The CompSOC Design Flow for Virtual Execution Platforms

Sven Goossens[1,†], Benny Akesson[1], Martijn Koedam[1],
Ashkan Beyranvand Nejad[2], Andrew Nelson[2], Kees Goossens[1]
[1]Eindhoven University of Technology
[2]Delft University of Technology
[†]s.l.m.goossens@tue.nl

ABSTRACT

Designing a SoC for applications with mixed time-criticality is a complex and time-consuming task. Even when SoCs are built from components with known real-time properties, they still have to be combined and configured correctly to assert that these properties hold for the complete system, which is non trivial. Furthermore, applications need to be mapped to the available hardware resources and correctly integrated with the SoC's software stack, such that the real-time requirements of the applications are guaranteed to be satisfied. However, as systems grow in complexity, the design and verification effort increases, which makes it difficult to satisfy the tight time-to-market constraint.

Design tools are essential to speed up the development process and increase profit. This paper presents the design flow for the CompSOC FPGA platform: a template for SoCs with mixed time-criticality applications. This work outlines how the development time of such a platform instance is reduced by means of its comprehensive tool flow, that aids a system designer in creating hardware, the associated software stack, and application mapping.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Verification

Keywords

Virtual Execution Platform, Composability, Predictability, FPGA, CompSOC

1. INTRODUCTION

The complexity of embedded systems is increasing, caused by the growing functionality that is embedded in a single device. This trend is driven by increased connectivity, closer interaction with the outside world through sensors and actuators, and diversification of applications [10]. The power budget of embedded systems does not increase proportionally [1,28], so resources have to be shared to reduce costs and power. This leads to the creation of heterogeneous multiprocessor architectures [19, 24] that execute many applications of mixed time-criticality on a single chip.

In mixed time-criticality systems, some applications have real-time requirements, while others do not. Sharing chip resources amongst multiple applications makes their functional and timing behavior interdependent, increasing the verification effort significantly [1,10,22]. This leads to growing verification costs and reduced test coverage.

To alleviate these issues, applications would ideally be independently developed and verified. *Composability* and *predictability* are two concepts that facilitate this [4], and the CompSOC platform template [11,14] demonstrates this design philosophy. CompSOC provides applications with their own composable *virtual execution platform*, thus isolating their behavior down to the cycle level. This enables independent development and verification of applications, because they experience no inter-application interference at run time. Each virtual execution platform is also predictable, such that guarantees on the worst-case performance for formally analyzable applications can be derived.

Virtual execution platforms are hosted on a physical SoC. Creating such a SoC is a big multi-disciplinary task that involves the design of hardware, both at the component and system level, algorithms for performance analysis, synthesis tools, OS integration and application programming, mapping and verification. Tools and tool flows are essential for design teams to bring all these disciplines together in an efficient way and to be sufficiently productive.

This paper presents an outline of the FPGA tool flow used to generate CompSOC platform instances. It consists of three sub-flows that aid the system designer in several ways. It offers:

1. A hardware tool flow, capable of translating a high-level description of a CompSOC platform instance into a fully synthesized implementation.
2. A system software flow, generating a software stack including a composable micro kernel, resource managers, drivers, and a virtual platform boot loader.
3. An application flow that automatically generates a virtual platform configuration for applications that use the Cyclo-static Data Flow (CSDF) [9] model of computation. The application is automatically mapped to that virtual execution platform.

This tool flow applies platform-based design techniques [18] to greatly reduce the design effort by automatically combining a set of verified hardware and software components into a platform instance based on a flexible template. Combined with the complexity reduction resulting from the use

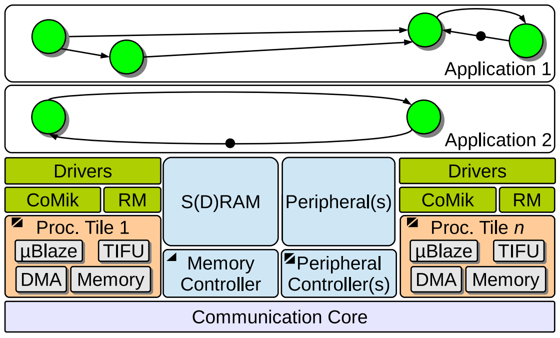


Figure 1: High-level view of the platform.

of virtual execution platforms, the time-to-market decreases significantly.

This paper starts with a brief introduction to the CompSOC platform in Section 2. Section 3 discusses the three sub-flows of the tool flow. Finally, Section 4 shows the results of a run through the hardware flow, followed by conclusions in Section 5.

2. COMPSOC HARDWARE PLATFORM

CompSOC is a template for a heterogeneous processing platform that provides virtual execution platforms to its applications. It contains processing, communication and memory resources, and a set of optional peripherals. A range of techniques is applied to share these physical resources in a composable and predictable way [4]. The tool flow presented in this paper is targeted at an FPGA development board, although most of the flow is target agnostic.

Fig. 1 shows a high-level view of a CompSOC instance, where master and slave Intellectual Property (IP) components (marked with a black triangle) use the communication core to exchange information: a master initiates transactions while a slave can only react to requests from a master. Section 2.1 discusses the master and slave IPs in more detail and Section 2.2 explains the contents of the communication core.

2.1 Master and Slave IPs

The processor tiles [6] execute the tasks that constitute the applications running on the platform. The tiles are built from a standard MicroBlaze processor, amended with a custom-made Timer Interrupt Frequency Unit (TIFU), which contains several hardware timers and controls the operating frequency of the processor. It communicates with a Composable Micro Kernel (CoMik) that runs on the tile to share the processor cycles amongst multiple virtual execution platforms in a composable manner. The TIFU also enables composable frequency scaling [21] and virtualizes interrupts [8]. CoMik builds on the concepts introduced in [17] to achieve this. At the most basic level it uses non-work-conserving Time-Division Multiplexing (TDM) application scheduling to isolate their execution in time. There are many details that have to be addressed to achieve complete cycle-level isolation [4], but they are out of the scope of this paper.

Tiles can be extended with one or more Direct Memory Access (DMA) units, used by applications to decouple computation and communication [20]. This allows applications to be preempted while their DMA processes a transaction independently, thus making the minimum preemption interval independent from the communication latency. Each tile contains a configurable number of communication memories that are primarily used as software-controlled FIFOs for inter-task communication, both within the same tile and across tiles.

Memories are complementary to the master IPs in the system: they are the slaves that react to requests from the masters. Composable shared memory access is made possible using the complete real-time memory solution proposed in [5]. It consists of a generic front-end [3] that can be connected to either an SRAM or SDRAM real-time memory controller back-end [12].

Peripherals may have both master and slave ports. For example, a TFT controller is the master of its connection to a frame buffer memory, while it is a slave to the processor that controls the output resolution it uses.

2.2 Communication Core

The communication core (Fig. 1) is the glue layer that connects masters to slaves. At its heart lies the time-triggered *(d)AElite NoC* [16,23] that uses non-work-conserving TDM-based scheduling to temporally isolate traffic from separate virtual execution platforms. Its primary purpose is to bridge the physical distance between IPs in the platform in a pipelined, wire-efficient way. It uses a streaming protocol, uniform data widths and a single clock frequency. An optional *Ethernet bridge* can be added to connect two NoCs residing on separate chips [7].

All the shared memories are mapped within a global address space, making them accessible to the masters in the platform through the communication core. Furthermore, all run-time configurable components have memory-mapped configuration registers within this same address space and are reachable using Memory Mapped I/O (MMIO) connections. To realize this distributed shared memory system, two more hardware layers are built on top of the NoC.

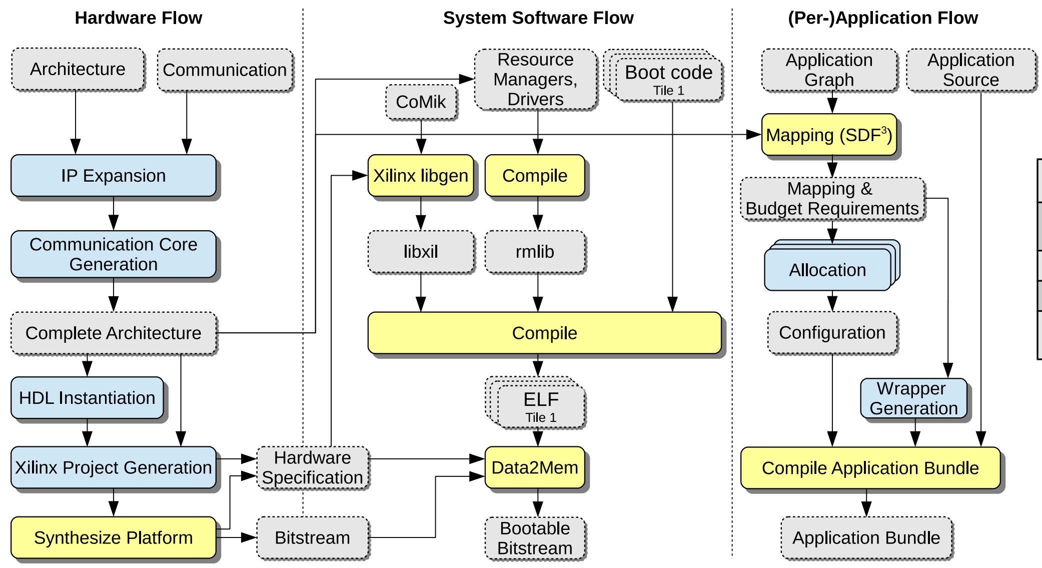
The first layer is responsible for *harmonization / diversification* of the connections that enter / leave the NoC. It handles protocol conversions, bus width conversion and clock domain crossings, and is basically a set of adapters that allow a diverse set of masters with different properties to use the same shared resource.

The *sharing / distribution* layer can multiplex traffic from multiple masters into a single slave port, or demultiplex traffic from a single masters into multiple slave ports [3, 15]. Several kinds of arbiters like TDM, round-robin (RR) or Credit-Controlled Static-Priority (CCSP) [2] are available in the CompSOC hardware library, and can be instantiated to control accesses to shared resources. The following sections show how these building blocks are used in the CompSOC flow.

3. TOOL-FLOW OVERVIEW

The flow can be divided in three sub-flows (Fig. 2), each of which is responsible for generation of different aspects of the platform:

1. The *Hardware flow* constructs the hardware and wraps it in a project that is used by the FPGA synthesis tools.
2. The *System software flow* generates a software stack including a composable micro kernel, resource managers, drivers, and a virtual platform boot loader.
3. The *Application flow* maps applications to the tiles, memories and communication resources on the platform, generating a virtual platform configuration. It creates a bundle of this configuration and the application code, which can be loaded and started at run-time.



Source Lines of Code (SLOC)	
Contributed Tools	56k (c++)
Drivers	2.5k (c, asm)
CoMik	1.5k (c, asm)
Hardware library	42k (vhdl, verilog)

Figure 2: The tool flow. Blue, yellow and gray block represent contributed tools, pre-existing tools and files, respectively.

As shown in Fig. 2, there are dependencies between the three sub-flows, requiring the user to first generate the hardware before initiating the system or application flow. It is important to note, however, that applications can be developed independently from one another, because their run-time behavior is isolated within their virtual execution platform.

The tools are written in C++, and most of the hardware is written in VHDL. The majority of the input and output files are proprietary XML documents that are incrementally refined by the steps in the flow. This makes it easy to insert or remove tools when required, and creates intermediate points where manual modifications are possible. In the following sections, each of the sub-flows is discussed in more detail.

3.1 Hardware Flow

The goal of the hardware flow is to generate instances of the CompSOC platform with the least possible effort from the designer by automating as much as possible. The principle by which the flow is built is that a minimal high-level architecture description should produce a working and fully synthesized platform, based on a sensible default specification. It is possible for the designer to deviate from the standard by describing the required platform more specifically and let the tools check the specification for consistency. The end product of the hardware flow is a bit file that can be programmed onto the FPGA.

The platform is generated based on existing components that are instantiated and connected on demand. Some of these components are custom made, while others are standard hardware components from Xilinx's library. Custom hardware is used to attain the predictability and composability required to create virtual execution platforms. What happens in the each of the steps in the hardware flow (Fig. 2) is explained in the next sections.

3.1.1 IP Expansion

The designer provides an *architecture* file, describing the master and slave IPs to be included in the platform, e.g. processor tiles, memories and peripherals. The second input to the IP expansion tool is the communication file that describes the fan-in and fan-out of the slave ports and master ports, respectively.

This minimal description is enough to start the tool flow, but more details can be supplied to customize the hardware synthesis process. For example, it is possible to specify extra clock domains and to bind these to IP ports.

The IP expansion tool refines the architecture specification with the implicit components that follow from the CompSOC template. IP components that are known within our hardware library are automatically populated with their default parameters and interfaces. It is thus not required to specify their exact content or even interface, saving time and reducing the chance of specification errors. For the processor tiles for example, the TIFU, DMA, internal buses, and communication, instruction and data memories are automatically added. The interface description of the tile, which is exposed to the rest of the platform, is automatically generated. The number of DMAs and the size of memories are example parameters that can be explicitly set if required. The tool requires a more verbose description for IPs that are not known within the hardware library, enumerating their ports and buses, the used bus standards, and data widths.

All ports are automatically annotated with their address range, effectively generating the memory map. By the end of this step, the interface descriptions of the IPs in the platform is fully known, which is required to generate the communication core, as discussed in Section 3.1.2.

3.1.2 Communication Core Generation

The communication core is the glue layer that connects IPs, and it is generated completely automatically. As discussed in Section 2.2, it consists of 3 layers that are created sequentially.

First, the tool adds the sharing/distribution layer by connecting (de-)multiplexers to the IP ports based on the communication file. The memory front-end [3] is inserted before all shared memory ports. The designer may choose what type of arbitration to use for a slave port, or leave it up to the tool to instantiate a default round-robin arbiter. It is possible to annotate connections to bypass the third (NoC) layer completely, which can be used for IPs that require a very low latency connection and can afford to be physically close to each other on the platform.

Next, the harmonization/diversification layer is added, which adapts all port interfaces such that they can be connected to their immediate target (either the NoC or another IP port). It resolves all mismatches in the interfaces automatically by instantiating protocol adapters, bus width converters and clock domain crossings.

Finally, the NoC layer is generated. The designer has to specify the NoC topology (e.g. mesh, ring, or fat tree), and the number of routers and network interfaces (NI) to generate. The number of ports on the NIs and routers is then automatically determined by the flow, depending on the number of connections that were specified. Optionally, it is possible to bind specific network interfaces to an IP or port, in case the designer wants to force a specific layout onto the design.

All run-time configurable components in the platform, like memory controller arbiter priorities, NI slot tables and router paths for example, are accessible through a shared configuration channel that is mapped on the NoC [13]. Demultiplexers are added at NIs that are responsible for configuring multiple components.

A final dimensioning step determines the buffer and slot table sizes that are used throughout the platform. The slot table size determines the granularity at which resource budgets can be handed out to virtual execution platforms, while the buffer sizes influence the maximum attainable throughput on a physical connection. By default, they are dimensioned such that it is unlikely they limit mapability or are the connection bottleneck, respectively, but the designer is free to set the size manually when desired.

Once the communication core has been generated, the architecture is ready to be instantiated.

3.1.3 HDL Instantiation

The communication core that was generated by the previous steps in the flow is turned into a HDL description of the hardware in this step. It prepares the communication core for processing by the synthesis tools by gathering the HDL files of its components, and prints a top-level VHDL file that instantiates them. The components are customized with their generics and the inter-component wiring added.

The communication core is a generic hardware block that can be used with any HDL synthesis tool. To make it compatible with the tools for the target FPGA, a Virtex-6 from Xilinx, it is wrapped in their proprietary *PCore* format, essentially turning it into one large black box. All custom IPs that are not within the communication core receive the same treatment, such that the complete platform can be instantiated in a Xilinx project.

3.1.4 Xilinx Project Generation

The last step in this flow synthesizes the platform into a bit file that can be programmed onto an FPGA. It is vendor specific, and targets a Virtex-6 FPGA on either a ML605 board [29], or the Sundance FlexTiles board [27].

Xilinx Platform Studio (XPS) is used to synthesize the platform. Our tool generates a platform description (MHS-file) and synthesis constraint file (UCF-file), simply by iterating over all IPs that are used in the platform, including the communication PCore, and printing their contributions. All specified clocks are assigned to clock generators, and the global reset is synchronized to each clock domain, creating per-domain synchronous resets. Fig. 3 shows an example of a complete platform instance.

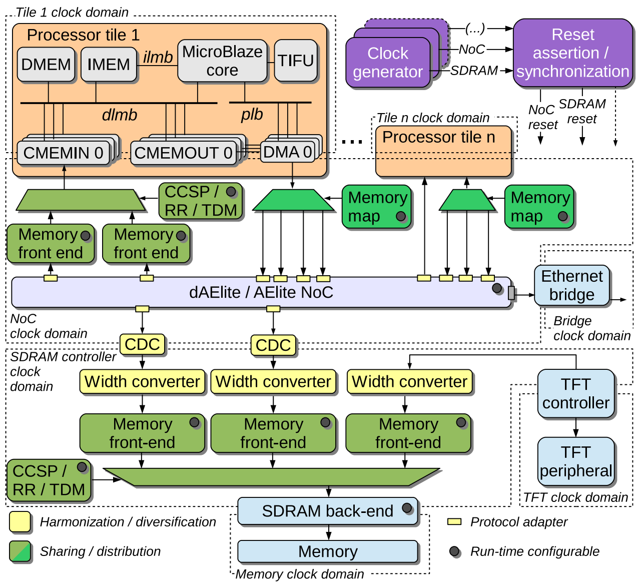


Figure 3: An example platform instance.

After generation of the project, the platform can be synthesized into a bit file. No user input is required after the initial input at the top of this flow, assuming all used IPs are in CompSOC's or Xilinx's IP library. If other IPs are used, their contributions to the XPS project would have to be added manually.

3.2 System Software Flow

The system software flow builds a software stack for the platform, consisting of resource managers, hardware drivers and code required to boot virtual platforms. This flow was also shown in Fig. 2, and is discussed here in more detail.

Each of the tiles runs an instance of CoMik, which is a micro kernel that provides temporally isolated partitions in which applications are executed. CoMik interfaces with the hardware timers and interrupts provided by the TIFU to achieve this isolation. It additionally handles the stack and heap management, and provides an API for using the DMAs. CoMik is treated as an OS by the Xilinx *libgen* tool, which combines it with the MicroBlaze drivers into a single library.

The *resource managers* use the *drivers* to configure the remaining run-time configurable resources in the platform, e.g. the NoC paths, and the (de-)multiplexer arbiters and memory maps for example (see Fig. 3). The resource managers act as an abstraction layer, keep track of the resources that are in use, and do basic checks for faulty configurations. The completed architecture description specifies which configurable resources are present in this platform instance, and what their memory addresses are. This information is compiled into the resource management library.

The final piece of software that is added is the virtual platform *boot loader* for the different tiles, which parses the application bundles produced by the application flow at run time (see Section 3.3.2) to instantiate a virtual platform.

The above mentioned software is compiled into an ELF file for each tile, which is loaded into the appropriate memories in the bit stream by the Xilinx *Data2Mem* tool. The bit stream now contains a bare bootable system that is ready to receive the application bundles from the application flow, and to instantiate them at run time.

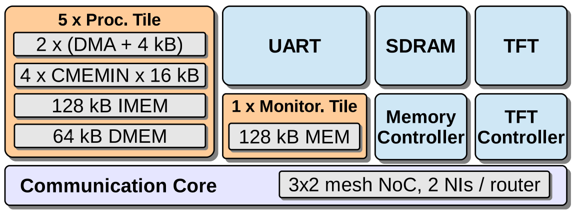


Figure 4: The architecture used in the hardware generation experiment.

3.3 Application Flow

The final sub-flow is responsible for mapping applications to processor tiles, memories and communication resources on the platform. This can be seen as the dimensioning step for the application's virtual execution platform. The end product is an *application bundle* that combines the application code with a description of its virtual execution platform.

3.3.1 Mapping and Allocation

For real-time CSDF applications, the application flow starts with the *mapping* tool. Its input is the application graph, annotated with requirements on throughput and response times, and the source code for its actors. A description of the architecture produced by the hardware flow describes the available hardware resources. With these inputs the mapping tool invokes the SDF^3 tools [25, 26] to do a data-flow-based design-space exploration that attempts to find a valid mapping that satisfies the application requirements. The mapping process can optionally be constrained to reserve portions of the physical platform for other virtual execution platforms. Alternatively, the result of previous mappings can be used to determine which resources are already in use. The first option separates the mapping process of different applications, potentially at the cost of over-allocation. The result of the design-space exploration is a description of the used platform resources, and the required budget on those resources in case they are shared.

The *allocation* tools then find configurations for the resources that satisfy these budget requirements. The exact output depends on the resource type: for a processor tile it contains the TDM schedule, for the communication resources it contains the path to take through the communication core and the NoC TDM schedule, and for shared memories it contains the multiplexer arbiter configuration.

For applications that use a different model of computation, like time-triggered [8], general data flow, or non-real-time applications, the designer manually has to specify the requirements for the virtual execution platform. The allocation tools can then generate a configuration for the resources to satisfy those requirements.

3.3.2 Application Bundles

The final step in the application flow is to generate the software that runs on each processor tile. This involves the creation of initialization and wrapper code for the actor functions that contain the appropriate API calls to use the processor's communication interfaces. Furthermore, the platform configuration is translated into a format that can be parsed by the run-time resource managers. These two components are combined into a single *application bundle* that contains all required information to deploy the application on the platform.

Application bundles are stored in the SDRAM. When an

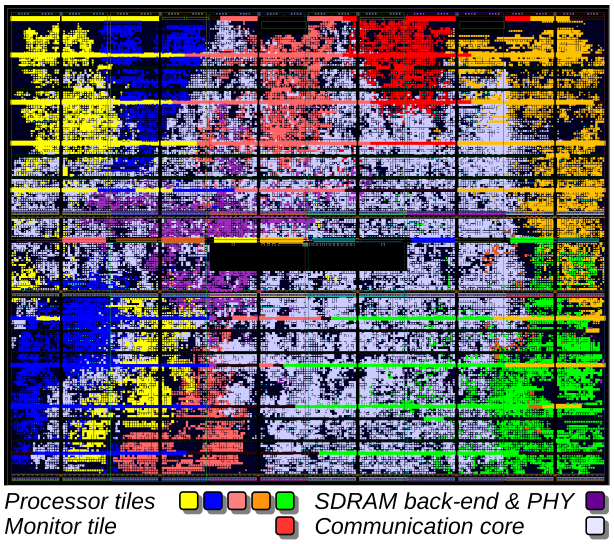


Figure 5: A 5-tile platform.

application is started, the bundle's header is read by the virtual platform boot loader, and the application's virtual platform configuration is interpreted by the resource managers. After applying this configuration to the platform, the application's memory sections are copied to the appropriate physical memories. The virtual execution platform is then booted, and application execution starts.

4. HARDWARE FLOW DEMONSTRATION

This section demonstrates an example run of the hardware flow, targeted at the ML605 board that contains a Virtex-6 LX240T FPGA. As input, we specify an architecture (Fig. 4) with five processor tiles, an SDRAM and a TFT peripheral. Each tile has four input communication memories (CMEMIN), and two DMAs with an attached output communication memory. The sizes of the local tile memories are set by hand (see Fig. 4). All tiles run in their own 100 MHz clock domain, while the SDRAM controller is clocked at 150 MHz. A small monitor tile gathers debug information, which it sends to a host PC using a UART connection.

The connectivity graph is specified as follows: each DMA is connected to three NoC ports. The SDRAM receives five NoC-based input ports, and each input communication memory connects to two ports. A 2x3 mesh NoC topology is specified to implement these connections. The TFT controller works autonomously by reading pixel data from a predefined SDRAM range using a sixth SDRAM port, and it bypasses the NoC layer.

With this input, the tool flow generates a Xilinx project in less than a minute, after which the synthesis tool uses about three hours to create the bit stream. The resulting placement is shown in Fig. 5. The complete design uses 19% (59513) of the registers, and 59% (88953) of the LUTs on the FPGA, spread out over 82% of the available slices. The local tile memories are implemented as BRAM blocks, visible as large horizontal bars in Fig. 5. The BRAM utilization is 87% and has a large influence on the placement of the tiles, since it forces them to be spread across the entire FPGA to get close to their memories. For the three left-most tiles, the TIFU is separated from the other logic, and located on the opposite end of the FPGA. The placement of the SDRAM back-end and its PHY is driven by the location of the pins connected to the SDRAM.

The communication core uses 8% of registers and 25% of the available LUTs. This includes all the buffers used in the NoC and memory front-ends, including that of the SDRAM controller. These buffers are implemented such that they map to *distributed RAM* and *SRL shift register* primitives to efficiently store relatively large amounts of data on the FPGA fabric.

After synthesis is complete, the system software flow is executed. A small self-test is included in the generated files to verify that all platform components are reachable and all end-to-end connections function correctly. All tiles report their success or failure to the monitor, which forwards the status to the user, who can then start the application flow.

5. CONCLUSIONS

The growing complexity of SoCs increases their design and verification time. This makes it more and more difficult to finish development and verify real-time application behavior within tight time-to-market constraints. In this paper, we have presented the comprehensive CompSOC tool flow, which reduces the development and verification time by aiding the system designer in the areas of hardware design, software stack generation, and application mapping. These tools can be used to create customized instances of the CompSOC platform, which uses virtual execution platforms to enable verification in isolation for its applications, further reducing the total system development time.

6. ACKNOWLEDGEMENTS

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, Catrene CA104 COBRA, CA505 BENEFIC and CA703 OpenES, and NL STW 10346 NEST.

7. REFERENCES

[1] International Technology Roadmap for Semiconductors (ITRS), 2011. http://www.itrs.net/reports.html.

[2] B. Akesson *et al.* Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Proc. RTCSA*, 2008.

[3] B. Akesson *et al.* Composable resource sharing based on latency-rate servers. In *Proc. DSD*, 2009.

[4] B. Akesson *et al.* Composability and predictability for independent application development, verification, and execution. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, chapter 2. Springer, 2010.

[5] B. Akesson *et al.* Architectures and modeling of predictable memory controllers for improved system integration. In *Proc. DATE*, 2011.

[6] J. Ambrose *et al.* Composable local memory organisation for streaming applications on embedded MPSoCs. In *Proc. Computing Frontiers*, 2011.

[7] A. Beyranvand Nejad *et al.* A Hardware/Software platform for QoS Bridging over Multi-Chip NoC-Based Systems. *Parallel Computing*, 2013.

[8] A. Beyranvand Nejad *et al.* A Software-Based Technique Enabling Composable Hierarchical Preemptive Scheduling for Time-Triggered Applications. In *Proc. RTCSA*, 2013.

[9] G. Bilsen *et al.* Cyclo-static data flow. In *Proc. ICASSP*, 1995.

[10] M. Duranton *et al.* The HiPEAC Vision for Advanced Computing In Horizon 2020, 2013. [Online; accessed 09-Jun-2013].

[11] K. Goossens *et al.* Virtual Execution Platforms for Mixed-Time-Criticality Applications: The CompSOC Architecture and Design Flow. *SIGBED Review*, 2013. In press.

[12] S. Goossens *et al.* A Reconfigurable Real-Time SDRAM Controller for Mixed Time-Criticality Systems. In *Proc. CODES+ISSS*, 2013.

[13] A. Hansson *et al.* Trade-offs in the configuration of a network on chip for multiple use-cases. In *Proc. NOCS*, 2007.

[14] A. Hansson *et al.* CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM TODAES*, 14(1), 2009.

[15] A. Hansson *et al.* An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In *Proc. CODES+ISSS*, 2009.

[16] A. Hansson *et al.* The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proc. DAC*, 2010.

[17] A. Hansson *et al.* Design and Implementation of an Operating System for Composable Processor Sharing. *MICPRO*, 35(2), 2011.

[18] K. Keutzer *et al.* System-level design: orthogonalization of concerns and platform-based design. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 19(12), 2000.

[19] P. Kollig *et al.* Heterogeneous Multi-Core Platform for Consumer Multimedia Applications. In *Proc. DATE*, 2009.

[20] A. Molnos *et al.* Decoupled inter- and intra-application scheduling for composable and robust embedded mpsoc platforms. In *Proc. SCOPES*, 2012.

[21] A. Nelson *et al.* Composable power management with energy and power budgets per application. In *Proc. SAMOS*, 2011.

[22] B. Rumpler. Complexity Management for Composable Real-Time Systems. In *Proc. ISORC*, 2006.

[23] R. Stefan *et al.* dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-up. *IEEE Trans. Comput.*, 99, 2012.

[24] STMicroelectronics and CEA. Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology, 2010. White paper.

[25] S. Stuijk *et al.* SDF^3: SDF For Free. In *Proc. ACSD*. IEEE, 2006.

[26] S. Stuijk *et al.* Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proc. DAC*, 2007.

[27] Sundance. Flextiles SMT166 board. `http://www.flextiles.biz/product_info.php?products_id=140`, 2012. [Online; accessed 7-Jun-2013].

[28] C. van Berkel. Multi-core for Mobile Phones. In *Proc. DATE*, 2009.

[29] Xilinx. ML605 Documentation. `http://www.xilinx.com/support/#nav=sd-nav-link-140997&tab=tab-bk`, 2012. [Online; accessed 28-Mar-2013].