SoCLog: A Real-Time, Automatically Generated Logging and Profiling Mechanism for FPGA-based Systems On Chip

Ioannis Parnassos, Panagiotis Skrimponis, Georgios Zindros, Nikolaos Bellas
Center for Research & Technology Hellas (CERTH) & University of Thessaly, Greece
{ioparnas, skrimpon, zindros, nbellas}@inf.uth.gr

***Abstract*— Recent advances in FPGA technology and the proliferation of High Level Synthesis (HLS) tools makes it possible to implement complex System on Chip (SoC) designs that realize complete applications in a single FPGA device. To be able to exploit the large performance vs. area search space of such modern FPGA-based SoCs, system architects must have the appropriate performance analysis tools to evaluate-preferably at runtime-the computational requirements and the data flow of such a system to determine potential performance bottlenecks when running realistic workloads.**

In this paper we introduce SoCLog, a framework that automatically enhances the platform architecture with additional hardware components used to generate activity logs when the base SoC architecture executes an application. This real-time information can be analyzed by the system designer to expose performance bottlenecks not only on aggregate, but also at clock cycle granularity thus revealing potential design inefficiencies when there is burst of activity in the system. We evaluate our framework with a number of SoC designs to show that such a logging information can be valuable for the design of a complex SoC in a modern FPGA with minimal area overhead.

I. Introduction

In the last years, we have experienced the proliferation of FPGA High Level Synthesis (HLS) tools and methodologies both from the industry (e.g. Xilinx Vivado HLS, Altera OpenCL SDK) and from the academia [1][2][3]. HLS researchers have been investigating programming models based on marginally extending high level language such as C/C++ and OpenCL, and creating tools to compile application kernels to logic gates. Without any exception, those tools have a restricted view of the application, since they basically translate into hardware heavily executed kernels of the application, without interest on the global flow of data and synchronization.

Mapping multiple software kernels to hardware accelerators with appropriate interconnects and memory hierarchies to build a System on Chip (SoC) is still a manual effort and may involve a lot of trial-and-error iterations by a hardware designer. The designer has to balance multiple constraints in terms of performance, area and power requirements in an effort to find a global solution that meets user requirements.

In contrast, software programmers increase their productivity by using a plethora of tools for profiling and dynamic code analysis and visualization such as VTUNE for x86 [4] or Streamline for ARM processors [5]. No such dynamic analysis toolset exists for the hardware part of an SoC, i.e. the part that is implemented on the FPGA fabric.

Our paper proposes SoCLog, a hardware/software framework that automatically instruments the base SoC architecture with low area overhead hardware components that log activity information and present this information to the designer at runtime. Such activity includes utilization of hardware accelerators and peripherals, as well as activity of the interconnects such as buses and streaming channels.

We believe that such a toolset is important to understand the limitations of the platform and the sources of potential bottlenecks and optimize the design to meet specific performance requirements. For example, real time activity visualization reveals that the inverse DCT (iDCT) accelerator which consumes data from a producing DCT accelerator remains idle most of the time, even if DCT is utilized at peak rate (see section III). This prompts the hardware designer to re-design the DCT targeting higher throughput rate to alleviate performance mismatches. The novelties of the proposed framework are the following:

- Runtime capture and visualization of activity of the components of an SoC. In contrast, tools like VTUNE and Streamline work offline after code execution.
- Automatic generation of all hardware components required and easy access of all activity information at the application software level.

The rest of the paper is organized as follows. Section II describes the proposed SoCLog architecture and toolflow. Section III describes the experimental evaluation and section IV presents previous work. Section V concludes the paper.

II. SoCLog Design

SoCLog uses the RIFFA 2.0 framework to communicate data and profile information between software threads running on a CPU and the hardware SoC using the PCIe link [6]. RIFFA can open and sustain multiple communication channels between the CPU and the FPGA so that each channel can perform reads from and writes to the FPGA simultaneously using a simple streaming protocol. Using an extended RIFFA API, SoCLog extends this infrastructure to send log data collected at the FPGA back to the CPU whenever the programmer requests such information. In the next sections we describe the toolflow and hardware architecture of SoCLog followed by the extensions to the RIFFA software interface.

A. *Toolflow*

Figure 1 shows the SoCLog flow and the resulting SoC architecture which consists of two parts: a) the base architecture (in blue), and b) the profiling mechanisms (in green) which are automatically generated and attached to the base architecture without user intervention.

Figure 1 shows that the software kernel code(s) to be accelerated are synthesized to hardware using either an HLS tool (like Vivado HLS) or manually by a hardware designer. An Interface Generator attaches the distributed per-component Sampling and Trigger mechanism to each component. Finally, a System Generator aggregates the instrumented components with standard components like RIFFA, buses, memory controller, other peripherals (e.g. on-chip SRAMs) as well the System Manager. Some of these components will be described in detail in the next section. All this process is automated using a Vivado Tcl script.

B. *Hardware Architecture*

The Sampling and Trigger (S&T) mechanism is a parameterizable component that receives input control signals from the accelerator and the bus and generates triggers to the profiling mechanism when it detects the occurrence of an event that needs monitoring (see Figure 1). Events that are monitored include accelerator/peripheral utilization, bus transactions with producer-consumer granularity, and data transfers in the RIFFA channels.

To monitor accelerator/peripheral utilization, the S&T mechanism only requires a Start and a Done signal to be set high for at least one cycle to mark the beginning and end of the accelerator invocation. Note that S&T does not log events within an accelerator since this would presume knowledge of the internal accelerator microarchitecture.

Monitoring bus activity is slightly more complicated and requires signals to mark the beginning and end of AXI4 bus transactions. The signals WVALID and WREADY are used by the S&T component to detect the beginning of a transaction in the AXI4 Write Data Channel, and the corresponding WLAST to mark the end of the transaction. Likewise, for signals RVALID, RREADY and RLAST for the Read Data Channel. Finally, the S&T component use signals TVALID, TREADY and TLAST to monitor the AXI4-Stream.

Each time there is an event, the corresponding S&T mechanism generates a trigger pulse to the Profiler Logic to register the event. The Profiler Logic assembles trigger signals from all sources and registers their occurrence in an on-chip BRAM (Profiler BRAM) tagged with the appropriate time stamp. In fact, the Profiler Logic combines the beginning and the end of a transaction into a single BRAM entry allocating a pair of bits for this specific transaction: one bit for the Start event and one bit for the Stop event. The time stamp is generated by a 64-bit global counter at one cycle frequency. The counter is reset when the bitstream is downloaded in the FPGA and the SoC is ready to operate. The memory-mapped BRAM can be accessed by a user-level API (described in section II.C) to send profile back to the CPU memory.

Finally, the System Manager is used a) to orchestrate data movement between the RIFFA channels, accelerators and memories, b) to schedule the operations in the SoC conforming to data dependences, and c) to pull the profile data from the Profiler BRAM and send them back to the CPU memory through the RIFFA TX channel. The System Manager can be a microcontroller (e.g. Microblaze) running software to implement these functionalities. Alternatively, for simpler SoCs, the System Manager can be implemented in hardware as an FSM.

C. *Software Interface*

The SoCLog software interface extends the RIFFA 2.0

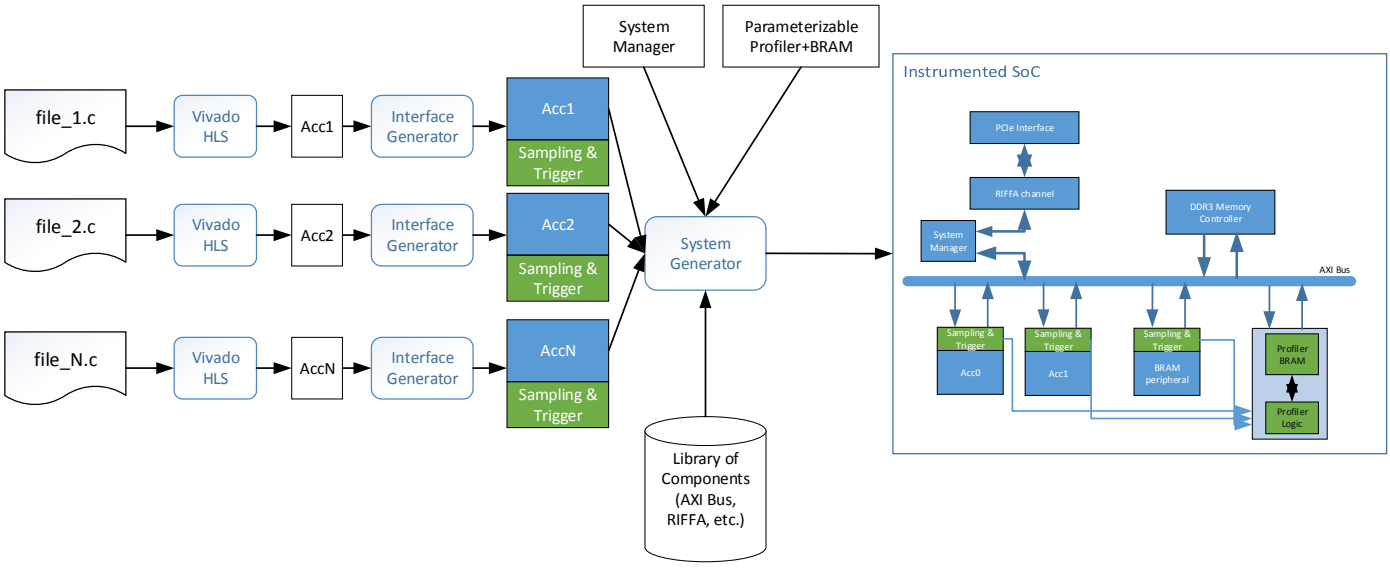


Figure 1. SoCLog toolflow and resulting architecture.

API with a function to receive profile data from the FPGA to the CPU memory. Table 1 shows the fpga_recv_log function which is used at the user C/C++ code to initiate transfer of logging data from the Profiler BRAM to the CPU memory buffer. There is a separate function used by RIFFA 2.0 API to receive actual data (fpga_recv at [6]) so that the System Manager can differentiate between the two types of transfers (i.e. the actual output of the computation and the logging data). This function is synchronous and blocks until the transfer is completed.

SoCLog supports the remaining RIFFA 2.0 API functions to open, close, transmit and receive data [6]. This is a simple interface based on transferring data to and from the FPGA using a streaming interface and assumes that that the SoC (and specifically the System Manager) knows how to access data from the correct addresses in the SoC memory and communicate them to the desktop CPU.

Table 1. SoCLog software C/C++ extensions with respect to RIFFA 2.0

API Name and Description
int fpga_recv_log (fpga_t *fpga, int chnl, void * data, long timeout) Receives log data from the FPGA channel chnl, to the CPU data buffer data. The fpga id is returned when initializing the connection. Timeout defines how long to wait for the transfer of the log data. The function returns the number of 4-byte words received.

III. SoCLog Experimentation

A. Benchmarks and Methodology

All evaluations were conducted using Vivado and Vivado HLS 2014.4. All designs were implemented and run on the VC707 evaluation board with a Xilinx XC7VX485T (Virtex-7) FPGA. We have tested SoCLog for three application benchmarks shown in Table 2, and we will present the results in this section focusing on how we optimized the DCT/iDCT performance using SoCLog. In each case, the clock frequency is 250 MHz.

Monte Carlo (MC) was developed to provide an alternative method of approaching multi-domain, multi-physics problems. It breaks down a single Partial Differential Equation (PDE) problem to produce a set of intermediate problems. The core kernel of this application performs random walks from initial points in a 2D grid to estimate the boundary conditions of the intermediate problems. MC is heavy in double FP operations and also massively parallel allowing for multiple accelerators to execute in parallel.

The *Blur* image processing algorithm first applies a horizontal and then a vertical 3-tap low pass filter to an incoming image, temporarily storing the output of the horizontal filter to the main memory. This code results into two hardware accelerators, which have to communicate via a large memory implemented either as an external DRAM or as an on-chip BRAM (if there is enough BRAM in the FPGA).

DCT/iDCT applies the Discrete Cosine Transform (DCT) to an input image to produce 2D frequency coefficients. Quantization and inverse Quantization compress the range of coefficient values to a narrower range of values. Finally, inverse DCT (iDCT) reconstructs a lossy version of the initial image using the output of the inverse Quantization kernel. DCT/iDCT is applied to 8x8 blocks of pixels to facilitate data locality and allow temporary data storage within an accelerator. This application is organized as a pipelined SoC consisting of three accelerators in which the output of an accelerator is fed via the AXI4 bus to the input of the following accelerator.

Table 2. Benchmarks used for SoCLog evaluation

App.	Description	Input Set
Monte Carlo	Monte Carlo simulations in a 2D space	120 Points, 5000 Walks per point
Blur Filter	Blur 2D filter	4096×4096 image
DCT/iDCT	Discrete Cosine Transform (DCT), Quant/iQuant, inverse DCT	512x512 image in sequences of 8x8 blocks

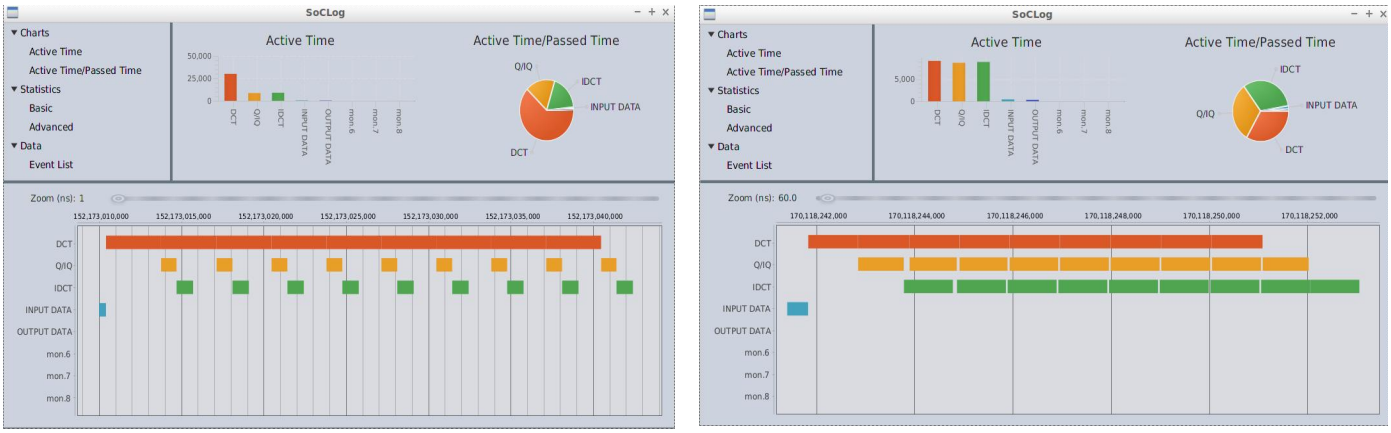
B. Experimentation

Fig. 2 which shows a snapshot of the SoCLog GUI indicates how SoCLog can be used in a realistic scenario to assist a platform designer in detecting system imbalances and optimize for throughput. The initial design shown in Fig. 2a includes a DCT accelerator which requires 839 cycles to produce an 8x8 block output feeding the faster accelerators Q/iQ and iDCT. Such mismatch results in under-utilization of the iDCT which is only used 30% of its peak rate. By iteratively unrolling and pipelining the DCT loops (using pragmas #HLS UNROLL and #HLS PIPELINE of Vivado HLS), the designer can reduce execution time to 255 cycles per block, thus closing the performance gap and tripling the effective throughput (Fig. 2b). In both cases, overhead for data transfers is negligible, thus there is no need to optimize data accesses.

Table 3 shows resource utilization for selected cases for the three benchmarks of Table 2. By default, each SoC includes the PCIe Endpoint and RIFFA 2.0 hardware, a DDR3 memory controller and the AXI4 bus. The main take away from this Table is the minimal overhead of the logging mechanism.

The Monte Carlo benchmark shown in Table 3 includes a single accelerator which executes all random walks (5000 in our case) for a single point each time it is invoked. MC includes a lot of complex math functions such as trigonometric operators (sin, cos) which are not pipelined. Moreover, MC kernel computations in a single walk are sequential and cannot be parallelized. All walks from one point require 97,000 cycles.

The DCT/iDCT benchmark numbers refer to the optimized version of Fig. 2b.



(a) (b)

Figure 2. SoCLog visualization showing DCT/iDCT activity for two cases: (a) when DCT accelerator is slower than iDCT, and (b) when both accelerators are fully optimized for performance and execute in approximately the same number of cycles. The Active Time panels at the top show the aggregate time (and percentage) each component is utilized. We do not show the activity of bus transactions since it is negligible compared to the computations. INPUT_DATA monitors the transfer of the 512x512 image from RIFFA channel 0 to main memory. Note that this figure is a snapshot: in SoCLog data are updated continuously as they are read out of the FPGA by the user.

Table 3. Benchmark SoC resource utilization in absolute numbers and as percentage of available resources in XC7VX485T. Both with and without the SoCLog overhead (second column). These numbers include PCIe Endpoint and RIFFA hardware for one channel. The SoCLog BRAM is 8K x 128 bits.

SoC	SoCLog	LUTs	FFs	BRAMs	DSPs
Monte Carlo	-	42725 (14.1%)	55451 (9.1%)	79 (7.6%)	130 (4.6%)
	+	43804 (14.4%)	56335 (9.3%)	111 (10.7%)	130 (4.6%)
Blur Filter	-	19986 (6.6%)	26281 (4.3%)	107 (10.3%)	32 (1.1%)
	+	21189 (7%)	27464 (4.5%)	138 (13.4%)	32 (1.1%)
DCT/iDCT	-	28510 (9.4%)	41821 (6.9%)	84 (8.1%)	467 (16.7%)
	+	29826 (9.8%)	42870 (7.1%)	116 (11.2%)	467 (16.7%)

IV. Prior Work

Tools that capture the run time activity of a system have attracted both academic and industrial research. VTUNE [4] and Streamline [5] are commercial tools that perform offline analysis of software code running in a CPU. FPGA hardware debug tools are used to probe internal FPGA signals and monitor their activity at run time [7]. Their functionality is geared towards hardware error detection and not performance analysis.

V. Conclusion

We have presented SoCLog, a software and hardware infrastructure to automatically generate logging and profiling information for FPGA-accelerated applications. SoCLog instruments a base SoC architecture with low-overhead hardware components that capture system-level activity and present the log output visually at run time. We showed how the logging information is used by an FPGA platform designer to analyze potential system level bottlenecks and accordingly optimize the performance of individual components.

An interesting extension of our work is to link SoCLog with software performance analyzers like ARM's Streamline [5] to acquire a realistic understanding of hardware-software interactions in a complex processor-based SoC. This approach can provide a much more intuitive strategy for single-component HLS optimizations, workload partitioning and data management in complex embedded systems than resorting only to separate CPU-based profiling and hardware simulations as most platform designers currently do.

References

[1] Alexandros Papakonstantinou et al. "*FCUDA*: Enabling efficient compilation of CUDA kernels onto FPGAs. Proceedings of the 7th Symposium on Application Specific Processors, pp.35-42, July, 2009, Boston, MA

[2] Andrew Canis et al. "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems", Proceedings of the IEEE International Symposium on Field Programmable Gate Arrays (FPGA), pp. 33-36, February 2011, Monterey, CA

[3] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, Christos D Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), May 1-3, 2011, Salt Lake City, UT

[4] Intel VTUNE Amplifier. Available at https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[5] ARM Streamline Performance Analyzer. Available at http://ds.arm.com/ds-5/optimize/

[6] M. Jacobsen, and R. Kastner. "RIFFA 2.0: A reusable integration framework for FPGA accelerators." 23rd International Conference on Field Programmable Logic and Applications (FPL), 2013.

[7] Vivado Hardware Debug. Available at http://www.xilinx.com/products/design-tools/vivado/debug