A Design Methodology for Compositional High-Level Synthesis of Communication-Centric SoCs

Giuseppe Di Guglielmo, Christian Pilato, Luca P. Carloni
Dept. of Computer Science - Columbia University, New York, NY - USA
{giuseppe,pilato,luca}@cs.columbia.edu

ABSTRACT

Systems-on-chip are increasingly designed at the system level by combining synthesizable IP components that operate concurrently while interacting through communication channels. CAD-tool vendors support this System-Level Design approach with high-level synthesis tools and libraries of interface primitives implementing the communication protocols. These interfaces absorb timing differences in the hardware-component implementations, thus enabling compositional design. However, they introduce also new challenges in terms of functional correctness and performance optimization. We propose a methodology that combines performance analysis and optimization algorithms to automatically address the issues that SoC designers may accidentally introduce when assembling components that are specified at the system level.

Categories and Subject Descriptors
B.5 [**RTL Implementation**]: Design Aids
General Terms
Algorithms, Design, Experimentation
Keywords
High-Level Synthesis, SystemC

1. INTRODUCTION

The complexity of modern Systems-on-Chip (SoC) is driving the adoption of Electronic System Level (ESL) design methodologies that raise the level of abstraction above RTL design and promote the reuse of components [13]. These components are increasingly specified with high-level programming languages, such as C++ and SystemC, to speed up system-level simulation and enable their implementation as energy-efficient hardware accelerators [18].

Fig. 1 shows an abstraction of the hardware part of a typical ESL design flow. An application (or part of an application) that will be implemented in hardware is specified as a set of components which operate concurrently while interacting through communication channels. These components are expressed, for instance, as *synthesizable SystemC processes*. Their specifications may be the result of partitioning and refinement steps from a higher-level algorithmic description or may be taken from libraries of pre-designed Intellectual Property (IP) blocks. High-level synthesis (HLS) tools are used to synthesize a hardware implementation for each component after performing various micro-architectural optimizations. Indeed, given a SystemC process, SoC designers can obtain several alternative implementations by applying a variety of "HLS knobs" such as: loop unrolling, loop pipelining, resource sharing, etc. These implementations form a Pareto-optimal set of choices in a multi-objective optimization space that represent design trade-offs in terms of performance metrics versus area/power costs.

DAC '14, June 01 - 05 2014, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2593069.2593071.

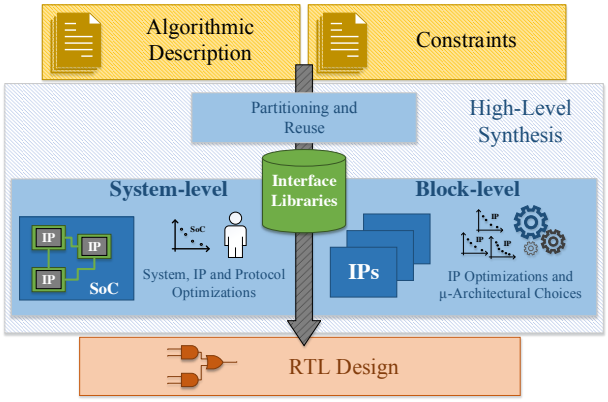


Fig. 1: ESL design flow for SoC accelerators.

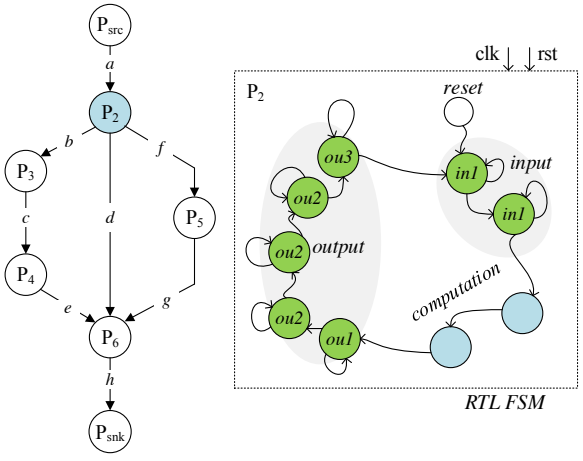
State-of-the-art HLS tools offer excellent *intra-process* optimization but very limited support for *inter-process* optimization. Hence, choosing the best implementation of each component for a given SoC and combining these implementations into an optimal system design are still manual, time-consuming tasks. To assist SoC designers in this effort, CAD-tool vendors provide libraries of *interface primitives*. These offer an application programming interface (API) to specify communication and synchronization among processes at the system-level as well as synthesizable implementations that can be combined with the implementation of each process in a modular fashion [4, 7, 16]. Examples of these primitives are the *blocking read/write commands* to receive/send messages in a synchronized way on a point-to-point channel between two processes. By encapsulating low-level signals, absorbing the timing differences across processes, and providing pre-designed implementations, these interface libraries support Transaction-Level Modeling (TLM) [8] and relieve SoC designers from the tedious task of creating a communication protocol.

On the other hand, the lack of automated tools for performance analysis and optimization at the system level makes the use of these libraries very challenging. The larger the number of components the harder the challenge. In particular, since the interface primitives are called within separate SystemC processes, they introduce *serialization* into the data transfers across the corresponding components. This may cause performance loss and, in the worst case, system deadlock. We address this problem by making three contributions: (1) a formal model to capture the impact of the interface primitives on system-level performance; (2) an efficient algorithm that, for a given system design, optimizes the use of these primitives in each process while ensuring absence of deadlock; and (3) a CAD tool to optimize the implementation of the computation part within each process and the inter-process communication channels.

Combined these contributions support a novel design methodology that enables compositional HLS and efficient system-level design-space exploration, thus improving ESL design productivity.

2. MOTIVATING EXAMPLE

The graph of Fig. 2(a) models a simple system consisting of five processes (represented by vertices $P_2 \ldots P_6$) communicating via eight point-to-point unidirectional channels (represented by arcs



a. b.

Fig. 2: A motivating example.

$a \ldots h$.) When specifying this kind of systems, designers develop also a testbench that captures the environment in which they operate: this is modeled by vertices P_{src} and P_{snk}, which produce and consume data for the system under design, respectively.

A portion of the synthesizable SystemC code of process P_2 is reported in Listing 1. The process behavior is specified in an untimed or loosely-timed TLM style [15] following a common structure: after a reset phase (lines 13-14), there is the main *infinite loop* (lines 15-27). Each loop iteration consists of three phases: input reading (lines 16-18), computation (line 20, hereby omitted), and output writing (lines 22-26). In particular, during the input and output phases the process communicates with other processes using the interface primitives. With the `get` primitive it acquires input data that are stored in a local memory array (lines 16-18). With the `put` primitives it emits output data on channels b, d and f, respectively. The process executes continuously the sequence of three phases unless suspended due to either an explicit `wait()` statement in the computation phase or *implicit waits* present in the implementation of the interface primitives, as explained next.

The interface primitives used in this example implement a *blocking communication protocol* similar to the one described in [4]: a `put` in a process (e.g. `ou1.put` in P_2, line 23) is associated to a corresponding `get` in the communicating process at the other hand of a channel (e.g. a `get` in P_3 on channel b); only when both processes reach the corresponding statements the transfer of data can occur. Conversely, if a process reaches its statement before the other, it gets suspended until the other is also ready to communicate. The use of such blocking protocol is very common and all CAD vendors support it by providing API and implementation libraries for similar interface primitives[1] Notice that the logic to suspend the protocol is transparent to the user as it is specified within the implementation of the `get` and `put` primitives.

The use of predefined interfaces simplifies the compositional design and synthesis of SoCs but it is not risk free. Although EDA vendors guarantee the timing and functional correctness of the interface implementations, designers may accidentally introduce system-level bugs or performance degradation when assembling components through these primitives. Debugging these problems is time consuming as it may requires many simulations and repeated HLS tool runs.

The *serial nature* of the SystemC process induces an order not only among the three main phases of its behavior (input, computation, and output) but also on the execution of the primitive statements. For instance, process P_2 first sends data to P_3 on channel b with `ou1.put`, then to P_6 on d with `ou2.put`, and finally to P_5 on f with `ou3.put`. Similar orders are defined among multiple `get` statements. For instance, we could assume that the code of process P_6 was specified such that it first reads data from P_5, then from P_2, and finally from P_4. If this was the case, however, the combination of these specifications would lead to a major problem: deadlock! In fact, process P_2 is prevented from sending data to P_5 on channel f because it is suspended on its `ou2.put` on channel d waiting to communicate with P_6. But P_6 is suspended on its `get` on channel g waiting to read from P_5, which itself cannot reach the put statements in the output phase because is suspended on its `get` on channel f waiting for P_2.

[1] Other protocols are also provided. In this paper we focus on blocking protocols but our approach applies also to *non-blocking protocols* as described in [6].

Listing 1: Synthesizable SystemC code of process P_2.

```
1  #include <systemc.h>
2  #include <hls_interface_library.h>
3  SC_MODULE(P2) {
4    sc_in<bool> clk, rst;
5    b_get_socket<packet_t> in1;
6    b_put_socket<packet_t> ou1, ou2, ou3;
7    SC_CTOR(P2) {
8      SC_CTHREAD(p2, clk.pos());
9      reset_signal_is(rst, false);
10     //...
11   }
12   void p2(void) {
13     // reset ...
14     wait();
15     while(true) {
16       // input
17       for (int i = 0; i<2; i++)
18       { data_in1[t] = in1.get(); } //from P_{src}
19                                    //via channel a
20       // computation ...
21
22       // output
23       ou1.put(data_ou1); //to P_3 via channel b
24       for (int i = 0; i<3; i++)
25       { ou2.put(data_ou2[i]); } //to P_6 via channel d
26       ou3.put(data_ou3); //to P_5 via channel f
27     }
28   }
29 private:
30   packet_t data_ou1, data_ou3;
31   packet_t data_in1[2], data_ou2[3];
32 };
```

To avoid deadlock is necessary to properly reorder the interface-primitive statements within the processes. For instance, we could reorder the `put` statements of P_2 such that channel f is written before channel b, which in turn is written before channel d and, at the same time, reorder the `get` statements of P_6 such that channel e is read before g, which is read before d. But the task of detecting and removing deadlock by code inspection becomes much harder with the complexity of the design. In the simple example of Fig. 2(a) there are already 36 possible order combinations. More generally, this number grows as: $\prod_{p \in \mathcal{P}} \left((|in\_chan(p)|)! \times (|out\_chan(p)|)! \right)$, where P is the set of processes.

Further, not all deadlock-avoiding orders are the same in terms of system performance: e.g. the second order given above avoids deadlock but yields a design with a suboptimal data-processing throughput because it forces a serial execution among processes that could run concurrently. One may think that such serialization would only impact the simulation speed and not the final implementation because, after all, hardware is inherently parallel. But this is not the case. The *serial nature* of the SystemC process in combination with the use of interface libraries and HLS may impact negatively the performance of the final hardware implementation.

Example. Consider the finite state machine (FSM) of Fig. 2(b) that is generated by a commercial HLS tool as part of the synthesis of the RTL implementation (Verilog code) for process P_2 of Listing 1. This FSM is the result of combining the implementation of the blocking primitives for the I/O phases of P_2 from the interface library (where the protocol is already implemented in a cycle-accurate manner) with the HLS of the computation phase. The resulting hardware circuit also iterates among the input, com-

putation, and output phases. There are as many I/O states as the number of `get/put` statements to be executed in the original process. The self-loop in each of these state allows the circuit to stall for multiple clock cycles when it must wait on a given channel for the circuit implementing the corresponding process to be ready for a data transfer. The chains of input and output states are separated by a chain of computation states. The length of this chain depends on the micro-architecture obtained through HLS: the more parallel is the micro-architecture, the shorter is the chain of computation states (but, generally, the more costly is the circuit implementation in terms of area and power.) A similar FSM is generated for each of the processes of Fig. 2(a). While each FSM enforces a serial progress of the computation of its circuit, naturally all hardware circuits can progress in parallel. Still, *the degree of parallelism in the operation of the RTL implementation corresponds directly to the degree of concurrency in the execution of the corresponding SystemC specification.* At the core of this correspondence there is the relative ordering of the I/O primitive statements. In particular, a shrewd order reduces the number of clock cycles that a component circuit spends waiting for a successful communication. This order, however, depends also on the micro-architectural choices made during the HLS synthesis of the computation part of each process. □

Hence, in addition to the question of how to avoid deadlock efficiently, there is a second question: is it possible to jointly optimize the intra-process computations and the inter-process communications in order to maximize the system performance? The rest of the paper will answer positively to both questions.

3. PERFORMANCE ANALYSIS

The previous section showed how the main infinite loop in the synthesizable SystemC process P_2 translates into the cyclic-structure FSM of Fig 2(b), which controls the hardware circuit obtained from P_2 through HLS. A similar FSM is synthesized as part of the implementation of each other process in the system. Since pair of processes communicate via a blocking interface, the implementation of the channel connecting the corresponding pair of circuits translates also in a cyclic structure, with *request/acknowledge* signals controlling the synchronized data transfers. In summary, the resulting hardware implementation of the overall system is characterized by a set of cyclic control structures, which have various lengths and interact in a way that is determined by the topology of the system specification. The performance of such deterministic concurrent system can be formally modeled using Timed Marked Graphs. This is a sub-class of Petri Nets can efficiently model concurrent systems, particularly those with periodic behavior [3, 14].

Definition 1. A timed marked graph is a graph defined as a 5-tuple $\mathcal{G} = (P, T, F, d, M_0)$, where P is a finite set of places, T is a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $d : T \to \mathbb{N}^+$ is a timing function, and $M_0 : P \to \mathbb{N}^+$ is the initial marking, and such that $P \cap T = \varnothing \wedge P \cup T = \varnothing$ and $\forall p \in P.(|\{t.(t,p) \in F\}| = |t.(p,t) \in F| = 1)$.

In other words, a TMG is a bipartite directed graph with two kinds of vertices (places and transitions), where each place has exactly one incoming edge and one outgoing edge; the timing function associates a delay to each transition; places can hold zero or more tokens; transitions cannot hold tokens, but they can fire; the initial marking specifies how many tokens each place holds before any firing. A *firing* creates a new marking by moving tokens around in the graph. A transition is enabled to fire when the place on each of its incoming edges has at least one token. When a transition fires, it takes a token from each of its incoming places and puts a new token into each of its outgoing places. While the firing activity may change the overall number of tokens in a TMG $\mathcal{G}$, the number $M_0(c)$ of tokens that are present on a cycle c of $\mathcal{G}$ is *invariant* under any firing sequence.

Definition 2. The cycle time $\pi(t)$ of a transition t of a TMG $\mathcal{G}$ is the average time separation between two consecutive firings of t and its reciprocal gives the average firing rate of t.

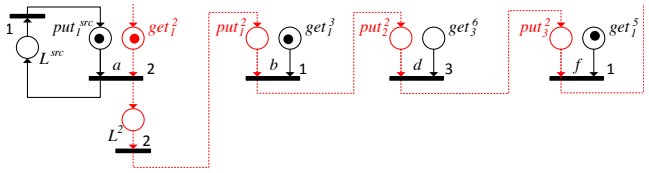


Fig. 3: The TMG model of P_2 in Listing 1 and Fig. 2.

If $\mathcal{G}$ is *strongly connected*, then a firing sequence eventually leads $\mathcal{G}$ back to the initial marking M_0 after firing every transition an equal number of times. All transitions of a strongly-connected TMG $\mathcal{G}$ have the *same cycle time*, which is called the *cycle time* of $\mathcal{G}$ and denoted as $\pi(\mathcal{G})$. This is a natural performance metric for the system modeled by $\mathcal{G}$ because its reciprocal is the rate of consumption/production of tokens, i.e., the *throughput* of the system.

Definition 3. The cycle mean of cycle c of TMG $\mathcal{G}$ is the ratio of the number $M_0(c)$ of tokens that are present on c divided by the sum of the delays of its transitions, i.e., $\mu(c) = \frac{M_0(c)}{\sum_{t \in c} d(t)}$.

The cycle time $\pi(\mathcal{G})$ of $\mathcal{G}$ is equal to the reciprocal of the minimum cycle mean across all cycles in $\mathcal{G}$. A cycle c with $\mu(c) = \pi(\mathcal{G})$ is *critical cycles*. Calculating the minimal cycle mean and identifying the critical cycles by the Definition 3 is impractical, since it requires the enumeration of all the elementary cycles of $\mathcal{G}$. More efficient methods exist, based on linear programming [12] or graph theory [5]. For our purposes we adopted Howard's algorithm [2], a polynomial-time algorithm in the size of the problem that is well-known to the stochastic-control community [5].

We developed a TMG model that allows us to complete an efficient perform analysis of the systems introduced in Section 2 without the need of time-consuming simulation. We present the model for the case of blocking primitives but it can be applied also to other primitives such as non-blocking [2]. The computation phase of a process is modeled by a single place connected to a transition whose time delay is equal to the latency of the micro-architecture implementation obtained through HLS. For the I/O phases, each channel used by the process is modeled by two places (a *put-place* and a *get-place*) that feed the same *channel transition*, whose time delay is equal to the minimum latency of the channel.

Example. Fig. 3 shows the portion [3] of TMG associated with process P_2 in Listing 1 and Fig. 2. The serial nature of the process translates into a chain of transitions in the TMG: the transition associated to channel a is followed by transition L^2 that models the computation phase of P_2 and then by three transitions, for channels b, d and f, respectively. Each channel transition is fed by a place modeling the get (put) statement in P_2 and a place modeling the put (get) statement in the corresponding process on that channel. For instance, the transition for channel b is fed by a put-place for P_2 and a get-place that is part of the model of P_3. □

Since the process iteratively executes the three phases, the first read operation, e.g., get_1^2 follows the last write operation, e.g., put_3^2. For the initial marking, a token is placed in the *first* get-place of each process, e.g., get_1^2 to model that its behavior starts with the first read operation. Also, a token is placed on the put-place of the test-bench process (e.g., put_1^{src}) to model the behavior of an environment that is always ready to provide new input data: when transition a fires a new token will be available (after some latency) in put_1^{src}.

4. CHANNEL ORDERING

In Section 2 we explained how the system performance depends both on the micro-architecture implementing the computation phase of each process and the relative ordering of the `put` and `get` statements in their I/O phases. The graph of Fig. 4(a) represents the

[2] The model for the non-blocking case is given in [6].

[3] The complete TMG for the system is reported in [6].

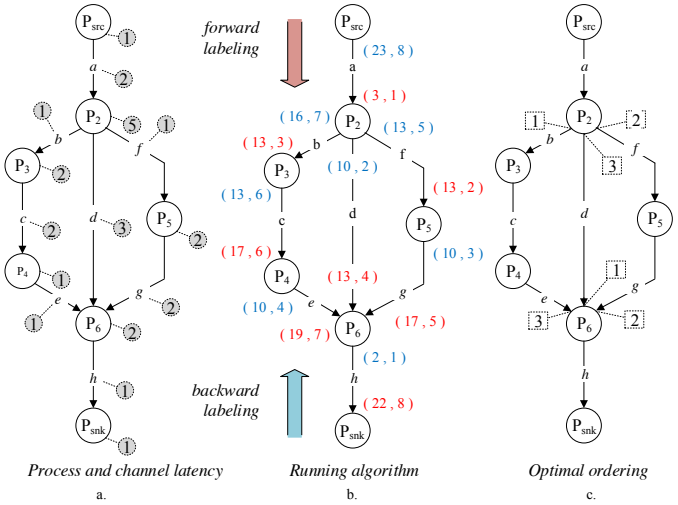


Fig. 4: Finding the optimal channel ordering.

result of running HLS on the system of graph Fig. 2: each vertex is now annotated with a number representing the latency of the process computation, which depends on the synthesized microarchitecture; each arc is annotated with the *minimum* latency required to complete the transfer of one data item on the corresponding channel [4] With these latencies values, the suboptimal ordering discussed in Section 2 yields a data-processing throughput equal to 0.05. This is the reciprocal of a cycle time equal to 20, which can be computed with the method presented in Section 3. However, the optimum cycle time is actually 12, i.e. 40% better. This is obtained by ordering the `put` statement of P_2 to access the channels as (b, d, f) and the `get` statements of P_6 to access the channels (d, g, e) (see Fig. 4(c).) Algorithm 1 finds this optimum ordering while guaranteeing also absence of deadlock.

Algorithm Description. The algorithm is a sequence of three main steps (lines 1-5): Forward Labeling, Backward Labeling and Final Ordering.

Forward Labeling performs a modified depth-first traverse of the system graph starting from the source T_{src}, while using a queue to track the vertex to be considered next (line 10-11). While visiting a vertex, each of its outgoing arcs $e = (x, y)$ is considered (line 13) following any order among its `put` statements (this could be an order given by the designer or the suboptimal of Section 2.) The head of arc e is labeled with (w, t), where $w \in \mathbb{N}^+$ is a *weight* and $t \in \mathbb{N}^+$ is a *timestamp*. The weight is equal to the sum of three values: the maximum among the weights associated with the arcs entering x, the sum of the latency of each arc leaving x, and the latency of x (line 17). The assigned timestamp is a global progressive number for the forward traversal. Finally, if arc e is the last arc visiting y then y is put in the queue.

Example. When Forward Labeling reaches P_2, the outgoing arcs are considered in the order f, b, d. Hence, the corresponding timestamps are 2, 3, and 4. The weight associated to these arcs is 13, i.e., the sum of $\texttt{MaxInArcWeight}(P2) = 3$, $\texttt{SumOutArcLatency}(P2) = 5$, and $\texttt{GetVertexLatency}(P2) = 5$. See the red labels marking the arc heads in Fig. 4(b). □

Backward Labeling is a similar procedure and is not reported here for lack of space [5] The procedure starts from the sink vertex T_{snk} and traverses the graph backward. The key difference is that when a vertex is visited, its incoming arcs are considered following the increasing order of the timestamps which have been assigned to the arc heads during Forward Labeling. Also, the weight is computed as the sum of: the maximum value of the weights associated with the arcs leaving x, the sum of the latency of each arc entering x, and the latency of x. The assigned timestamp is a global progressive number for the backward traversal.

Example. When the backward-labeling procedure reaches P_6, its incoming arcs are considered following the order d, g, e. The weight associated to these arcs is 10, i.e., the sum of $\texttt{MaxOutArcWeight}(P_6) = 2$, $\texttt{SumInArcLatency}(P_6) = 6$, and $\texttt{GetVertexLatency}(P_6) = 2$. See the blue labels marking the arc tails in Fig. 4(b). □

Final Ordering sorts the `get` statements of each process according the *ascending* values of the head weights (line 28) and the `put` statements according to *descending* values of the tail weights (line 32). In both cases, ties among the weight values are broken according the ascending values of the timestamps: this tie-break is necessary to avoid certain deadlock situations, which may occur in graphs with some symmetric structures.

Example. Since the head weights of arcs e, d, g are $19, 13, 17$, process P_6 read first from channel d, then g, and finally e. Also, since the tail weights of arcs b, d, f, are $16, 10, 13$, process P_2 writes first channel b, then f and finally d. □

The basic idea of the algorithm is to sort the chain of `put` statements in each process by giving priority to those statements that *start a path* whose aggregate latency is longer than the others and to sort the chain of `get` statements in each process by giving priority to those that *end a path* whose aggregate latency is shorter than the others. The result of this choices is an optimization of the system performance.

The overall complexity of the algorithm is $O(n\ log(n))$ in the size of the graph. It requires two depth-first visits that take time $O(|E|)$ and space $O(|V|)$ in the worst case to store the verices in the queue. The sorting of the arcs takes $O(|E|\ log(|E|))$, which dominates the overall complexity.

Algorithm 1: Channel ordering algorithm.

```
1  Procedure ChannelOrdering(G)
       Data: G = ⟨V, E⟩ is a direct graph
       Result: G' where the arcs have been ordered
2      G' ← ForwardLabeling(G)
3      G'' ← BackwardLabeling(G')
4      G''' ← FinalOrdering(G'')
5      return G'''
6  Procedure ForwardLabeling(G)
7      CreateEmptyQueue(queue)
8      SetAllNodesNotVisited(V)
9      timestamp ← 1
10     Enqueue(queue, SourceVertex)
11     while IsNotEmpty(queue) do
12         x ← Dequeue(queue)
13         foreach arc ∈ OutArcs(x) do            // arc is (x, y)
14             MarkAsVisited(y)
15             if LastVisitingArc(arc, y) then
16                 Enqueue(queue, y)
17             weight ← MaxInArcWeight(x) +
                        SumOutArcLatency(x) +
                        GetVertexLatency(x)
18             SetArcHeadWeight(arc, weight)
19             SetArcHeadTimeStamp(arc, timestamp)
20             timestamp ← timestamp + 1
21     return G
22 Procedure BackwardLabeling(G)
23     //omitted
24 Procedure FinalOrdering(G)
25     foreach x ∈ Vertices(G) do
26         order ← 1
27         foreach arc ∈ OrderByHeadWeight(InArcs(x)) do
28             SetHeadOrder(arc, order)
29             order ← order + 1
30         order ← 1
31         foreach arc ∈ OrderByTailWeight(OutArcs(x)) do
32             SetTailOrder(arc, order)
33             order ← order + 1
34     return G
```

[4] A data item can be decomposed in packets to be transferred through multiple `put`/`get` iterations.

[5] The complete algorithm is reported in [6].

5. DESIGN METHODOLOGY

In Section 4 we described how the algorithm for producing the optimal channel reordering is based on the process latencies, defined by the synthesized micro-architecture implementing their computation phase. Varying these latencies may result in a different ordering and, in turn, a different system-level performance. To explore the design space we developed the methodology based on the idea of separation of concerns [10] between computation (optimizing of process latencies) and communication (channel reordering). As shown in Fig. 5, the starting point is a system model with a set of Pareto-optimal μ-architectures that differ in terms of latency and area occupation. The designer can also specify the targets (e.g. the target cycle time T_{CT}) of the overall system. At each iteration, given the current cycle time (CT) and the associated critical cycle computed with the performance model presented in Section 3, the goal is to optimize the overall system to meet the given constraints by first exploring the available μ-architectures and then applying the channel ordering based on the resulting process latencies.

Given the target cycle time T_{CT}, we define the *performance slack* s_p as $s_p = T_{CT} - CT$. If $s_p > 0$, the constraint is met and we can thus perform area optimization (*area recovery*). If $s_p \leq 0$, we reduce the latency of the processes on the critical cycle (*timing optimization*).

Given the set of processes P and the implementations I, a binary variable $x_{i,p} \in P \times I$ denotes if the final system includes implementation i for process p. Each process must have one and only one implementation. The *latency gain* $l_{i,p}$ is the (positive or negative) difference between the current latency of process p and the latency of i. The *area gain* $a_{i,p}$ is the (positive or negative) difference between the current area of p and the area of i. These values represent the differences introduced by selecting implementation i instead of the current one for process p.

We define two problems.

Area recovery: given a system with $s_p > 0$, determine the IP configurations that minimize the area occupation without affecting the critical cycle (i.e., maintaining $CT < T_{CT}$).
This problem is a variant of the *knapsack problem* to maximize $\sum_{i,j} x_{i,j} * a_{i,j}$ while respecting constraint $\sum_{i,j} x_{i,j} * (-l_{i,j}) \leq s_p$ for the processes on the critical cycle. Since the implementations are Pareto optimal, moving towards a positive area gain corresponds to a negative latency gain and vice versa. ■

Timing optimization: given a system with $s_p < 0$, determine the IP configurations to minimize the difference $CT - T_{CT}$.
This problem corresponds to a classical optimization problem, where we aim at maximizing $\sum_{i,j} x_{i,j} * l_{i,j}$, i.e., the cumulative latency gain for all the processes j on the critical cycle. ■

These problems can be formulated as instances of Integer Linear Programming (ILP), with constraints to discard the configurations already optimized. The formulation with area constraints is the dual problem and is omitted due to lack of space.

6. EXPERIMENTAL RESULTS

We developed ERMES, a prototype CAD tool based on the proposed methodology. It uses the GLPK (GNU Linear Programming Kit) package to solve the ILP formulations. We applied ERMES to the case study of an MPEG-2 Encoder, which had been previously designed in our team [11]. This design was refactored so that each process is specified using a loosely-timed TLM style following the common structure described in Section 2. The resulting system-level specification consists of 26 processes described in synthesizable SystemC and interconnected through 60 blocking channels (Table 1). These processes are connected to two additional processes which act as a test-bench by providing the image streams and receiving the encoded images, respectively.

The characteristics of the MPEG-2 Encoder algorithm are good to highlight the main challenges of designing a complex SoC accelerator. In particular, its system-level block diagram contains structures that may cause deadlock or performance degradation if the communication protocol is not properly configured: these include recovergent paths (as for the example of Fig. 1) and feedback loops.

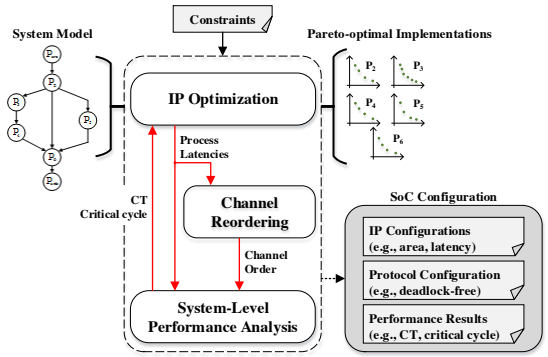


Fig. 5: Outline of the proposed methodology.

Table 1: Experimental setup of the MPEG-2 Encoder.

Processes	26	**Channels**	60
Lines of Code	~9,000	**Image Size (pixels)**	$352x240$
HLS knobs	loop pipelining, loop unrolling, ..	**Pareto points**	171
Technology	$45nm$	**Frequency**	1GHz

By applying the method for compositional HLS and design-space exploration proposed by Liu and Carloni [11], we derived many alternative Pareto-optimal μ-architectures for the inner computation core of each process. Since computation is not intertwined with communication, these characterizations are not affected by channel ordering and can be performed as a preprocessing step. We performed the characterization of the channel latencies based on the quantity of the data to be transferred and the physical constraints imposed by the HLS tool for the channels. These latencies range from 1 to 5,280 clock cycles and do not depend on channel ordering or the process implementations.

Using the approach proposed in [11], we obtained also a set of of Pareto-optimal implementations for the overall system. Notice that these implementations are optimal among all those that can be obtained without making any automatic modification to the SystemC code (including possible reordering of the interface-primitive statements within the processes.) Indeed, they are based on the choice of a conservative ordering that guarantees absence of deadlock but may introduce unnecessary serialization of processes that could run in parallel, similarly to the situation described in Section 2. Without the support of a tool like ERMES, it is difficult to go beyond such conservative ordering because there are simply too many possible ordering combinations to consider, with the additional risk of introducing a deadlock. Furthermore, the identification and correction of deadlock situation typically require multiple lengthy simulations of the entire SystemC design.

Instead, the applications of the proposed methodology and the ERMES tool allows us to perform richer design-space explorations and to obtain better implementations. To show its capabilities we report the results obtained starting from two implementations from the given set. Implementation $M1$ has the best performance as it features the fastest implementations for the computational part of each process: specifically, it delivers a CT of 1,906 KCycles with an area occupation of $2.267mm^2$. Implementation $M2$ is the result of trading-off performance for a smaller area occupation: it has a CT of 3,597 KCycles with an area of $1.562mm^2$.

When applied to implementation $M1$, ERMES is capable of detecting some unnecessary serialization of processes that could run in parallel. By reordering the interface primitives of some processes, it resolved this issue without making any change on their core computational parts. The result is **a 5% improvement of the CT without any increase in area occupation**.

When applied to implementation $M2$ with different input con-

straints, ERMES can perform automatically two different types of iterative design-space explorations. The left-hand side of Fig. 6 shows a timing-optimization exploration as the result of imposing a constraint on the target cycle time $T_{CT} = 2,000$ KCycles with the goal of substantially improving performance. The right-hand side of Fig. 6 shows an area-recovery exploration that is based on running ERMES with a much more relaxed constraint ($T_{CT} = 4,000$ KCycles) in order to reduce the area occupation. Even if they start from the same implementation $M2$, the two explorations present a different evolution due to the different input constraints.

In the first exploration, ERMES immediately generates a new implementation that meets the target cycle time while increasing the area occupation. Then, it tries unsuccessfully to reduce the area overhead, as the second implementation violates the given constraint. The situation is recovered in the third iteration and the last one confirms that no further changes can be applied. The final implementation gives a **speed-up of 2x** with respect to the initial one, with an **area overhead of 44.57%**.

In the second exploration the starting point already meets the target cycle time. Hence, ERMES turns to optimize the area. The first generated implementation gives a significant area reduction but violates the timing constraint. In the following iteration ERMES removes the timing violation without affecting the area occupation. After the last iteration, the resulting implementation yields an **area reduction of 32.46%** with respect to $M2$, in exchange for a **timing degradation of less than 1%**.

ERMES automatically executes very interesting optimizations. For example, during the third iteration of the first exploration, it improves the system performance by selecting much faster implementations for some of the processes on the critical cycle. The corresponding area overhead is recovered by selecting smaller implementations for other processes (including other processes on the critical cycle), provided that the cumulative balance of their latency gains remains positive. As a result, the cycle time is effectively reduced, with minor additional changes in the area occupation of the final design. Notice that, as it generates a new implementation, the algorithm for channel reordering optimizes the performance in almost all the cases, but the last ones where no changes are applied to the process latencies.

Analysis of scalability. Since the complexity of the proposed approach only depends on the system topology, we designed a set of synthetic SoC benchmarks to evaluate also its scalability. We generated graphs with up to 10,000 processes interconnected with 15,000 channels, along with a corresponding set of hypothetical μ-architectures. The resulting benchmarks have characteristics similar to those of the MPEG-2, including the presence feedback loops and reconvergent paths. The experimental results demonstrate that our approach scales well, as ERMES takes a time of the order of a few minutes in the worst cases [6].

7. RELATED WORK

The serialization of the communication that may arise with the use of commercial HLS tools is a problem that has received limited attention in the literature. One may think that this problem can be solved by breaking the channel operations in multiple concurrent processes. However, this often leads to inefficient designs because HLS tools create as many memory ports as the number of concurrent processes insisting on that memory and the memory size scales badly with the number of ports. Many designers thus prefer to reduce the number of processes and use a design style with three phases (*input reading*, *computation* and *output writing*) like the one assumed in Section 2. This simplifies system-level analysis and optimization and has been effectively adopted in many works based on synchronous dataflow and Kahn process networks (e.g. [9, 19]). These models of computation, however, lead to communication channels based on FIFOs, which must be carefully sized [10]. Also, they do not handle the serialization issues introduced by the interface primitives offered with modern HLS tools. Deadlock detection has been studied for blocking primitives in CSP systems [17], but the interactions and the ordering among the channels was not considered. Deadlock removal has been addressed in the context of SystemC simulation kernels [1], but no approaches were proposed for preventing a deadlock and optimizing the system performance in hardware designs described in SystemC.

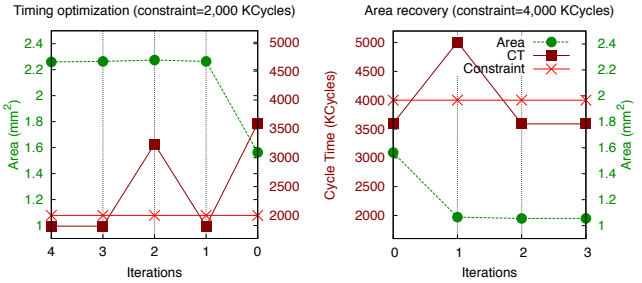


Fig. 6: Design-space exploration starting from *M2*.

8. CONCLUSIONS

We proposed a novel methodology and an associated CAD tool for supporting compositional HLS of complex SoC accelerators. It focuses on the integration of IP components through the use of the interface libraries offered by HLS commercial tools and co-optimizes the computation micro-architecture and the communication channels. The experimental results show the efficiency of our approach in the automatic optimization of a complex design. Future work will involve the co-optimization of the memory elements.

Acknowledgments. This work is partially supported by the STARnet Center for Future Architectures Research (C-FAR), the DARPA PERFECT program, and the NSF (A#:1219001).

9. REFERENCES

[1] C.-N. Chou et al. Formal deadlock checking on high-level systemc designs. In *Proc. of ICCAD*, pages 794–799, 2010.

[2] J. Cochet-Terrasson et al. Numerical computation of spectral elements in max-plus algebra. In *Proceedings of IFAC CSSC*, 1998.

[3] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(5):511–523, 1971.

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

[5] A. Dasdan, S. Irani, and R. K. Gupta. An experimental study of minimum mean cycle algorithms. Technical Report 98-32, UC Irvine, 1998.

[6] G. Di Guglielmo, C. Pilato, and L. P. Carloni. A design methodology for compositional high-level synthesis of communication-centric SoCs. Technical report, Dept. of Computer Science, Columbia University, New York, Apr. 2014.

[7] M. Fingeroff. *High-level synthesis blue book*. Mentor Graphics Corp., 2010.

[8] F. Ghenassia. *Transaction-Level Modeling with SystemC*. Springer-Verlag, 2006.

[9] K. Huang et al. Embedding formal performance analysis into the design cycle of mpsocs for real-time streaming applications. *ACM Trans. Embed. Comput. Syst.*, 11(1):8:1–8:23, 2012.

[10] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 19(12):1523–1543, 2006.

[11] 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, 2012.

[12] J. Magott. Performance evaluation of concurrent systems using petri nets. *Information Processing Letters*, 18(1):7–13, 1984.

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

[14] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[15] Open SystemC iv Initiative (OSCI). *OSCI TLM-2.0 Language Reference Manual*. 2009.

[16] J. Sanguinetti, M. Meredith, and S. Dart. Transaction-accurate interface scheduling in high-level synthesis. In *ESLsyn Conference*, pages 31–36, 2012.

[17] B. Shao, N. Vasudevan, and S. A. Edwards. Compositional deadlock detection for rendezvous communication. In *Proc. of EMSOFT*, pages 59–66, 2009.

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

[19] S. van Haastregt and B. Kienhuis. Automated synthesis of streaming C applications to process networks in hardware. In *Proc. of DATE*, pages 890–893, 2009.