High-Level Synthesis of Accelerators in Embedded Scalable Platforms

Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni
Department of Computer Science, Columbia University, New York, NY - USA
{paolo,giuseppe,luca}@cs.columbia.edu

Embedded scalable platforms combine a flexible socketed architecture for heterogeneous system-on-chip (SoC) design and a companion system-level design methodology. The architecture supports the rapid integration of processor cores with many specialized hardware accelerators. The methodology simplifies the design, integration, and programming of the heterogeneous components in the SoC. In particular, it raises the level of abstraction in the design process and guides designers in the application of high-level synthesis (HLS) tools. HLS enables a more efficient design of accelerators with a focus on their algorithmic properties, a broader exploration of their design space, and a more productive reuse across many different SoC projects.

I. Introduction

The system-on-chip (SoC) has emerged as the most important computing platform across many application domains from embedded systems to data centers [23, 25, 28]. To achieve energy-efficient high performance while facing the challenges of dark silicon [15], SoC architects realize heterogeneous architectures by combining a few processor cores with an increasing number of *accelerators* [33]. Heterogeneity, however, increases system complexity, reduces architecture regularity, and prolongs design time.

The *reuse* of intellectual property (IP) blocks, including specialized hardware accelerators, is one of the most promising strategy to mitigate the SoC design complexity [30]. To create and maintain reusable IP blocks, however, is estimated to be 2 to 5 times harder than their creation for one-time use [1]. But the biggest challenges are actually in their integration into the SoC, a manual process that involves both hardware and software aspects and requires to deal with many interfaces and protocols. This leads to very long design cycles and high costs (estimates are that $50M is required to get a complex SoC to first prototype [19]), which ultimately reduce the appealing of starting new entrepreneurial efforts and seeking innovation in hardware design. CAD researchers must raise the level of abstraction above register-transfer level (RTL) [6, 12, 31] and make *system-level design (SLD)* become a commercial success.

Motivated by these challenges, in the System-Level Design group at Columbia University we have been developing the concept of *Embedded Scalable Platforms* (ESP), which brings together a new platform architecture with a companion SLD methodology. The architecture addresses the complexity of IP-block integration by balancing hardware specialization and design regularity with a tile-based approach. The methodology seeks to increase productivity by moving the bulk of the engineering effort to the system level and reducing the gap between hardware design and software programming.

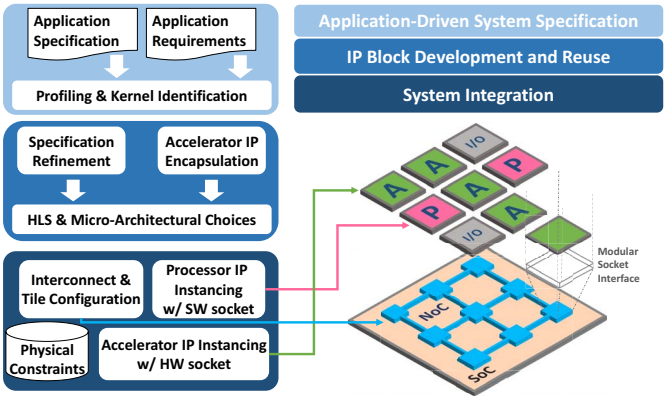

Fig. 1. The SLD methodology for embedded scalable platforms.

Fig. 1 illustrates the relation between the methodology and the architecture. An SoC is an instance of an ESP architecture that is obtained by specifying a mix of tiles. Each tile may implement a processor, a hardware accelerator, or some auxiliary functionality like I/O access. The number and mix of tiles of a particular ESP instance depends on its target application domain. The choice of a specific tile combination is the result of an application-driven *design-space exploration (DSE)* that is guided by our methodology, which is supported by a combination of state-of-the-art commercial CAD tools and in-house developed tools [14, 22, 26]. The premise of our approach is that the target workloads must drive both the software-programming and hardware-design efforts throughout all stages of the system realization.

The system specification involves the definition of the application requirements and the development of the application software. Software profiling identifies those critical computational kernels that deserve to be implemented in hardware. The corresponding accelerators can be either developed from scratch or acquired as reusable IP blocks. Critically, for both cases our methodology advocates and promotes the use of *high-level programming languages* such as SystemC [5] and *high-level synthesis (HLS) tools* [9, 11, 17] to design parameterized IP blocks and provide a richer spectrum of power/performance tradeoff points. This augments reusability because architects of

978-1-4673-9569-4/16/$31.00 ©2016 IEEE

very different SoCs can synthesize those IP-block implementations that are more suitable to their purposes.

At the level of individual IP block, the task of DSE consists in deriving a set of alternative implementations, each offering a particular cost-performance tradeoff. In this contest, the *Pareto curve* (or Pareto set) represents the set of implementations that are Pareto optimal: i.e. no higher performance implementation exists for the given cost [32, 36]. The broader are the cost and performance ranges spanned by the Pareto curve, the higher is the reusability of the IP block. At the system level, the task of DSE is inherently a component-based design effort, as the choice of a particular RTL implementation for a module must be made in the context of the choices for all the other modules that are also components of the given system. A particular set of choices leads to a point in the multi-objective design space for the whole SoC. So, the process of deriving the diagram of Pareto-optimal points repeats itself hierarchically at the system level [22].

Our methodology mitigates the complexity of integrating heterogeneous components by providing a regular but flexible socket-based template and a set of *platform services*, including: accelerator reservation and configuration, data transfers, performance counters, and diagnostics. In particular, the accelerator tiles contain high-throughput accelerators that are loosely coupled with the processor: each accelerator typically works on large (e.g. $300MB$) data sets by leveraging a private local memory that is tailored to its specific needs and exchanges data with main memory through a private DMA controller [10].

The platform services are supported by: (1) a *scalable communication and control infrastructure*; (2) a set of *configurable hardware sockets* that interface the components to the interconnect and are designed for modularity and flexibility by following the Protocols and Shells Paradigm of latency-insensitive design [7, 8]; and (3) a set of *software sockets* that convey the illusion of a simpler homogeneous architecture to the programmer. The interconnect can be realized either with a bus or a network-on-chip (NoC), depending on the needs in terms of bandwidth and platform services. Designs that have a larger number of components typically rely on an NoC, which can be scaled up by adding more virtual channels or physical planes [35].

The system-integration phase is completed by the validation of the given ESP instance through its emulation with an FPGA board. This prototyping effort is strongly simplified by the adoption of SLD methods. For example, HLS tools provide an immediate way to re-target an accelerator implementation from an ASIC to an FPGA technology. FPGA emulation is critical not only to validate the correctness of the design but also to obtain an accurate analysis of its performance. For instance, as illustrated later in the paper, it allows us to assess how the performance of a given hardware accelerator is affected by its interaction with all the other system components.

TABLE I
CHARACTERISTICS OF THE WAMI-APP SOURCE CODE.

FUNCTION	LOC	IF	LOOP	ASSIGN	FCALL	ARROP
Debayer	195	4	24	70	12	56
Grayscale	21	2	2	8	0	4
Warp	88	12	0	51	3	11
Gradient	65	7	4	34	0	54
Subtract	36	7	2	13	0	3
Steep.-Descent	34	0	3	21	0	3
SD-Update	55	9	4	20	0	5
Hessian	43	0	6	18	0	4
Matrix-Invert	166	33	8	59	8	20
Matrix-Mult	55	7	5	20	0	5
Reshape	42	11	1	15	0	2
Matrix-Add	36	7	2	13	0	3
Change-Detect.	128	12	9	62	3	41
Total	964	111	70	404	26	211

II. HLS-BASED ACCELERATOR DESIGN

C and C++ languages are popular input formats for HLS tools for several reasons: a large number of existing algorithms are written in these languages; they facilitate hardware/software co-design since most embedded software is written in C; and C-level functional execution is much faster than RTL simulation. Over the years, however, they have also manifested some proven limitations because they inhibit the specification, or the automatic inference, of important properties of hardware systems related to concurrency, timing, data formatting, and communication. Many C/C++ applications are sequential programs written with a focus on software performance without any intent to map them on specialized hardware. Applications written to exploit multi-core architectures use multi-threading libraries that make the code difficult to be refined for HLS tools.

SystemC is an IEEE-standard language that has been developed to overcome these limitations and that has become a *de-facto* standard for HLS. The task of porting an application specification (or a portion of it) to SystemC to enable the HLS of specialized accelerator may be time consuming. A well-structured C/C++ code that partitions the specification into functional blocks helps the engineer who performs this task. The goal is not only to obtain any SystemC specification that can be synthesized by the given HLS tool but to do so in a way that enables the exploration of a broad design space by evaluating many architectural and micro-architectural optimization choices. To illustrate some of the aspects of this process we present the design of an ESP instance implementing WAMI-App, an accelerated version of the Wide-Area Motion Imagery (WAMI) application.

A case study: WAMI-App. WAMI is an image processing application used for aerial surveillance [27]. It processes a sequence of input frames to extract masks of "meaningfully changed" pixels. For example, it can detect and track vehicles moving on the ground, while discarding environmental noises, e.g. shadows, surface reflections, etc. As a starting point for our case study, we use the WAMI specification that is available in the PER-

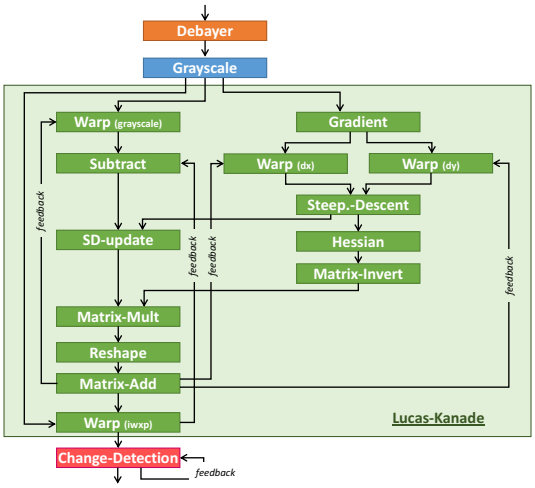

Fig. 2. Block diagram of the WAMI-App.

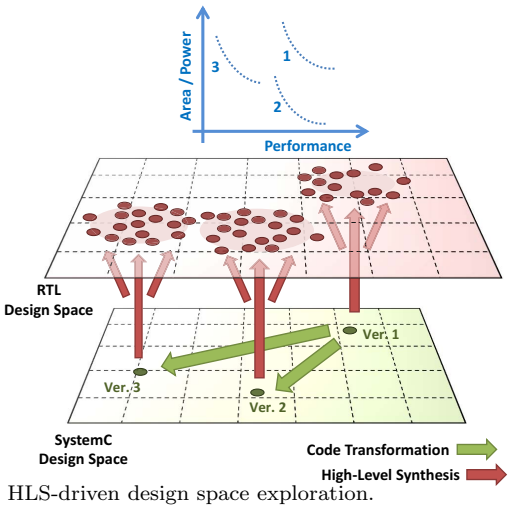

Fig. 3. HLS-driven design space exploration.

FECT Benchmark Suite [4], together with a testbench and some input sensory data. This specification is written using a subset of the C language: e.g, it has a limited use of pointers and does not use dynamic memory allocation and recursion; also, with the exception of few mathematical functions (e.g. `exp()`, `sqrt()`, etc.), no external library functions are called. This simplifies the task of porting this specification from C into a subset of SystemC that can be synthesized effectively with HLS tools. Table I reports some of the characteristics of the WAMI specification. Specifically, for each main (FUNCTION), it reports the number of: lines of C code (LOC), conditional statements (IF), loops (LOOP), assignments (ASSIGN), calls to functions, including both functions provided with the source code and functions from the `C math` library (FCALL), and read/write operations on arrays (ARROP).

Code Partitioning and Potential Parallelism. The WAMI specification in C consists of four main computational kernels: the DEBAYER filter, the RGB-TO-GRAYSCALE conversion, the LUKAS-KANADE image alignment, and the CHANGE-DETECTION classifier. As we ported the WAMI code into SystemC, we partitioned the LUKAS-KANADE kernels into nine SystemC processes to have the option of synthesizing an accelerator for each of them, thereby increasing parallelism at the architecture level. The block diagram of Fig. 2 highlights the data-dependency relations among the WAMI kernels and the *potential* for parallel execution: e.g., MULT must run after SD-UPDATE and INVERT-GJ, which instead can run concurrently. The data-dependency relations apply to the processing of a single input frame. Overlapping the processing of multiple frames in a pipeline fashion allows more accelerators to execute in parallel.

Refinement and Micro-Architectural Choices. Fig. 3 shows the relationship between the SystemC design space and the RTL design space. HLS tools provide a rich set of configuration knobs [9, 11, 17] that can be applied to synthesize a variety of RTL implementations. These implementations are based on different micro-architectures and provide different cost-performance tradeoffs points. The micro-architectural knobs are "push-button" directives of the HLS tool represented by the red arrows in Fig. 3. In addition, the engineer may perform manual transformations (represented as green arrows) to obtain revised versions of the SystemC specification: these transformations preserve the functional behavior but extend the RTL design-space exploration. For example, they can expose parallelism, remove false dependencies, increase resource sharing etc. In this way, the engineer may either reduce the area/power (*ver.* $1 \rightarrow 2$) or improve the performance (*ver.* $1 \rightarrow 3$).

Example. A simple HLS directive is function inlining. *It allows HLS to further optimize the body of the function in the context of the caller and it removes performance degradation due to inter-module communication; but the complexity of the synthesized hardware may increase due to resource replications. Function-inlining knob is provided with most HLS tools. In contrast,* function encapsulation *is a manual code transformation that identifies frequent patterns in the C-like implementation and encapsulates them with new functions. Fig. 4 shows a portion of the* GRADIENT *kernel of WAMI-App: the engineer may be able to identify the pattern* `(a - b) >>1` *in the before-transformation code and encapsulate it in the function* `cntrl_diff()`. *This function is then used at any occurrence of the pattern. Function encapsulation allows HLS to reduce the large number of states in the main module that may produce an inefficient circuit with a long critical path delay due to the complicated control; in addition, the body of the functions can be highly optimized and reused.* □

ESP Accelerator Model. With ESP, we propose a model for the accelerators that is *loosely-coupled* with the processor [10]. The accelerator is located outside the processor core and interacts with it and the off-chip memory via DMA through the on-chip interconnect. We came to define this model after implementing many different accelerators for high-performance embedded application kernels. Each kernel has distinctive characteristics that influence the design of the corresponding accelerator. These include the degree and granularity of computational par-

```
/* ******************** */
/* before transformation */
/* ******************** */
for (y = 1; y < nRows - 1; y++) {
  for (x = 1; x < nCols - 1; x++) {
    z = y * nCols + x;
    Xgrad[z] = (  Iin[y*nCols + (x+1)] - Iin[y*nCols + (x-1)]) >>1;
    Ygrad[z] = (- Iin[(y-1)*nCols + x] + Iin[(y+1)*nCols + x]) >>1;
  }
}

/* ******************** */
/* after transformation */
/* ******************** */
float cntrl_diff(float a, float b) {
  return ((a - b) >>1);
}

for (y = 1; y < nRows - 1; y++) {
  for (x = 1; x < nCols - 1; x++) {
    z = y * nCols + x;
    Xgrad[z] = cntrl_diff(Iin[y*nCols + (x+1)], Iin[y*nCols + (x-1)]);
    Ygrad[z] = cntrl_diff(Iin[(y+1)*nCols + x], Iin[(y-1)*nCols + x]);
  }
}
```

Fig. 4. Code transformation by function encapsulation.

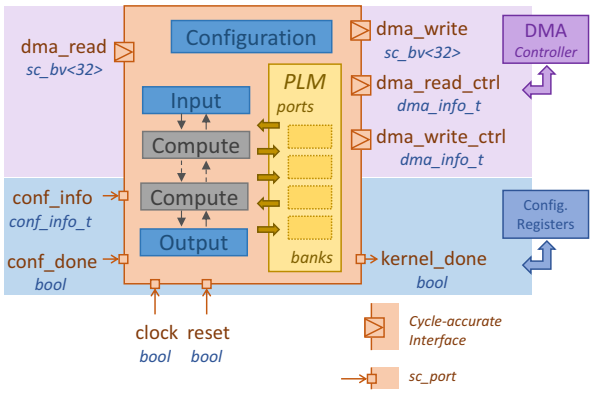

Fig. 5. ESP Model for accelerator design and integration.

allelism, the ratio of computation versus communication with main memory, and the memory access patterns to read and write data. For instance, Debayer has a simple-strided pattern, Warp has a data-dependent pattern, while Gradient has a sequential-access pattern. Despite these differences, however, we identified a main structure in the behavior of loosely-coupled accelerators that allows us to define our model and develop a configurable interface that can be applied effectively to synthesize many kernels.

As shown in Fig. 5, the accelerator behavior is organized in four main phases. First, there is the *configuration* that entails the interaction between software and hardware. By accessing state and command registers, a device driver checks the status of the accelerator, configures it, and starts a new execution. The configuration phase takes a negligible time with respect to the others. When invoked, the accelerator iterates over three main phases: *input*, *computation* and *output*. These repeat for portions or chunks of data until the entire input is processed. During the input phase, the accelerator issues a DMA request for a chunk of data from the main memory. Such request is autonomous and not controlled by the processor. The accelerator transfers the chunk of data from the main memory to a properly-sized *private local memory (PLM)* using transaction-level modeling (TLM) primitives. When data are available in the PLM the accelerator performs the actual computation specified by the synthesized functionality. Finally, upon completion, an autonomous write request is issued to store the results back into main memory.

Component-Based Design-Space Exploration. By leveraging a mix of state-of-the-art commercial CAD tools and in-house developed tools [14, 22, 26], we performed the HLS-driven DSE of Fig. 3 on twelve kernels of WAMI-App. For this case study, we targeted an industrial 32*nm* ASIC technology. Fig. 6 reports the DSE result as Pareto sets for four representative accelerators. For each synthesized implementation, we plot the area and *effective latency*, which is the product of the clock cycle count times the clock period.

For example, we obtained eight Pareto-optimal RTL implementations of the Debayer accelerator. Point *d1* in Fig. 6 is synthesized from a baseline synthesizable SystemC specification that is *not* partitioned according to the model of Fig. 5. Then, in order to achieve higher throughput, we synthesized the input and output processes to transfer data with the off-chip main memory and implemented the PLM as a circular data buffer. Circular-buffering improves throughput by overlapping in time computation and communication. The resulting RTL implementation (*d2*) has a better effective latency (9% improvement) with respect to *d1* but a slightly bigger area (0.5%) due to the larger memory of the circular buffer.

To generate other Pareto-optimal implementations, we focused on the Debayer memory access pattern. The Debayer algorithm interpolates pixels row by row and uses 5×5-interpolation masks centered on the pixel of interest. In particular, the input process pre-fetches at least 5 rows of the image and stores them in the PLM; from this, the computation reads 9 to 11 neighbouring pixels to interpolate a single pixel. For the implementations *d1* and *d2*, we designed the PLM using traditional vendor memories provided by the adopted HLS tool that support at most two read ports. To improve the performance of our accelerators, we used our memory compiler to generate a customized PLM [26] that provides more ports, thus supporting many concurrent read/write operations. Correspondingly, we instructed the HLS to leverage the availability of a higher number of ports during the scheduling phase. By progressively increasing the number of read ports, we obtained six new distinct implementations (*d3-d8*): with *11* read ports, *d8* gives an 3.8X improvement of effective latency with respect to the baseline, in exchange for a 3.4X area increase (due to the complexity of the memory-arbitration logic).

We performed similar HLS-driven DSE for Gradient and Warp to obtain four and three Pareto-optimal implementations, respectively. Here, increasing the number of ports does not necessarily lead to an optimal implementation (e.g. *g5* and *w4*) because a higher number of ports requires a more complex datapath and control logic that may increase the critical path.

For Change-Detection, the Pareto-optimal implementations *c1-c4* are the result of combining manual code refac-

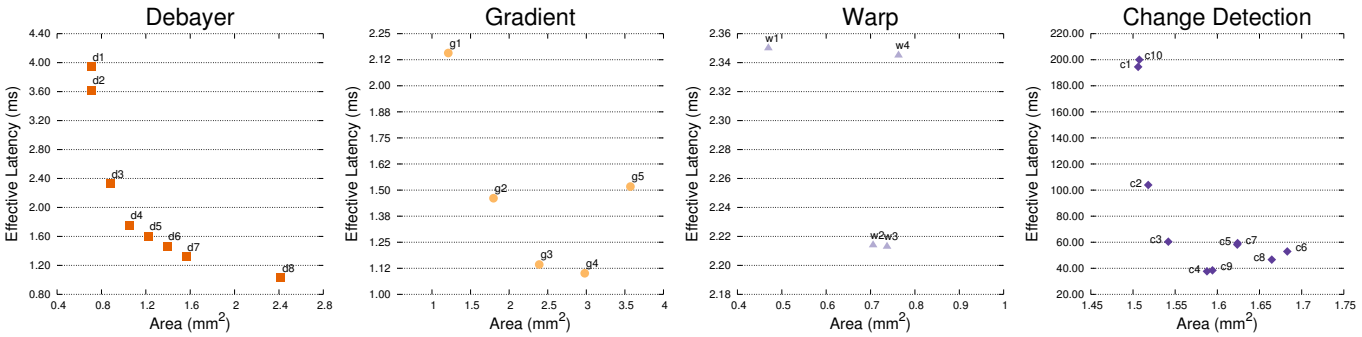

Fig. 6. IPs Pareto curves for DEBAYER, GRADIENT, WARP and CHANGE-DETECTION.

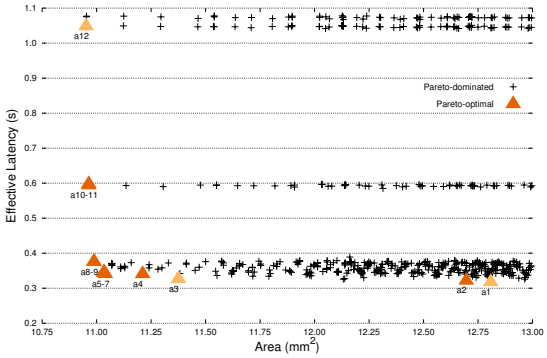

Fig. 7. Compositional DSE results for WAMI-APP.

toring aimed at decoupling the access to the PLM (i.e. reference to large arrays and data structures in the C specification) from computation with the aggressive application of HLS-knobs (e.g., the *c4* is obtained with deep partial-unrolling of some computational intensive loops); RTL implementations *c5-c9*, obtained with partial-unrolling combined with function inlining, are suboptimal due to the reduced resource sharing.

We performed *compositional* DSE on the whole WAMI-App by combining the data-dependency graph of Fig. 2 with the synthesis results for each accelerator. Fig. 7 shows the combined latency/area trade-off points, including twelve Pareto-optimal points. Notice that the fastest system implementation (*a1*) is not the result of composing the fastest implementation for each accelerator; instead, it is obtained by composing *d1* (smallest), *g1* (smallest), *w3* (fastest), and *c4* (fastest) from Fig. 6. Intuitively, this corresponds to accelerate as much as possible CHANGE-DETECTION, which is the bottleneck, while keeping DEBAYER and GRADIENT small to improve area occupation.

With the available information at this stage, the compositional DSE ignores contention for shared-resources as well as the interconnect area overhead. In this ideal scenario, each accelerator starts the computation as soon as the predecessors have completed theirs. In Section IV we explain how to overcome these limitations.

III. THE ESP FLEXIBLE SOCKETED ARCHITECTURE

HLS enables quick tuning of accelerator IP blocks in isolation. In addition, application-level analysis can be performed by combining the results from each accelerator with a dependency graph. The effective cost and performance of a design, however, can be estimated only accounting for the interaction across multiple accelerators, the interconnect, and the system I/O. A system-level DSE must be performed to obtain the SoC Pareto curve, which considers the environment in which the accelerators run. The combination of the architecture and methodology of Embedded Scalable Platforms enables fast system-level DSE because it simplifies the integration of heterogeneous IP blocks into an SoC. In particular, we designed the ESP tile-based architecture to strike the right balance between heterogeneity and regularity. It consists of a set of templated hardware and software sockets that enable the generation and programming of a complete SoC by assembling a configurable infrastructure with off-the-shelf processors and accelerators. The latter are designed following the steps described in Section II.

ESP Accelerator Sockets. Fig. 8 shows the encapsulation of an accelerator IP block in an ESP configurable tile, which implements the platform services and provides access to the system interconnect. An accelerator tile hosts a hardware socket, whose signal-level interface matches the accelerator interface described in Section II. The interface exposes:

- Input read and output write requests from the accelerator; these are in the form of DMA bursts with configurable bit-widths. The burst length and accelerator virtual address are set at run-time by `load_input()` and `store_output()` SystemC threads. The socket implements a latency-insensitive protocol [8] matching the behavior of the TLM point-to-point channels used during the accelerator design. Relaxing interface requirements with a latency-insensitive protocol enables a seamless replacement of an IP implementation with any other alternative ones taken from its Pareto-optimal set [7].
- Memory-mapped configuration registers are used for run-time setup of the accelerators. Besides common command and status registers, all user-defined registers and their memory mapping are generated based on the data structure `conf_info` defined by the interface of Fig.5.
- Interrupt notifications; the accelerator triggers an interrupt request when it completes or in case of error.

The ESP hardware sockets implement the TLM ab-

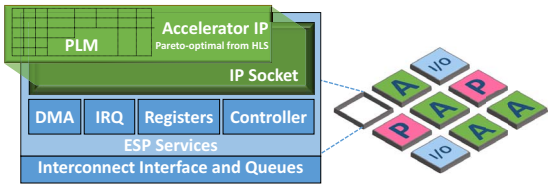

Fig. 8. Configurable ESP Accelerator tile with hardware socket.

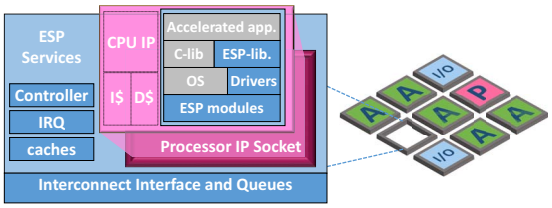

Fig. 9. ESP Processor tile with software socket.

straction used during the HLS of the accelerators. In this way the IP designer can be completely unaware of the SoC interconnect specifications. Transactions are translated into platform-dependent messages directed to either a processor or an I/O tile. Similarly, all environment requests are forwarded to the simple accelerator's interface.

ESP Processor Sockets. ESP eases the process of integrating the accelerators in a heterogeneous SoC by pairing accelerator sockets with software sockets running on processor. Similarly to accelerators, a processor is also encapsulated in a tile which implements ESP platform services. In this case, the DMA engine is replaced by a cache, which gives the illusion of a traditional homogeneous system and decouples the processor bus from the rest of the SoC. Hence, legacy software can transparently execute on the processor. The processor socket is completed by three software layers as shown in Fig. 9.

- *ESP Linux modules.* This low-level software allows Linux to recognize the accelerators in the ESP tiles. It implements interrupt registration/handling and primitives to configure all ESP common registers with one `IO_CTL` system call. The modules relieve the IP designer from writing complex low-level routines for each accelerator.
- *ESP Linux device drivers.* Since ESP accelerators are seen by the operating system like any other peripheral, they need a device driver. The use of ESP template drivers requires the programmer only to implement the behavior of user-defined control registers. Lower-level routines, in fact, are provided by ESP modules. For instance, across all accelerators for WAMI-App, the "accelerator specific" code that is user-provided represents on average less than 2% of the entire device driver.
- *ESP user-level library.* This library has two main purposes: (1) it implements an API to invoke accelerators within user-level applications and (2) it provides a multi-threaded infrastructure to perform a DSE of multi-accelerator applications. For instance, an application like WAMI-App consists of multiple kernels, each potentially implemented by an accelerator. Thanks to the library, each accelerator is controlled by a `Pthread` and a hidden queue-based mechanism synchronizes its execution.

The combination of ESP hardware and software sockets allows us to bring up a system in a very short time and to perform a DSE from a system-level viewpoint. Similarly to TLM, which decouples computation from communication, our software library decouples accelerators management from the application data-flow. To build an ESP application we must perform a few tasks. First, we reserve a buffer in DRAM. Such memory region should be large enough to hold input, output and intermediate data processed by the accelerators. Then, we prepare a data structure for every accelerator which includes all necessary memory offsets with respect to the memory buffer and the configuration for the user-defined registers. ESP modules take care of building a page table for each accelerator which allows them to perform autonomous DMA transactions. Finally, for every accelerator we prepare a data structure specifying the name of the device and the list of devices that produce its inputs. Each accelerator, in fact, is associated with an output queue that corresponds to a memory buffer in main memory. A thread runs and starts its accelerator when all of its input queues have at least one set of valid data and the output queue has room to store the result. The size of the queues is configurable and it has a direct impact on the parallelism that can be exploited by the multi-threaded ESP application. In general, larger queues enable more concurrency among the accelerators. In practice, power caps, contention for shared resources, data dependencies, and diversity in the accelerator execution time may impose unexpected limits to the benefits of parallelism.

IV. Full-System Design Space Exploration

To show how ESP supports the system integration phase of Fig. 1, we generated various SoCs for WAMI-app: each SoC is an ESP instance featuring one processor tile, two I/O tiles connected to DRAM banks, a multi-plane NoC, and a set of twelve to fifteen accelerator tiles. Each accelerator tile maps to one kernel of WAMI-app except from Matrix Inversion, which is executed in software to preserve floating-point precision. For each ESP instance we built a prototype on a `Xilinx Virtex7` FPGA.

The relations among the WAMI-app kernels shown in Fig. 2 translates into data dependencies across the corresponding accelerators: e.g., some kernel-level parallelism can be exploited by supporting multiple concurrent executions of the four Warp kernels. In fact, the Pareto curve of Fig. 7 is derived assuming four parallel Warp accelerators. For our experiment, we selected three implementations from Fig. 7: the fastest (labeled `a1`), the smallest (`a12`), and a medium one (`a3`). For each of these implementations, we considered three additional derivative implementations, each obtained from the previous one by removing one Warp accelerator. Hence, we have 12 ESP instances that correctly implement WAMI-app while of-

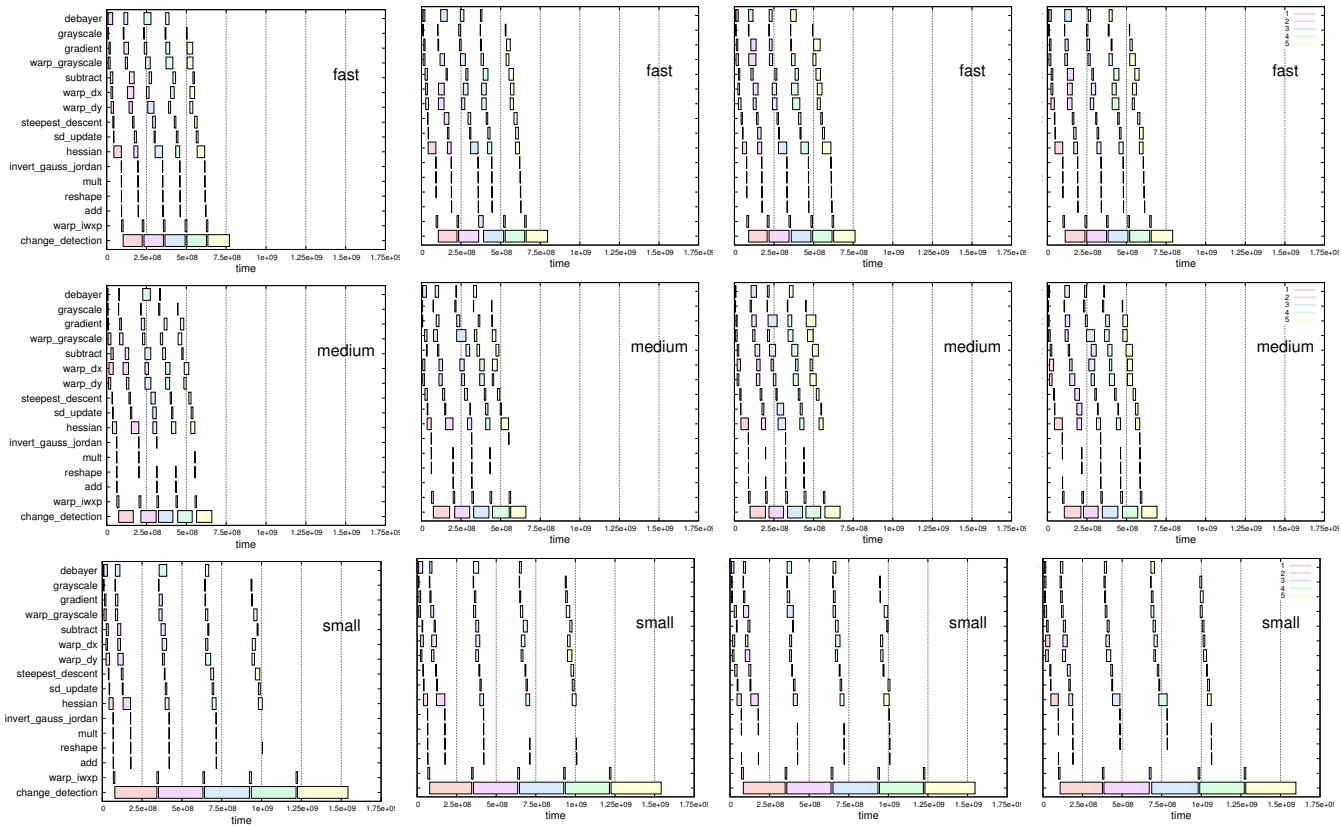

Fig. 10. Example of full-system DSE. The Gantt charts are obtained from emulating 12 ESP instances for WAMI-app on an FPGA board.

fering different degrees of parallelism for the WARP kernels.

Fig. 10 shows the Gantt chart of the WAMI-app execution on each ESP instance. The four top, middle, and bottom charts of Fig. 10 correspond to `a1`, `a3`, and `a12`, respectively. Moving from left to right we see the effect of reducing the number of WARP accelerators available in the system. In each chart, a bar corresponds to a time interval where a thread is active. For example, the first line of each chart shows the active time intervals for the thread controlling DEBAYER. The color of the bars identifies the frame number (1 to 5) processed by the accelerators. Note that the active interval includes the waiting time for the accelerator interrupts, as well as additional waiting time due to contention on shared resources. For instance, by looking at the three yellow bars for WARP-GRAYSCALE, WARP-DX and WARP-DY in the rightmost top chart, we notice that the latter is immediately granted access to the sole WARP accelerator, even if WARP-GRAYSCALE is activated first. When WARP-DY releases the accelerator, WARP-GRAYSCALE resumes. Eventually, WARP-DX can also reserve the accelerator and processes Frame #5. The analysis of Fig. 10 shows that the `a1` and `a3` implementations enable much higher parallelism than `a12`. The latter suffers more the fact that CHANGE-DETECTION dominates the execution time, preventing other accelerators to proceed.

The latency vs. area chart of Fig. 11 reports the points corresponding to the twelve implementations of Fig. 10 together with the three Pareto-optimal points (the empty triangles) taken from Fig. 7. The arrows highlights how these three points "migrate" in the chart when one accounts for the area of the interconnect, processor, and network interfaces, as well as for all the system factors impacting the actual execution time (reported in Fig. 10). While the SoC area is easy to estimate, the FPGA emulation is critical to obtain a precise performance estimation through an accurate analysis of the interaction among all SoC components. By enabling a quick generation of different SoC instances, our ESP architecture and methodology supports a fast and accurate DSE of complex heterogeneous SoCs. For example, the analysis of Fig. 10 shows that reducing the number of WARP accelerators saves area with a negligible performance penalty. And the analysis of Fig. 11 shows that all implementations derived from `a1` are no longer Pareto optimal in a real scenario because the small performance gain over `a3` of the ideal scenario evaporates when the accelerators must share the interconnect fabric and the two DRAM channels.

V. RELATED WORK

Since specialized hardware is key to energy-efficient performance [21], many approaches have been proposed for accelerator design [16, 20, 34, 29]. With ESP, we focus on high-throughput accelerators that are designed independently from processor cores and operate independently from them to execute large workloads [10]: these loosely-

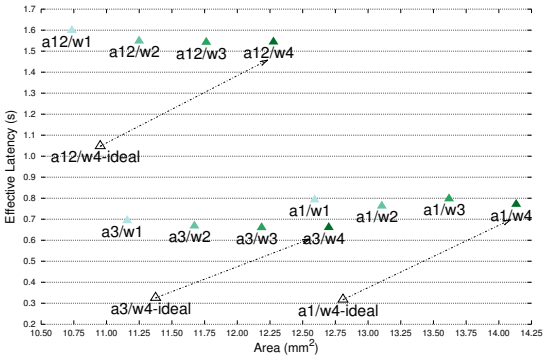

Fig. 11. Pareto-point migration from ideal to full-system scenario.

coupled accelerators are easier to integrate and reuse.

SLD methodologies, which raise the design process to a higher level of abstraction from the familiar RTL thereby simplifying the specification, implementation and verification of complex SoCs, have been advocated for years [3, 13, 18]. The 2011 ITRS, however, still lamented that CAD tools provide little support for SLD and that this situation must change if necessary advances in productivity are to be achieved [1]. Recently there has been some progress in the commercial production of SLD tools, particularly for HLS [9, 11, 17] but their adoption is not widespread and HLS usage is still limited to small portions of the designs [24]. We propose a SLD methodology that embraces existing commercial HLS tools and provides guidelines to exploit it for increased productivity in heterogeneous SoC design. Instead of proposing a new language for hardware design [2], we leverage SystemC, an IEEE-standard, and we provide an infrastructure to guide its use for improved SLD productivity.

DSE is a crucial step of the design process that should be brought to the highest possible abstraction level. While simulators that enable early power-performance analysis are useful for modeling individual accelerators [33], with ESP we advocate the use of HLS to build prototypes that enable full-system DSE, thus accounting for the complex interactions among all SoC components.

VI. Conclusions

Embedded Scalable Platforms (ESP) combine an architecture and a methodology. The flexible socketed architecture addresses the complexity of component integration in heterogeneous SoCs. The companion methodology raises the level of abstraction to system-level design to promote close collaboration among software programmers and hardware engineers in selecting the best mix of components for a given application-driven SoC. By balancing regularity, flexibility, and specialization, ESP simplifies the reuse and integration of IP blocks. It also accelerates the deployment of full-system prototypes, which are key to an accurate evaluation of cost-performance trade-offs.

Acknowledgments. This work is supported in part by DARPA PERFECT (C#: R0011-13-C-0003), the NSF (A#: 1219001), and C-FAR (C#: 2013-MA-2384), an SRC STARnet centers.

References

[1] 2011 International Technology Roadmap for Semiconductors. *www.itrs.net*.
[2] J. Bachrach et al. Chisel: Constructing hardware in a Scala embedded language. In *Proc. of DAC*, pages 1212–1221, June 2012.
[3] B. Bailey and G. Martin. *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer-Verlag, 2006.
[4] K. Barker et al. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. PNNL and GTRI, Dec. 2013. `http://hpc.pnnl.gov/PERFECT/`.
[5] D. C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up, Second Edition*. Springer-Verlag, 2010.
[6] S. Borkar. Design perspectives on 22nm CMOS and beyond. In *Proc. of DAC*, pages 93–94, 2009.
[7] L. P. Carloni. From latency-insensitive design to communication-based system-level design. *Proc. of the IEEE*, 103(11):2133–2151, Nov. 2015.
[8] L. P. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD*, 20(9):1059–1076, Sept. 2001.
[9] J. Cong et al. High-level synthesis for FPGAs: from prototyping to deployment. *IEEE Trans. on CAD*, 30(4):473–491, Apr. 2011.
[10] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proc. of DAC*, pages 202:1–202:6, June 2015.
[11] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.
[12] W. Dally, C. Malachowsky, and S. Keckler. 21st century digital design tools. In *Proc. of DAC*, pages 1–6, May 2013.
[13] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli. A platform-based taxonomy for ESL design. *IEEE Design & Test of Computers*, 23(5):359–374, May 2006.
[14] G. Di Guglielmo, C. Pilato, and L. P. Carloni. A design methodology for compositional high-level synthesis of communication-centric SoCs. In *Proc. of DAC*, pages 128:1–128:6, June 2014.
[15] H. Esmaeilzadeh et al. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376, June 2011.
[16] H. Esmaeilzadeh et al. Neural acceleration for general-purpose approximate programs. In *Proc. of Micro*, pages 449–460, 2012.
[17] M. Fingeroff. *High-level Synthesis Blue Book*. Mentor Graphics Corp., 2010.
[18] A. Gerstlauer et al. Electronic system-level synthesis methodologies. *IEEE Trans. on CAD*, 28(10):1517–1530, 2009.
[19] R. Goering. Are SoC development costs significantly underestimated? http://www.cadence.com/Community/blogs.
[20] V. Govindaraju et al. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.
[21] M. Horowitz. Computing's energy problem (and what we can do about it). In *ISSCC*, pages 10–14, Feb. 2014.
[22] H.-Y. Liu, M. Petracca, and L. P. Carloni. Compositional system-level design exploration with planning of high-level synthesis. In *Proc. of DATE*, pages 641–646, Mar. 2012.
[23] H. Mair et al. A highly integrated smartphone SoC featuring a 2.5GHz octa-core CPU with advanced high-performance and low-power techniques. pages 424–425, Feb. 2015.
[24] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.
[25] J. Park et al. A 646GOPS/W multi-classifier many-core processor with cortex-like architecture for super-resolution recognition. In *ISSCC*, pages 168–169, 2013.
[26] C. Pilato, P. Mantovani, G. D. Guglielmo, and L. P. Carloni. System-level memory optimization for high-level synthesis of component-based SoCs. In *Proc. of CODES+ISSS*, pages 18:1–18:10, Oct. 2014.
[27] R. Porter, A. M. Fraser, and D. Hush. Wide-area motion imagery. *IEEE Signal Processing Magazine*, 27(5):56–65, 2010.
[28] J. Pyo et al. 20nm high-K metal-gate heterogeneous 64b quad-core CPUs and hexa-core GPU for high-performance and energy-efficient mobile application processor. In *ISSCC*, pages 420–421, Feb. 2015.
[29] W. Qadeer et al. Convolution engine: balancing efficiency & flexibility in specialized computing. In *Proc. of ISCA*, pages 24–35, June 2013.
[30] R. Saleh et al. System-on-chip: Reuse and integration. *Proc. of the IEEE*, 94(6):1050–1069, 2006.
[31] A. Sangiovanni-Vincentelli. Quo vadis SLD: Reasoning about trends and challenges of system-level design. *Proc. of the IEEE*, 95(3):467–506, 2007.
[32] O. Shacham et al. Rethinking digital design: Why design must change. *IEEE Micro*, 30(6):9–24, Nov.-Dec. 2010.
[33] Y. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. The Aladdin approach to accelerator design and modeling. *IEEE Micro*, 35(3):58–70, May 2015.
[34] G. Venkatesh et al. Conservation cores: reducing the energy of mature computations. In *Proc. of ASPLOS*, pages 205–218, Mar. 2010.
[35] Y. Yoon, N. Concer, and L. P. Carloni. Virtual channels and multiple physical networks: Two alternatives to improve NoC performance. *IEEE Trans. on CAD*, 32(12):1906–1919, Dec. 2013.
[36] V. Zyuban and P. Strenski. Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels. In *Proc. of ISLPED*, pages 166–171, 2002.