ARACompiler: A Prototyping Flow and Evaluation Framework for Accelerator-Rich Architectures

Yu-Ting Chen, Jason Cong, and Bingjun Xiao
Computer Science Department, University of California, Los Angeles, CA, USA
Email: {ytchen, cong, xiao}@cs.ucla.edu

***Abstract*—Accelerator-rich architectures (ARAs) provide energy-efficient solutions for domain-specific computing in the age of dark silicon. However, due to the complex interaction between the general-purpose cores, accelerators, customized on-chip interconnects, customized memory systems, and operating systems, it has been difficult to get detailed and accurate evaluations and analyses of ARAs on complex real-life benchmarks using the existing full-system simulators. In this paper we develop the ARACompiler, which is a highly automated design flow for prototyping ARAs and performing evaluation on FPGAs. An efficient system software stack is generated automatically to handle resource management and TLB misses.We further provide application programming interfaces (APIs) for users to develop their applications using accelerators. The flow can provide 2.9x to 42.6x evaluation time saving over the full-system simulations.**

I. INTRODUCTION

The current multi-core scaling may not sustain due to the power limit on a single device [1]. Accelerator-rich architectures are attractive alternatives to achieve both high-performance and low-power requirements by offloading the computation from general-purpose CPUs to accelerators. The important question we try to address in this paper is how to evaluate accelerator-rich architectures (ARAs) accurately and efficiently. The question can be divided into two parts: (1) accurate modeling of an ARA, and (2) efficient evaluation of an ARA on real workloads with OS interactions.

Most existing studies try to modify the state-of-the-arts full-system simulators for ARA evaluation. However, this approach has multiple drawbacks. First, current full-system simulators are designed to model general-purpose processors. However, the memory system, including both the memory hierarchy and the network-on-chip in an ARA, are usually quite different from CPU architectures. In particular, the ARA memory system is customized to provide fast on-chip accesses and sufficient aggregate on-chip/off-chip bandwidth to meet the high throughput demand from accelerators. It is difficult to accurately model and specify the memory system of an ARA based on current full-system simulators, e.g. [2][3], without significant modifications. Second, the speed of full-system simulation is slow. For example, gem5 [2] and Flexus [3] can achieve up to 300 KIPS in the detailed CPU mode and 3MIPS in the fast-forwarding mode. It is still about a 1000x slowdown compared to native execution (2GIPS). In this paper we discuss our design of a highly automated prototyping flow, called Accelerator-Rich Architecture Compiler (ARACompiler), to prototype our ARAs on a modern FPGA-SoC.

II. ACCELERATOR-RICH ARCHITECTURES AND PROTOTYPING PLATFORM

Figure 1 is an overview of an ARA. An ARA contains two planes: (1) the accelerator plane and (2) the processor plane. From a system perspective, the user applications are launched in the processor plane and offload the computation-intensive tasks to the accelerator plane. The system software stack acts as the interface between the two planes. It provides the services of reservations, starts, and releases for the accelerators. The software stack is implemented in the privileged mode and transparent to users. In addition, the TLB misses issued from the accelerators are handled in the software.

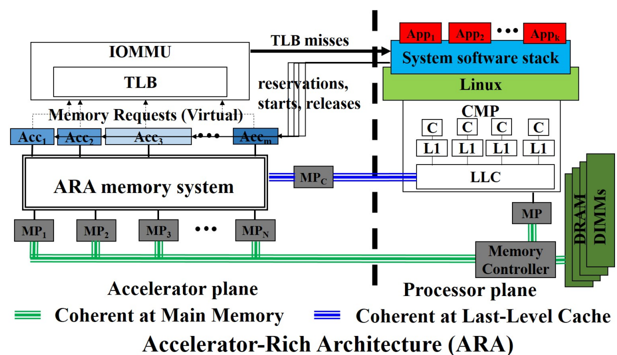

Fig. 1. ARA overview: accelerator plane and processor plane

We choose the Xilinx Zynq ZC706 evaluation board as our underlying prototyping platform. ZC706 has a Zynq SoC with 1GB on-board DRAM. A Zynq SoC is composed of a dual-core ARM Cortex-A9 and FPGA fabrics, which can be used to implemente accelerators and ARA memory system. The system software stack and user applications can be launched on the ARM processor with Linux support.

III. ARACOMPILER DESIGN AUTOMATION FLOW

The main challenges to architecture exploration through hardware prototyping is the long development cycle of each generation of ARA, requiring extensive coding in HDL. To overcome this, we provide an design automation flow, similar to the approach used in [4], with the following features: (1) high-level ARA specification file, (2) accelerator development using high-level synthesis tools, and (3) highly parameterized hardware templates for ARA memory system generation.

First, the ARA specification file is provided for users to specify components and configure the parameters in the accelerator plane. Users can easily design and evaluate their new accelerators to the reusable baseline prototype or perform design space exploration. Second, high-level synthesis tools, such as Vivado HLS, enable automatic synthesis from high-level specifications in C to low-level cycle-accurate RTL codes. We provide a standardized accelerator interface in HLS-compatible C to shorten the accelerator design cycle for users. Third, we provide a highly parameterized hardware templates

for ARA memory system generation to remove this burden from users. Figure 2 shows our design automation flow.

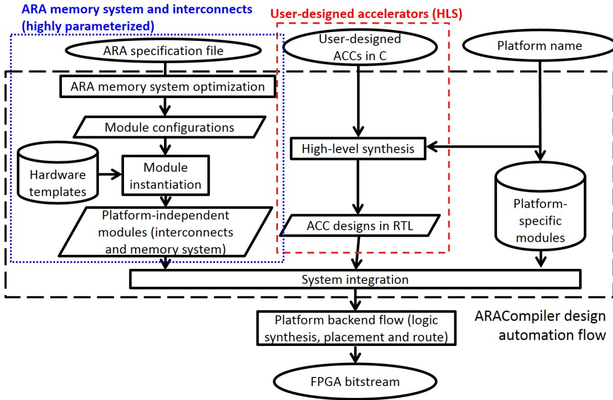

Fig. 2. ARACompiler design automation flow

IV. ARA System Software Stack

Figure 3 shows the ARA system software stack. ARACompiler can automatically generate the related software modules based on the ARA specification file. The four major components in the system software stack are: (1) global accelerator manager (GAM), (2) dynamic buffer allocator (DBA), (3) TLB miss handler, and (4) coherence manager. GAM is responsible for (1) interfacing with user applications, (2) accelerator resource management and task scheduling, and (3) requesting for buffer resource. User applications can talk to GAM with the provided APIs. DBA receives the task requests from GAM and then allocate the shared buffer resources for the requests. A starvation-free buffer allocation policy is provided by default for DBA. We use a software-based TLB miss handler in ARACompiler. IOMMU groups multiple TLB misses together and sends them to the miss handler at once to reduce the performance overhead. When a user directly write accelerator data to off-chip DRAM instead of the coherent L2 cache for higher memory bandwidth, the overlapping pages residing in L2 are invalidated by the coherence manager.

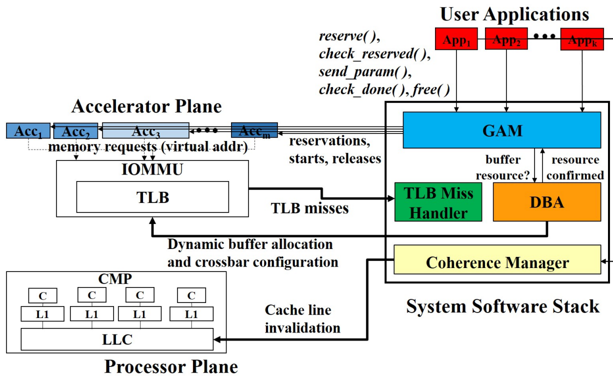

Fig. 3. System software stack and the interactions with the ARA and user applications

V. ARA Application Programming Interfaces

ARACompiler can generate the header file of accelerator APIs for programmers automatically by using the information of ARA specification file. For each type of accelerator, we provide APIs: (1) $reserve()$, (2) $check_reserved()$, (3) $send_param()$, (4) $check_done()$, and (5) $free()$. Users can develop their applications with the C++ classes and member functions to manipulate the accelerator in the ARA.

VI. ARACompiler Flow Runtime

Figure 4 provides the runtime comparison of (1) application execution time on the ARA prototype, (2) ARACompiler flow and prototype construction time, and (3) full-system simulation time collected from [5]. Xilinx synthesis flow contributes more than 97% of total runtime of (2). Even with such slow synthesis time, our flow can save 2.9x to 42.6x evaluation time over the full-system simulations. The saving becomes larger when the input sizes grow.

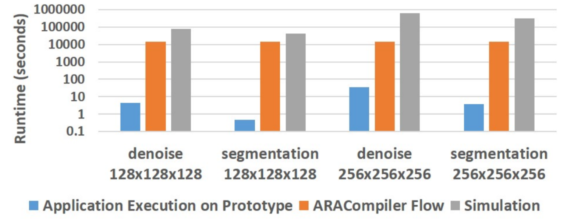

Fig. 4. Runtime comparison: (1) application execution time on the ARA prototype, (2) ARACompiler flow, and (3) full-system simulation [5]

VII. Conclusions

We propose ARACompiler, which is a high automated prototyping flow to generate ARAs on FPGA. Designers can easily integrate their accelerator designs with our reusable and highly parameterized hardware templates to customize the shared memory system. Furthermore, we provide a system software stack and user APIs for designers to develop and evaluate their applications on the prototype. We believe ARACompiler can be an attractive alternative for ARA evaluation.

VIII. Acknowledgement

This work is partially supported by the Center for Domain-Specific Computing under the Intel Award 20134321 and NSF Award CCF-1436827. It is also supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA'11*, pp. 365–376.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[3] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, Jul. 2006.

[4] Y.-T. Chen, J. Cong, M. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou, "Accelerator-rich cmps: From concept to real hardware," in *ICCD'13*, pp. 169–176.

[5] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich cmps," in *DAC'12*, pp. 843–849.