2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines

An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures

Jens Korinth, David de la Chevallerie, Andreas Koch
Embedded Systems and Applications (ESA)
Technical University of Darmstadt
Darmstadt, Germany
Email: {jk, dc, ak}@esa.cs.tu-darmstadt.de

***Abstract*—**

With heterogeneous parallel computing becoming more accessible from general-purpose languages, such as directive-enhanced C/C++ or X10, it is now profitable to exploit the highly energy-efficient operation of *reconfigurable* accelerators in such frameworks. A common paradigm to present the accelerator to the programmer is as a pool of individual threads, each executed on dedicated hardware. While the actual accelerator logic can be synthesized into IP cores from a high-level language using tools such as Vivado HLS, no tools currently exist to automatically compose multiple heterogeneous accelerator cores into a unified hardware thread pool, including the assembly of external control and memory interfaces. ThreadPoolComposer closes the gap in the design flow between high-level synthesis and general-purpose IP integration by automatically composing hardware thread pools and their external interfaces from high-level descriptions and opening them to software using a common API.

***Keywords*-FPGA; hardware thread pools; architecture; design automation; accelerators; meta flow; Zynq;**

I. Introduction

Modern parallel programming languages such as X10, Chapel, C/C++ with OpenACC directives, and others are evolving from *homogeneous multi-core computing* to *heterogeneous computing* by integrating *accelerator devices* to which suitable parts of the computation can be efficiently offloaded. But FPGAs, DSPs, and GPGPUs use vastly different models of computation, and require additional abstractions for seamless integration. One possible abstraction is the *thread pool*, i.e., a collection of processing elements executing independent threads. Viewing an FPGA-based compute unit as *hardware thread pool* has been investigated in different contexts in the literature (see, e.g., [1], [2], [3] or [4]). However, less attention has been paid to tool flows that automatically construct such thread pools from high-level descriptions, including all required interfaces and support blocks. ThreadPoolComposer is an extensible, highly customizable open-source tool flow that generates *complete thread pool designs*, including customizable memory and control infrastructure, and also encompasses a generic API hierarchy to achieve re-use and portability.

II. Related Work

System-level integration of hardware threads has been an active research topic for several years now, with many different approaches having been proposed in the literature. The solution in [1] is based on a OS-level API for hardware threads called *Hthreads*. The threads are compiled from C code into intermediate form called HIF, from which simple non-pipelined hardware accelerators are created in VHDL. The ReconOS project [2] unified software and hardware threads by defining software proxy threads for hardware, and integrated both in a real-time operating system. Similarly, in the FUSE framework [3], Ismail et al. also proposed a common interface for hardware and software threads, making the locality of execution transparent to the application. Other efforts, such as SPREAD [4], discuss topics such as a partially reconfigurable system architecture for streaming-based computation

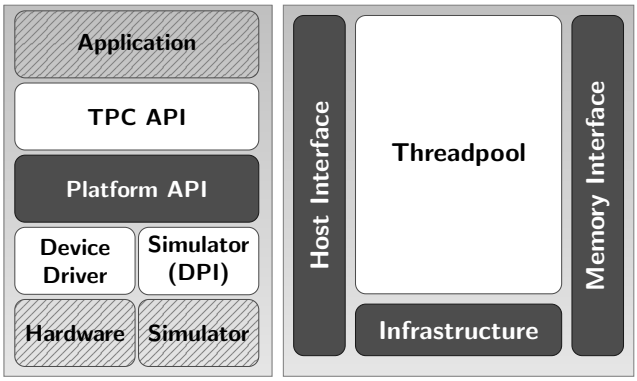
Our approach is probably most similar to FUSE [3] and ReconOS [2], but with a different focus. We are interested in performing automated design space exploration for heterogeneous thread pool architectures, and thus require tool support to automatically assemble a wide range of system implementations, including support logic (memory and interrupt controllers, host interfaces etc.), to perform area/time/power trade-offs on real or simulated hardware. We provide a flexible API for integrating the created thread pools into higher-level run-times (such as HSA, pocl for OpenCL, or FastFlow [5]).

III. ThreadPoolComposer

In this section we briefly describe the foundations upon which HW/SW systems using ThreadPoolComposer thread pools can be built. Throughout this paper, a *hardware thread* denotes an IP core, generated from a software kernel by high-level synthesis, that is capable of independent execution; it can receive input arguments and returns output results.

Figure 1 shows the proposed API hierarchy in *(a)*, and its relation to the overall hardware architecture in *(b)*. The latter consists of four parts: host and memory interfaces, support

978-1-4799-9969-9/15 $31.00 © 2015 IEEE
DOI 10.1109/FCCM.2015.22



(a) API hierarchy (b) Hardware Design Structure

Figure 1: API Hierarchy and Structure

infrastructure (e.g., interrupt control), and the actual, device-independent thread pool architecture.

A. TPC API

The fundamental interface to the thread pools built by ThreadPoolComposer is the TPC API: It provides methods to enumerate available kernels, manage device memory, perform data transfers (synchronously and asynchronously), define jobs (i.e., sets of parameters for a single kernel execution), actually launch jobs, and wait/poll for their completion. Listing 1 shows a tiny example of preparing and executing a job using the TPC API.

```
/* allocate 1 KB on device */
tpc_handle_t h = tpc_device_alloc(dev, 1024);
/* copy array 'data' to device */
tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);
/* prepare a new job for function id #10 */
tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, 10);
/* set argument #0 to handle h */
tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
/* launch job */
tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);
/* call blocks until completed, so get return value */
int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
printf("result of job: %d\n", r);
/* release job id */
tpc_device_release_job_id(dev, j_id);
```

Listing 1: Threadpool API Example

Furthermore, TPC API provides some optional management functions, e.g., to enumerate devices, reconfigure the device, and perform initial setup (e.g., of memory ranges shared with the host). Note that no specific implementation is assumed. Consider, for example, the job scheduler: The TPC API contains a generic function to launch a job, but the actual thread scheduling could happen in the implementation of **`tpc_device_job_launch`**, or at device driver level, or even in custom IP on the FPGA itself. We have actually done the latter for our current target platform [6], significantly reducing scheduling latency.

B. Platform API

To cleanly separate the device-dependent from the device-independent view, TPC API is itself implemented against the Platform API. The core tasks of the Platform API are to manage device buffers (**`platform_alloc/dealloc`**), provide access to the hardware registers (**`platform_read/write_ctl`**) and device memory (**`platform_read/write_mem`**), and to offer a general waiting mechanism (**`platform_write_ctl_and_wait`**).

We aim for maximum flexibility using this abstraction, e.g., register and memory could actually occupy a common address space, or a platform might not employ a separate register space at all and rely entirely on shared memory communication. Waiting could be implemented using polling or interrupts (or even a combination of the two). This interface is sufficiently flexible to accommodate all data transfer mechanisms examined in [7].

C. Simulator Target

TPC API does not assume anything about the nature or capabilities of the lowest layer used to implement the Platform API, which will likely be the device driver on most platforms.

For ease of debugging, it is very useful to implement Platforms in two modes: A regular device driver implementation wrapping the actual hardware, and an additional SystemVerilog DPI (Direct Programming Interface) simulator harness which implements the Platform API against an RTL simulation model. In this manner, the hardware simulation can be driven by the actual application without recompilation, allowing efficient hardware/software co-design. To support this, ThreadPoolComposer provides client/server libraries using UNIX sockets for IPC and includes a sample harness for the Zynq reference platform on the Mentor ModelSim/Questa simulator.

D. Flow Overview

The overall ThreadPoolComposer tool flow can be divided into three stages of *high-* (behavioral), *mid-* (system composition) and *low-level* (logic and layout) synthesis (HLS/MLS/LLS): We assume that the developer has already partitioned the application into per-thread kernels suitable for the high-level synthesis tool selected for the first stage, e.g., isolated C/C++ kernels for Nymble [8] or Vivado HLS. A thread pool hardware design is composed in three steps:

1) The HLS tool generates a behaviorally equivalent IP core with an interface which is defined by the selected *Architecture*, e.g., the **`baseline`** architecture generates an AXI4Lite register file for control and value arguments, and AXI4 master(s) for reference-arguments.
2) The MLS tool instantiates each IP core the user-specified number of times and creates the necessary intra thread pool infrastructure, as defined by the selected *Architecture*; e.g., the **`baseline`** architecture

generates an AXI4 Interconnect hierarchy to connect the AXI4Lite slaves and AXI4 masters to the platform.

3) Finally, the MLS tool instantiates the base design, i.e., the device-dependent infrastructure as defined by the selected *Platform*, and connects it with the thread pool architecture created in the previous step. E.g., the `zynq` platform instantiates the Xilinx Processing System 7 IP Core and AXI Interrupt Controllers, while the `vc709` platform instantiates the PCIe core and DMA engine subdesign.

The resulting hardware design can either be synthesized into a bitstream by the LLS tool, or used in simulation via the SystemVerilog DPI platform layer. ThreadPoolComposer is based on Scala/SBT and controlled by simple key-value configuration files; *Architecture* and *Platform* implementations consist mostly of template files which control, e.g., hardware interface generation, and software libraries implementing the TPC API and Platform API, respectively. Software applications are written against TPC API, and can be used without change or recompilation on different platforms simply by linking against the appropriate `libtpc` and `libplatform`.

IV. Case Study: Zynq-7000 Platform

To demonstrate the use of ThreadPoolComposer, we investigated the Xilinx Zynq-7000 series system-on-chip (SoC). The XC7Z045(-2) device we used integrates a dual-core ARM Cortex A9 CPU (called *Processing System, PS*) with a reconfigurable fabric, communicating over AMBA AXI3 interfaces:

- *general purpose* ports GP0-1 have masters communicating from the PS to the fabric, and slave ports communicating from the fabric to main memory
- *high performance* slave ports HP0-3 communicate from the fabric to main memory
- *application coherent port* ACP, which allows cache-coherent access from the fabric to shared main memory

The `baseline` *Architecture* generates AXI4Lite register sets for control and value arguments of each kernel, and AXI4 master interface(s) for the reference arguments. Our prototypical `zynq` *Platform* implementation uses GP0 to connect these register files to the host, and GP1 to control up to two instances of the AXI Interrupt Controller IP (from the Xilinx IP Catalog). We assume *thread-private memory access* in this evaluation, therefore the AXI4 masters are connected to HP0-3 (as coherency via ACP is not required). This simple environment provides a minimal, but already useful baseline for ThreadPoolComposer on Zynq.

V. Evaluation

A. Environment and Kernels

Our prototypical implementation of ThreadPoolComposer uses Vivado HLS 2014.4, the mid-level synthesis is based on the Tcl scripting interface of the Vivado IP Integrator, and the low-level step (logic synthesis, mapping, place & route) also relies on Vivado tools. To evaluate the designs generated by ThreadPoolComposer, we used the *MachSuite* benchmark set [9], which is designed specifically to exercise high-level synthesis systems, and and evaluate the resulting accelerator architectures. We are able to process all examples accepted by Vivado HLS, but (for space reasons) will focus the following discussion on the MachSuite *sort/merge* benchmark.

B. Evaluation

On our prototypical `zynq` *Platform*, we can support up to 48 hardware threads of the *sort/merge* kernel using ThreadPoolComposer, and aim for the 250 MHz supported as maximum clock frequency on the ZC706 board. Table I shows the actually achieved operating frequency F_{max}, the resources (both absolute as well as relative to the XC7Z045 device), and the overhead of the total architecture area vs. that of the actual thread cores.

sort/merge is one of the more complex kernels we examined from MachSuite. As Figure 2 shows, it easily scales from 1 to 48 hardware threads, always exceeding the 100 MHz system clock frequency commonly used on the ZC706 platform. At up to 24 instances, the thread pool array could even execute double-pumped at 200 MHz frequency. When scaling-up the number of instances, the size of the support infrastructure (interfaces, signaling, etc.) also begins to grow, but this is easily amortized for more complex thread cores: A thread pool consisting only of a single thread core has a support overhead of 58%, which drops to just 36% for a fully loaded thread pool with 48 hardware thread cores.

VI. Conclusions and Future Work

We have demonstrated how ThreadPoolComposer can be employed to provide a seamlessly automated flow from high-level language programming to system-level synthesis of hardware thread pools. The evaluation shows that ThreadPoolComposer-created systems-on-chip are scalable, as even pools of 48 cores do not slow down below the 100

#	F_{max}	LUTs	%	Regs.	%	Overh. %
1	250	4155	1.9	4862	1.1	57.6
4	250	12738	5.8	14646	3.4	44.6
8	200	23522	10.8	27171	6.2	40.0
12	200	34631	15.8	39750	9.1	38.9
16	200	45620	20.9	52299	12.0	38.2
20	200	56976	26.1	65801	15.1	38.1
24	200	67963	31.1	78362	17.9	37.7
28	150	78972	36.1	90910	20.8	37.5
32	150	88876	40.7	103436	23.7	36.5
36	150	100454	46.0	117108	26.8	36.8
40	150	111312	50.9	129669	29.7	36.7
44	150	122510	56.0	142223	32.5	36.7
48	150	132813	60.8	154763	35.4	36.3

Table I: Evaluation of sort/merge benchmark

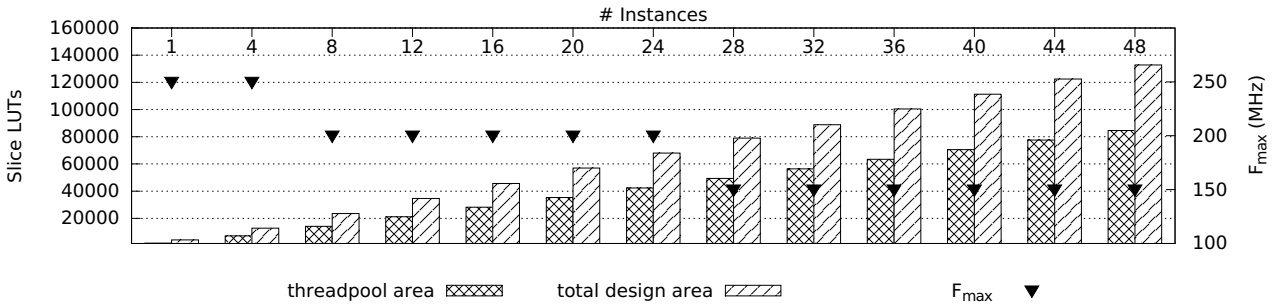


Figure 2: Detailed analysis of sort/merge benchmark

MHz typically employed as the reference clock frequency of Zynq-7000 devices.

ThreadPoolComposer will be available as open-source at [10], with the initial version supporting the Zynq-7000 SoC Platform. In the future, we aim to support more powerful PCI Express Gen3-attached platforms, with porting to the Xilinx VC709 board already being in progress. On the software side, we will integrate the hardware thread pools with the FastFlow run-time [5] for heterogeneous parallel execution. ThreadPoolComposer itself will be extended to support automated customization of the thread pool support logic and architecture exploration.

ACKNOWLEDGMENT

This work was performed in the context of "REPARA – Re-engineering and Enabling Performance and poweR of Applications" [10], a *Seventh Framework Programme* project of the European Union.

REFERENCES

[1] S. Ma, M. Huang, and D. Andrews, "Developing application-specific multiprocessor platforms on FPGAs," *Reconfigurable Computing and FPGAs (ReConFig), Int. Conference on*, 2012.

[2] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, 2014.

[3] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," *Field-Programmable Custom Computing Machines (FCCM), IEEE 19th Ann. Int. Symposium on*, 2011.

[4] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model," *Very Large Scale Integration Systems (VLSI), IEEE Transactions on*, 2013.

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, 2014.

[6] D. de la Chevallerie, J. Korinth, and A. Koch, "Integrating FPGA-based Processing Elements into a Runtime for Parallel Heterogeneous Computing," in *Field-Programmable Technology (ICFPT), International Conference on*, 2014.

[7] J. Kelm and S. Lumetta, "HybridOS: runtime support for reconfigurable accelerators," *Field Programmable Gate Arrays (FPGA), Proceedings of the 16th International ACM/SIGDA symposium on*, 2008.

[8] J. Huthmann, B. Liebig, and A. Koch, "Hardware/software co-compilation with the Nymble system," *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 8th Int. Workshop on*, 2013.

[9] B. Reagen, R. Adolf, Y. Shao, G. Wei, and D. Brooks, "MachSuite: Benchmarks for Accelerator Design and Customized Architectures," *Workload Characterization (IISWC), IEEE International Symposium on*, 2014.

[10] "REPARA - Reengineering and Enabling Performance and poweR of Applications," 2013. [Online]. Available: http://www.repara-project.eu

[11] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," *Design Automation Conference (DAC), 49th ACM/EDAC/IEEE*, 2012.

[12] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. S. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems," *Embedded Computing Systems (TECS), ACM Transactions on*, 2013.

[13] T. Oguntebi, S. Hong, J. Casper, N. G. Bronson, C. Kozyrakis, and K. Olukotun, "Farm: A prototyping environment for tightly-coupled, heterogeneous architectures," *Field-Programmable Custom Computing Machines (FCCM), 18th Annual Int. Symposium on*, 2010.

[14] Y. Wang, J. Yan, X. Zhou, L. Wang, W. Luk, C. Peng, and J. Tong, "A partially reconfigurable architecture supporting hardware threads," *Field-Programmable Technology (ICFPT), International Conference on*, 2012.