation and synthesis in the HW domain. In *Proc. of IEEE/ACM DATE*, pages 101–105, 2003.

[2] N. Bombieri, F. Fummi, et al. HDTLib: an efficient implementation of SystemC data types for fast simulation at different abstraction levels. *Design Automation for Embedded Systems*, 16(2):115–135, 2012.

[3] N. Bombieri, G. D. Guglielmo, M. Ferrari, et al. HIFSuite: Tools for HDL code conversion and manipulation. *EURASIP Journal on Embedded Systems*, 2010(436328):1–20, 2010.

[4] R. Buchmann and A. Greiner. A fully static scheduling approach for fast cycle accurate SystemC simulation of MPSoCs. In *Proc. of IEEE ICM*, pages 101–104, 2007.

[5] Cadence. Virtual System Platform. www.cadence.com.

[6] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory. Relaxing synchronization in a parallel systemc kernel. In *Proc. of IEEE ISPA*, pages 180–187, 2008.

[7] A. Donlin. Optimizing Models of an FPGA Embedded System. nascug.org, 2004.

[8] W. Ecker. Impact of SystemC data types on execution speed. In *www-ti.informatik.uni-tuebingen.de*, 2007.

[9] W. Ecker, V. Esen, L. Schonberg, et al. Impact of description language, abstraction layer, and value representation on simulation performance. In *Proc. of ACM/IEEE DATE*, pages 767–772, 2007.

[10] P. Ezudheen, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi. Parallelizing SystemC kernel for fast hardware simulation on SMP machines. In *Proc. of ACM/IEEE PADS*, pages 80–87, 2009.

[11] GNU. GNU Binutils, 2014. www.gnu.org/software/binutils/.

[12] R. Guindi and Y. Naguib. SplitPro: A tool to overcome SystemC scheduling inefficiencies. In *Proc. of IEEE ICM*, pages 347–350, 2010.

[13] F. Herrera. *Heterogeneous Specification and Automatic Software Generation from SystemC for Embedded Systems*. PhD thesis, University of Cantabria, 2008.

[14] HT-Lab. VH2SC. www.ht-lab.com/freeutils/vh2sc/vh2sc.html.

[15] Imperas Software. OVP - Open Virtual Platforms. www.ovpworld.org.

[16] R. Maciel, B. Albertini, et al. A Methodology and Toolset to Enable SystemC and VHDL Co-simulation. In *Proc. of IEEE VLSI*, pages 351–356, 2007.

[17] Mentor Graphics. Vista Virtual Prototyping for SystemC / TLM 2.0 and QEMU. www.mentor.com/esl/vista/virtual-prototyping.

[18] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. *Proc. of ACM/IEEE ASP-DAC*, pages 149–154, 2010.

[19] OpenCores, 2014. opencores.org.

[20] D. Perez, G. Mouchard, and O. Temam. A new optimized implementation of the SystemC engine using acyclic scheduling. In *Proc. of IEEE/ACM DATE 2004*, volume 1, pages 552–557, 2004.

[21] F. Petrot, D. Hommais, and A. Greiner. Cycle precise core based hardware/software system simulation with predictable event propagation. In *Proc. of EUROMICRO*, pages 182–187, 1997.

[22] Sourceforge. Verilog2C++. verilog2cpp.sourceforge.net.

[23] Synopsys. Platform Architect. www.synopsys.com.

[24] A. Takach, P. Gutberlet, and S. Waters. Fast bit-accurate C++ datatypes for functional system verification and synthesis. In *Proc. of ECSI FDL*, pages 337–345, 2004.

[25] R. E. Tarjan. Enumeration of the Elementary Circuits of a Directed Graph. Technical report, Cornell University, Ithaca, NY, USA, 1972.

[26] Tohoku University. Cryptographic hardware project - Aoki Laboratory, 2007. www.aoki.ecei.tohoku.ac.jp.

[27] G. Tumbush and M. Hupp. Dramatically increase the performance of SystemC simulations. In *Proc. of DVCon Conference*, 2007.

[28] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi. SAGA: SystemC acceleration on GPU architectures. *Proc. of ACM/IEEE DAC*, pages 115–120, 2012.

[29] H. Ziyu, Q. Lei, L. Hongliang, X. Xianghui, and Z. Kun. A parallel SystemC environment: ArchSC. In *Proc. of ICPADS*, 2009.

Sara Vinco (M'09) received the Ph.D. degree in computer science from the University of Verona, Verona, Italy, in 2013. She is currently a Post-Doctoral Research Associate at the Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy. Her main research interests are energy efficient electronic design automation and techniques for simulation and validation of heterogeneous embedded systems.

Valerio Guarnieri received the Masters and PhD degrees in Computer Science from the University of Verona, Verona, Italy, in 2009 and 2013. He is currently a Postdoctoral Research Associate at the Department of Computer Science of the University of Verona. His main research interests are abstraction of RTL descriptions to TLM/C++ and integration of hardware descriptions into virtual platforms.

Franco Fummi (M'92) received the Ph.D. degree in electronic engineering from Politecnico di Milano, Milan, Italy, in 1995. He is currently the Head of the Department of Computer Science, University of Verona, Verona, Italy, where he is a Full Professor, since 2000, and where he became an Associate Professor in computer architecture in 1998. Since 1995, he has been with the Department of Electronics and Information, Politecnico di Milano, as an Assistant Professor. He is a cofounder of EDALab, an EDA company developing tools for the design of networked embedded systems. His current research interests include electronic design automation methodologies for modeling, verification, testing, and optimization of embedded systems.