Integrating FPGA-based Processing Elements into a Runtime for Parallel Heterogeneous Computing

David de la Chevallerie, Jens Korinth and Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt, Darmstadt, Germany
Email: {dc, jk, ak}@esa.cs.tu-darmstadt.de

Abstract—**In this work, we present an approach how FPGA-based computing can be integrated into a heterogeneous computing environment in an embedded systems context, using the `x10rt` run-time of the X10 language system as a case-study. To this end, we present a hardware/software framework for pools of reconfigurable compute elements, and show how high-level synthesis can be employed to generate the actual processing cores. Our framework is sufficiently lean to deliver high performance FPGA implementations even at high area utilization (operating at 250 MHz with up to 90% of the device area used), and capable of low-latency access to pools of dozens of instances of custom IP cores, automatically generated by high-level synthesis tools.**

I. Introduction

A wide variety of computing architectures exists today (multi-core CPUs, GPGPUs, DSPs, FPGAs, ...), each offering specific advantages and disadvantages, such as massively parallel high-speed computation, reduced energy consumption, or low cost. At almost every scale, applications can benefit from taking advantage of a *combination* of these architectures. *Runtime systems* of modern programming languages for parallel heterogeneous computing act as glue between the processing elements, orchestrating both computation as well as data transfer over a number of communication links. In this paper, we present and evaluate an approach to exploit reconfigurable computing in the low-level runtime of the X10 language, `x10rt`.

II. X10RT

The *X10 programming language*, first introduced in [?], is based on the *Asynchronous Partitioned Global Address Space (APGAS)* model [?]. APGAS is a variant of *PGAS*, a programming model in which all memories available to the programmer are represented in a uniform address space, albeit with non-uniform access times. The PGAS model is well-suited to represent non-uniform memory hierarchies found on accelerator devices such as GPGPUs, DSPs and FPGAs, as well as distributed systems [?]. The PGAS notion of explicit *locality of data* is extended in the APGAS model by *locality of execution*, which permits both locally and remotely executed asynchronous *tasks*. APGAS is implemented by X10 using a tree-structured hierarchy of *Places*, where a Place represents a locale of execution and its associated storage. The communication between places is handled using message-passing by a low-level runtime called `x10rt`, which abstracts from the underlying communication layer(s).

III. Reconfigurable Compute Unit Architecture

The architecture, shown in Figure 1, includes an *Application Processor Unit (APU)* and a reconfigurable area, called *Programmable Logic (PL)*. The APU will execute the software parts of the application (and the `x10rt` runtime itself), while the Vivado HLS-generated hardware *kernels* are executed on the PL. Two main communication mechanisms exist between APU and PL: Communication initiated and controlled by the APU via an AXI-Lite interface, or by the PL via an AXI master-mode interface. Since in our use-case, the controlling instance is `x10rt` executing on the APU, we will use the the first model, aiming for low-latency communication from the APU to the PL. Logically, the PL is offered to the run-time as a *Reconfigurable Compute Unit* (RCU). The RCU provides a number of (generally different) *Functions*. Each *Function* implements one or more *Instances* of the hardware kernel. All Instances may execute in parallel, with hardware schedulers in the PL distributing processing *Jobs* received from the run-time to the actual Instances. Jobs may complete in any order and are identified across hardware and software by a unique Job ID.

A. Hardware Interface

The RCU is controlled by the APU using memory-mapped registers, organized into Global and Function address ranges. The first controls the overall behavior of the RCU (e.g., blocking/non-blocking operation modes, interrupt status), while the second allows interaction with individual Functions. Each Function accepts a new Job from the APU, when the parameters (Cmd, ID, Payload) have been written to the Function's appropriate registers. After Job completion the APU is notified (see Sec. III-B) and can then retrieve the computation results. The APU indicates completion of the result retrieval by clearing the Function's Command/Status register, allowing the next set of RCU results to move into the registers. Each Function is controlled by a function control unit, which handles Job scheduling and distribution/collection of parameters and results to/from kernel Instances. Since all kernels within a Function are identical, the scheduler distributes the incoming Jobs from their queue to the first available Instance (starting at Instance 0). Similarly, completed Jobs and their results are enqueued for retrieval by the APU. If multiple Instances complete at the same time, their results will be collected in the order of ascending Instance numbers.

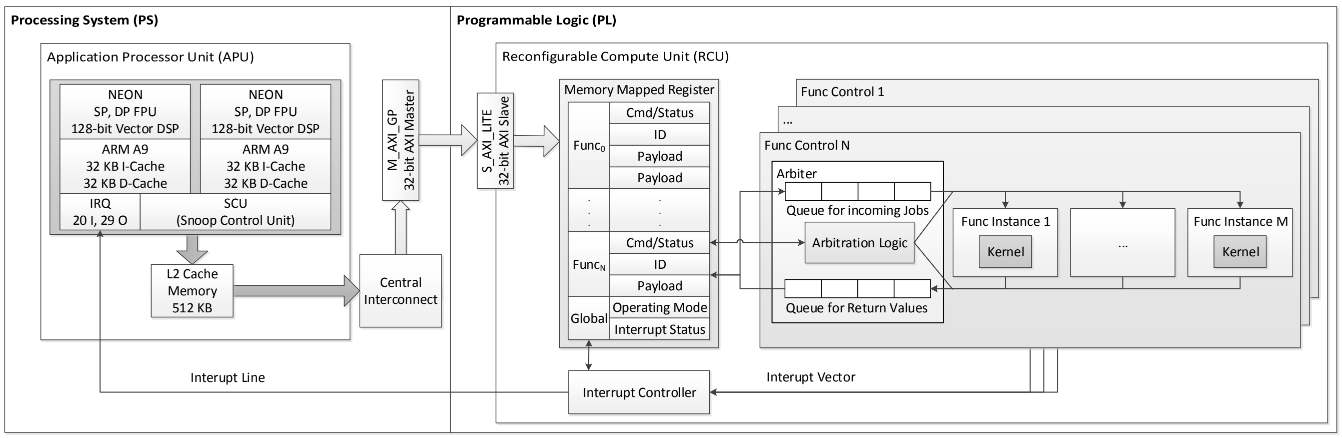

Fig. 1. Hardware Architecture

Notifying the APU of completed jobs can be performed by using one of two mechanisms: In *blocking* mode, the software thread issuing the job sleeps until the RCU issues an interrupt to announce job completion. In *non-blocking* mode, the software thread remains awake and has to explicitly poll the Function's Job ID register to determine whether a Job completed in the meantime.

B. Software Interface

On the software side, the mechanisms described above are realized by a Linux device driver. Every Function is represented as a node in /dev in order to gain access from user-space. Extra entries in /sysfs deliver necessary meta-information (e.g., the number of Functions and which kernel they implement). Three main communication mechanisms have been implemented for fast integration into different use-cases. Thread-safety is ensured by the driver for the first two approaches. **Inter-process shareable, blocking**: The user process prepares the data and issues an **ioctl**. The driver automatically generates a Job ID, transfers parameters, and starts the Function. The calling process will be put into sleep until the RCU signals completion of that specific Job ID. The driver awakes the corresponding process and delivers its results. **Inter-process shareable, non-blocking**: Here, the *user process* has to explicitly generate the Job ID, calls **write** and is free to continue. It has to probe the Job ID queue itself using **read** to retrieve the ID of a completed job. If the ID matches the expected one, **read** also delivers used the result data. The **count** parameter is used to distinguish between both modes (poll Job IDs, retrieve result data). **Non-shareable, non-blocking**: The hardware registers are directly made visible to one or more processes by the **mmap** system call. This allows the lowest latency access, since no context-switches are required. Only one process may write to the registers, but multiple processes may safely read them (this will be exploited, see Sec. IV).

IV. X10RT/RCU Interface

For our integration of FPGAs in x10rt (which will subsequently be called x10rt_fpga) we follow the example of the GPGPU integration from [?], but with the major extension that we allow out-of-order completion of Jobs: In x10rt_fpga, FPGA boards are represented by an accelerator Place in x10rt, which is hierarchically located below its containing PC's CPU Places (or, in case of a SoC, the Places of its embedded CPU). In this manner, every local CPU place can directly access the accelerator. The communication endpoint **x10rt_send_msg** is implemented using an additional API layer called RCU API, which abstracts from the device driver interface and provides methods to transfer a job to the hardware, to check return value availability, and to fetch and acknowledge return values. RCU API can be configured to use either the file system based approach (safe for concurrent access by multiple processes), or the lower-latency approach using **mmap**. As an optimization, the presence of completed Jobs is detected in both cases by the low-latency **mmap**-mechanism, since concurrent access to read-only registers is safe in any case. The implementation of **x10rt_send_msg** for accessing the RCU from x10rt calls the RCU API method to start a Job on the device, and then inserts the ID of the launched Job into an internal map. The latter allows the tracking of the out-of-order completion of different Jobs. Each call to the x10rt_fpga implementation of **x10rt_probe** polls a single Function (with multiple calls iterating over Functions in a round-robin fashion) and checks for the completion of a Job using an RCU API call. If a completed Job is found, the result data is retrieved and the Job marked as handled using another RCU API call.

V. Experimental Results

The Xilinx ZC706 evaluation board was used for the experimental evaluation. The hardware framework and the kernels are created using Vivado 2014.1, the framework modules are formulated in synthesizable Verilog HDL, and the kernels are written in C and compiled using Vivado HLS.

TABLE I
KERNEL RUNTIME BOUNDS FOR SINGLE JOBS

Benchmark	Min (ms)	Avg (ms)	Max (ms)
add	0.02	0.56	1.34
gol	0.02	0.22	1.83
jos	0.24	1,186.19	3,247.12
mnk	0.13	0.34	230.87
nqs	0.44	50.81	24,443.85

TABLE II
DEVICE UTILIZATION FOR DIFFERENT DEGREES OF PARALLELISM

Kernel	#Funcs.	#Insts.	LUT [%]	FF [%]	BRAM [%]	DSP [%]
mnk	1	1	11	6	1	3
mnk	3	3/3/3	94	50	5	30
framework	3	3/3/3	4	3	-	-
jos	1	1	9	5	-	1
jos	3	4/3/3	84	44	-	12
framework	3	4/3/3	3	3	-	-
nqs	1	1	1	1	1	-
nqs	3	15/15/15	25	15	29	-
framework	3	15/15/15	3	3	-	-

A. Benchmarks

To evaluate the x10rt integration we used five different test cases: The kernel add is a one-cycle 32-bit adder, the corresponding x10rt test case simply computes 100000 additions; it is mainly used to evaluate the overheads incurred by the framework (as its own execution time is negligible). gol is a simplified approach to the *Game of Life*: the kernel computes the next state of a single cell, given its neighborhood. The jos kernel solves an instance of the Josephus problem, using a while-loop approach, the corresponding test case iterates over 30 different instances with different runtimes. The resulting IP core uses 64-bit division units, requires significant chip area (see Table II for details), and has been selected because it is strictly sequential and profits little (if at all) from spatial parallelism (which is a worst case for HLS tools *and* our framework). The nqs kernel solves the N-Queens problem; the problem itself benefits significantly from spatial parallelism and lends itself well to FPGA implementation. Finally, the mnk kernel computes statistical information about an (m, n, k)-game instance: The test case computes the total number of possible games (i.e., sequences of turns), the number of games that end in draws, and the number of games that end in wins for Player 1 or 2, respectively, using brute force (counting). This test case was selected since it principally benefits from spatial parallelism, but the code itself is not well suited for HLS tools. Furthermore, mnk is the largest kernel considered in this work, with only nine instances already using $\approx 94\%$ of the ZC706 FPGA's LUTs (see Table II).

For reference, Table I shows typical kernel runtimes in milliseconds, including operating system overhead. Note that jos, nqs, mnk have input-dependent runtimes.

B. Hardware Performance

We examine the spatial scalability of the system by increasing the number of instances, while still maintaining a PL clock rate of 250 MHz. For all of the experiments, we use between one and three functions, each controlling 3 to 15 Instances of a single kernel. Table II shows the relative device utilization. With a total of 9 Instances organized into three Functions, mnk almost completely fills the device with respect to LUTs. Note that in all cases, our framework itself requires only 3…4% of the chip area, regardless of whether 10 or 45 instances are used. To examine the temporal scalability, we consider an RCU holding 10 Instances of the nqs kernel, organized in a single Function. We will compare calls with a short hardware execution time of $11\mu s$ with calls requiring

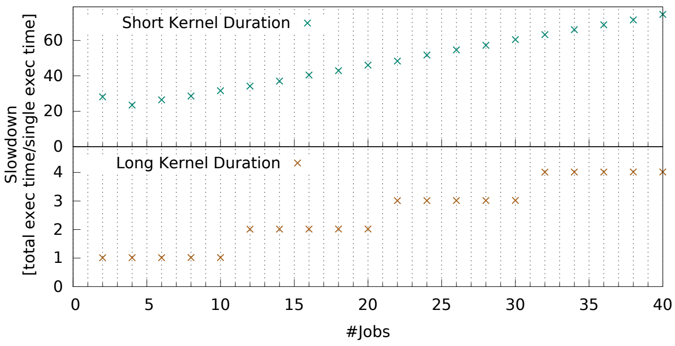

Fig. 2. Execution times relative to job duration and system load

long execution time $18,495\mu s$. One thread launches new Jobs using **write** as quickly as possible, the second one retrieves the results using **read**. Each run of the benchmark was executed 100 times to reduce the measurement error, with Figure 2 reporting the minimal execution times. The Y-axis shows the complete execution time (software-hardware-software) per Job, normalized to the hardware execution time t_{kernel} of performing a single nqs kernel (short or long) in the system. The X-axis shows the number of Jobs launched for the specific run. It takes $\approx 240\mu s$ to launch a single Job from user-space to actually starting the computation on an Instance. Short Jobs are thus finished faster than new Jobs can be spawned. As a result, only a single Instance will actually be used and we cannot achieve any gains by spatial parallelism. The framework's overhead becomes negligible, when executing long nqs Jobs. Jobs are launched faster than they complete, and thus begin to execute in parallel. Once the configured number of instances on the RCU (10 in our example) is exceeded, Jobs begin to execute sequentially. This leads to the stair shape at the bottom of Figure 2.

C. System-Level Performance

To evaluate the performance at the system level, overall wall-clock execution time of the benchmarks was measured. x10rt itself does not implement actual concurrency support (which is provided by the higher-level X10 language runtime). For our experiments, we used a custom, minimal thread pool implementation based on the Pthread library to utilize all available CPU cores. Figure 3 shows the speed-ups and slow-

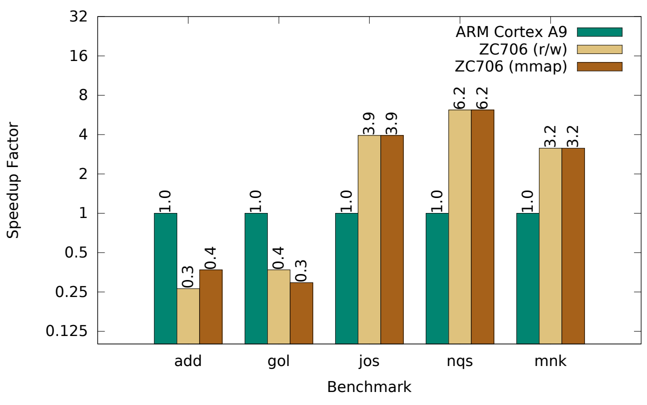

Fig. 3. Speedup compared to ARM Cortex A9

downs measured for the different RCU API implementations based on memory mapped registers ZC706 (mmap) and file system interface communication ZC706 (r/w), compared to the pure software execution on the Zynq's Cortex A9 cores alone (ARM Cortex A9). For longer kernels with runtimes exceeding the framework overhead (such as jos, nqs, mnk), there is no measurable benefit from using the mmap approach, since the communication with the hardware is no longer the performance bottleneck. When considering that the read/write approach offers the benefit of safe, system-wide access to the hardware (even across process boundaries), we conclude – contrary to our initial expectation – that the read/write approach is preferable in most of the intended use cases for our framework. Figure 3 also shows, that even for some of the relatively simple kernels examined here, performance gains using our framework over medium-sized embedded CPUs are already possible, despite the kernels just being high-level synthesized from C instead of manually optimized solutions.

VI. Related Work

There have been several approaches to integrate FPGAs in heterogeneous computing, e.g., [?]. In [?], a HLS flow is presented that constructs concurrent hardware modules using Pthread and OpenMP syntax, but no framework for an execution environment is specified. Such a framework was presented in [?]: FUSE provides accelerator integration which is completely transparent to the SW designer. However, this requires the same kernel code to be used for *all* devices, contrary to our approach. Several approaches for hardware/software co-execution have been shown, e.g., the tools described in [?] and [?]; however, integrating other hardware (e.g., GPGPUs) is currently not possible. In terms of scope, [?] is closest to ours, but focusing on GPGPUs. [?] and [?] introduce a combination of a polling and an interrupt based scheme. We support dynamic switching between polling and interrupt-driven signaling under developer control, but do not switch between the modes automatically. Task-management frameworks like [?] go a step beyond our own effort by realizing the entire scheduler in hardware. At this stage of our work, we leave these in the higher software layers of the X10 runtime. As a different implementation alternative to our combination of device driver and RCU API, a microkernel hypervisor [?] could have been chosen to encapsulate access to the RCU. However, as X10 expects a full-scale operating system such as Linux to be present, we chose the traditional approach in x10rt_fpga.

VII. Conclusion

We have shown that our hardware/software architecture for pools of high-level synthesized kernels is suitable for integration in heterogeneous parallel runtimes, such as x10rt. Naturally, hand-built custom solutions developed specifically for a given FPGA board could outperform such framework-based solutions, but we argue that the convenience, maintainability, and reduced development cost of a single-source solution justifies the potential performance losses.

VIII. Acknowledgment

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

IX. Bibliography

References

[B+02] D. Bonachea et al. Gasnet: A portable high-performance communication layer for global address-space languages. *CS258 Parallel Computer Architecture Project, Spring*, 2002.

[C+11a] A. Canis et al. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA int. symposium on field prog. gate arrays*, pages 33–36. ACM, 2011.

[C+11b] D. Cunningham et al. GPU programming in a high level language: compiling X10 to CUDA. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, page 8. ACM, 2011.

[C+13] Jongsok Choi et al. From software threads to parallel hardware in high-level synthesis for fpgas. In *Field-Programmable Technology (FPT), 2013 Int. Conference on*, pages 270–277, Dec 2013.

[D+06] K. Danne et al. Executing hardware tasks on dynamically reconfigurable devices under real-time conditions. In *Field Programmable Logic and Applications '06, Int. Conference on*, Aug 2006.

[E+04] Kemal Ebcioglu et al. X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access. In *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004.

[G+96] G.R. Gao et al. Polling watchdog: Combining polling and interrupts for efficient message handling. In *Computer Architecture, 1996 23rd Annual International Symposium on*, pages 179–179, May 1996.

[H+13] J. Huthmann et al. Hardware/software co-compilation with the nymble system. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th Int. Workshop on*, July 2013.

[I+11] Aws Ismail et al. FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. *IEEE 19th Ann. Int. Symposium on Field-Prog. Custom Computing Machines*, May 2011.

[L+96] Langendoen et al. Integrating polling, interrupts, and thread management. In *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers '96., 6th Symposium on the*, Oct 1996.

[L+07] H. Lange et al. An execution model for hardware/software compilation and its system-level realization. In *Field Programmable Logic and Applications, 2007, Int. Conference on*. IEEE, 2007.

[P+13] Khoa Dang Pham et al. Microkernel hypervisor for a hybrid arm-fpga platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th Int. Conference on*, June 2013.

[REP14] REPARA - Reengineering and Enabling Performance and poweR of Applications. http://www.repara-project.eu, 2014. Accessed: 07/14.

[S+10] Vijay Saraswat et al. The Asynchronous Partitioned Global Address Space Model. Technical report, Toronto, Canada, June 2010.