Accelerator-Rich CMPs: From Concept to Real Hardware

Yu-Ting Chen, Jason Cong*, Mohammad Ali Ghodrat, Muhuan Huang, Chunyue Liu, Bingjun Xiao*, Yi Zou
Computer Science Department
University of California, Los Angeles
Los Angeles, California, 90095
*{cong, xiao}@cs.ucla.edu

***Abstract*—Application-specific accelerators provide 10-100x improvement in power efficiency over general-purpose processors. The accelerator-rich architectures are especially promising. This work discusses a prototype of accelerator-rich CMPs (PARC). During our development of PARC in real hardware, we encountered a set of technical challenges and proposed corresponding solutions. First, we provided system IPs that serve a sea of accelerators to transfer data between userspace and accelerator memories without cache overhead. Second, we designed a dedicated interconnect between accelerators and memories to enable memory sharing. Third, we implemented an accelerator manager to virtualize accelerator resources for users. Finally, we developed an automated flow with a number of IP templates and customizable interfaces to a C-based synthesis flow to enable rapid design and update of PARC. We implemented PARC in a Virtex-6 FPGA chip with integration of platform-specific peripherals and booting of unmodified Linux. Experimental results show that PARC can fully exploit the energy benefits of accelerators at little system overhead.**

***Keywords*-customizable computing, computer architecture, prototyping, FPGA, design automation.**

I. Introduction

Accelerator-rich architectures can bring 10-100x energy efficiency by offloading computation from general-purpose CPU cores to application-specific accelerators [1–6]. On-chip accelerators can be categorized in two classes. While tightly coupled accelerators [1, 2] are constrained in a CPU's pipeline and thus experience limited benefits of a full customization, loosely coupled accelerators [3–6] completely bypass CPU overheads (instructions, caches, etc.) and can be optimized in a larger design space. Prior work [3–6] proposed a methodology for integrating a sea of loosely coupled accelerators along with very few CPU cores to build up an energy-efficient computing system. This methodology does not expect all the accelerators to be used all the time, but it guarantees that each computation task is executed by the most efficient hardware.

Methodologies for how to integrate massive CPU cores in a system have been well established [7]. The related research mainly works on the following three key issues: data transfer between userspace and device memories, on-chip memory architecture, and hardware resource management. These issues also arise in the integration of massive accelerators. However very little research has been performed on what changes should be made in the solution when we switch from CPU-centric architectures to accelerator-centric architectures.

This work undertakes an implementation study of a general framework for accelerator-rich CMPs in an FPGA-based prototype. We name our prototype of accelerator-rich CMPs as PARC. By realizing this in RTL and running it in real hardware, we find that many architecture assumptions and design choices used in prior research [4–6] work only for CPU cores and lose effectiveness when facing a sea of accelerators. We propose our own solutions, including hardware innovation and software automation, to meet the demand of accelerators. These solutions are:

1) Shared system IPs for accelerators to transfer data between userspace and device memories without cache overhead.
2) A dedicated interconnect between accelerators and memories to enable memory sharing.
3) An extensible accelerator manager in a standalone bare-metal processor to perform runtime scheduling of accelerator resources.

In addition, we find that due to the large scale and high heterogeneity of accelerator-rich architectures, their design cycles will significantly increase if we still follow conventional design methodologies. In our prototyping, we develop an automated flow to enable rapid development of PARC:

1) For accelerator designers, we integrate high-level synthesis (HLS) tools in our development flow to allow accelerators to be designed in a high-level abstraction (ANSI C), along with a standardized accelerator interface in HLS-compatible C.
2) For application programmers, we virtualize physical accelerator resources to accelerator classes, and provide objected-oriented APIs so that programmers can use different accelerators by calling member functions of different accelerator objects.
3) For system developers, we create a fully automated flow of system synthesis and generation from a high-level system description file.

Our PARC is verified in a commodity FPGA chip with the goal of providing guidelines for future ASIC implementation. We report experimental results after running the real hardware with an unmodified Linux. We demonstrate that PARC can fully exploit the energy benefits of accelerators at little system overhead.

II. Related Work

A good example of accelerator-rich CMPs is the Wire Speed Processor which has four accelerators (XML, Regex, Comp and Crypto) shared by several CPU cores [4]. A general framework for accelerator-rich CMPs (ARC) as proposed in [3] is shown in Fig. 1. ARC presented a hardware resource management scheme for accelerator sharing, scheduling, and virtualization. This scheme introduced a global accelerator manager (GAM) implemented in hardware to support sharing and arbitration of multiple cores for a common set of accelerators. It also proposed to use several new custom instructions for communicating with the GAM to avoid OS overhead in accelerator interaction.

The on-chip memory architecture of accelerator-rich CMPs is another research focus. The necessity of on-chip memory sharing among accelerators was reported in [5]. Later, a more complete

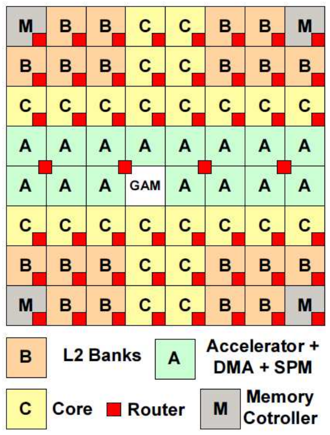


Figure 1: The general framework for accelerator-rich CMPs (ARC) in [3].

architecture that extends both [3] and [5] was proposed in [6]. It supports accelerator buffers in NUCA and uses a flexible paged allocation method to limit the impact of fragmentation in a shared buffer bank. It also developed a dynamic interval-based global allocation method to assign spaces to accelerators, where each has a range of buffer size requirements with different buffer utilization efficiencies.

III. Architecture Improvements

In this section, we report the challenges that we encountered during our development of PARC in real hardware, and the architecture improvements that we made over the prior work in [3] which was derived based on architecture simulation.

A. Data Transfer Between Userspace and Device Memories

Data for accelerators to process are initialized by a user's host applications and are stored in a physical space somewhere in the main memory that is managed by the OS. Accelerators need to prefetch userspace data into their device memories (for accelerators, usually scratch-pad memories, or SPMs) which they can manipulate in their customized data paths without the overhead of cache speculation. If an accelerator needs to transfer certain data between userspace and its SPMs, the data addresses need to be mapped from the virtual address space to the physical memory storages so that the data can be transfered by direct memory accesses.

1) Lessons Learned: A prior work [3] assumes that each accelerator has a private translation look-aside buffer (TLB) to do page translations and a direct memory access controller (DMAC) to execute data transfers. During our prototyping, we found that this kind of design led to two main drawbacks:

1) Resource underutilization. It is true that many CPU-centric architectures put a private TLB in each CPU core (usually in the data path between the L1 cache and L2 cache). As long as the system is kept busy and all the CPU cores are working, all the TLBs are used. In accelerator-rich architectures, however, most of accelerators are powered off, and a putting private TLB in each accelerator will always lead to resource underutilization. TLBs and DMACs of different accelerators should have the same circuit schematic and can be shared among accelerators.
2) Reduced portability. Putting a TLB and a DMAC in an accelerator requires an accelerator designer to be aware of how data are stored and transferred in its target accelerator-rich architecture. An accelerator needs to be redesigned when it is migrated to another accelerator-rich platform.

In our prototyping, We also found another problem about [3]. In [3], the DMACs always go to the L2 cache first when processing their data transfers. However, this decision was based on the small data size (32x32x32 image) chosen to avoid the long simulation time. This data size can fit in the L2 cache entirely. When we use a real data size (128x128x128 image) in our prototyping, most of data are stored in the main memory, and it is more desirable to skip the L2 cache in data movement. All the data in cache that is to be processed by an accelerator can be invalidated in parallel with the accelerator launch. In addition, going through cache will limit a data transfer by the unit of the cache line size (e.g., 64B), and thus will have to give up the optimization opportunities that emerge when accelerators request sequential data over long continuous address spaces.

2) Proposed Architecture: We decouple TLBs and DMACs with accelerators in PARC. We offload them to our system IPs that are designed specifically for accelerators — an input/output memory management unit (IOMMU) and direct memory access controllers (DMACs) — so that accelerators only need to send data transfer requests to IOMMU, as shown in Fig. 2. IOMMU

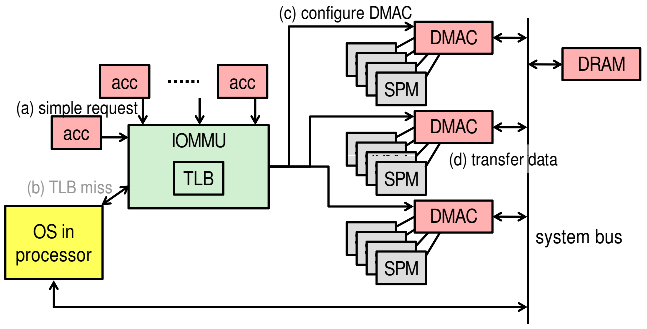


Figure 2: The architecture of our system support to offload data transfer from accelerators.

contains a translation lookaside buffer (TLB) for page translation from virtual addresses to physical addresses. If there is a TLB miss, IOMMU will automatically consult with the OS for page translation. IOMMU also cuts a data transfer on the page boundary into data blocks so that all the addresses in each block are continuous. Then IOMMU configures DMACs to transfer data from the physical space to accelerators' SPMs in burst mode. DMACs are directly connected to the system bus towards the main memory to perform data transfers in the most efficient way.

Based on our observations during prototyping, we decided to put only one IOMMU but multiple DMACs in PARC, as shown in Fig. 2. The reason for having only one IOMMU is that the task of IOMMU is lightweight so a single IOMMU is able to serve all working accelerators. Another reason to have a global IOMMU serving all the accelerators is that it can explore the data locality among successively launched accelerators to reduce the

TLB miss rate. The reason for having multiple DMACs is that they can transfer data in parallel to keep DDR busy and the start up time of their burst data transfers can be overlapped and hidden.

3) Implementation Details: Since our IOMMU needs to serve all the working accelerators instead of a single core, conventional private TLB designs may no longer be applicable. We observed that the processing of a data transfer request by IOMMU can be stalled by two kinds of latencies: 1) the TLB miss that needs to go to OS, and 2) waiting for DMAC to transfer a large amount of data from DRAM. These two latencies are very large compared to the workload of the IOMMU and are unpredictable since they depend on the external workloads outside of the IOMMU. Our initial design follows the conventional TLB scheme and cannot provide service for an accelerator if it has received a request from another accelerator and gets stalled by either of these two kinds of latency. When it needs to serve a sea of accelerators, this stalling results in a significant penalty on accelerator performance. We solve this problem by implementing IOMMU into a non-blocking design with three parallel submodules, as shown in Fig. 3. The "front"

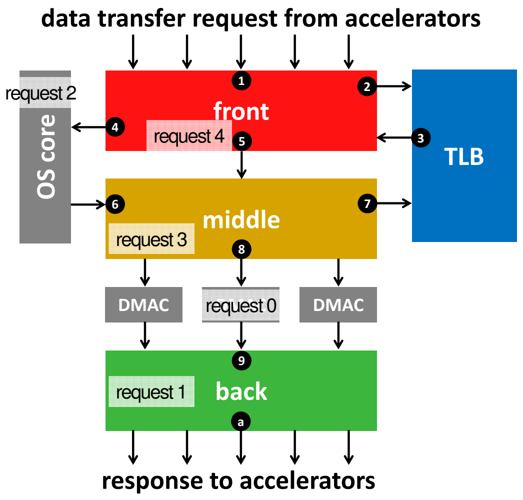


Figure 3: Our IOMMU design that processes accelerator requests without being blocked by external OS or DMACs.

module in this figure focuses on page translation. The "middle" module focuses on splitting data transfer on page boundaries. The "back" module focuses on monitoring DMACs and responding to accelerators. Each submodule can work on different requests as shown in Fig. 3. While some requests are stalled in OS (request 2) or DMACs (request 0), their successive requests can move forward. Furthermore, for quick TLB lookup, we designed the TLB to be 4-way associative with pseudo LRU replacement.

B. On-chip Memory Architecture

As proposed in [5, 6], with on-chip memory sharing, the saved transistors from memory sharing can be used to implement more accelerators to cover more application kernels.

1) Lessons learned: Prior work [5, 6] mainly focuses on the optimization of memory space allocation for accelerators during runtime. It is assumed in [6] that accelerators can switch between several operating modes with different requirements of buffer size and off-chip bandwidth. During our prototyping, we found that it is more efficient for accelerators to assume a fixed number of scratchpad memories (SPMs) and fixed SPM sizes during design time so that we can use hard-wired circuits to control these SPM resources. In this case, multiple SPM allocation options are not available (i.e. the BB-cuve in [6] for buffer allocation is reduced to a single operating point). This means that when an accelerator is launched, a resource manager has to find sufficient idle SPM banks to fit the SPM requirement of this accelerator. Furthermore, to avoid memory port competition, these SPM banks cannot be assigned to other accelerators until this accelerator finishes its task. Therefore the SPM allocation becomes much easier in this case, and in our prototyping, the SPM optimization scheme proposed in [6] is not needed.

What we observed to be the major challenge of memory sharing among accelerators is the interconnects between accelerators and shared memories; this issue is not addressed in prior work [5, 6]. A CPU performs a load/store every few clock cycles. Therefore, a simple interconnect, e.g., Fig. 4(a), will meet the bandwidth requirement. In contrast, accelerators run $>$ 100x faster than

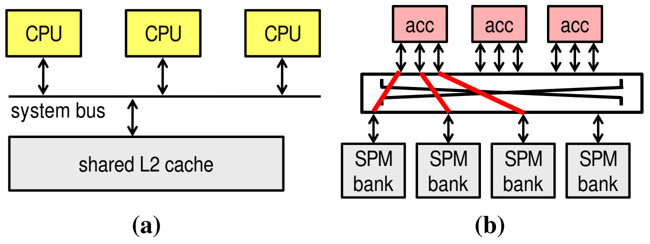


Figure 4: Difference between memory sharing among general-purpose CPUs and among accelerators. (a) A simple interconnect for CPU. (b) Demanding interconnects for accelerators.

CPUs [8] and need to perform several loads/stores every clock cycle. Not only accelerator IOs, but also the interconnects between accelerators and shared memories, need to meet this bandwidth requirement, as pictured in Fig. 4(b). Each accelerator needs to have at least N ports if it wants to fetch N data every cycle. The N ports need to be connected to N SPMs via N conflict-free data channels in the interconnects.

2) Guideline for Solving Challenges in Memory Sharing: To solve the challenge of the data demand between accelerators and shared memories, we need to modify the interconnects between them. If we follow the same design rules as those for interconnects between CPUs and shared memories and simply duplicate the interconnect hardware to meet the accelerator data demand, we will get a resource-consuming design close to a full crossbar. To solve this challenge, we develop a novel interconnect design that scales with a sea of accelerators, and we exploit three optimization opportunities that emerge in accelerator-rich architectures:

1) The multiple data ports of the same accelerators are powered on/off together, and the competition for shared resources among these ports can be eliminated to save interconnect transistor cost.
2) Dark silicon imposes a limit on the maximum number of accelerators powered on in an accelerator-rich architecture, and the interconnects can be partially populated to just fit the data access demand limited by the power budget.

3) The heterogeneity of accelerators leads to execution patterns among accelerators and, based on a probability analysis to identify these patterns, interconnects can be optimized for the expected utilization.

Since this paper focuses on a platform-based design, we will not go into further detail about our interconnects. These details can be found in [9].

C. Accelerator Management

Multiple applications running simultaneously in an accelerator-rich architecture may compete for accelerators. A global accelerator manager (GAM) is needed to perform arbitration of accelerators among applications.

1) Lessons Learned: During our prototyping, we found that popular OS scheduling methods of GPU management and many-core CPU management are also applicable for accelerators (e.g., scheduling based on estimation of execution time adopted in [3]). The freedom we have is to decide whether to implement the accelerator manger in either OS kernel, or a standalone processor, or ASICs. We decided to use a standalone lightweight CPU core based on the following considerations:

- Compared to an implementation in OS [10], the memory size limitation of the OS kernel cannot accommodate complex scheduling policies (e.g., machine learning which becomes meaningful with operation records from hundreds of accelerators). But a bare-metal processor (i.e., running without OS) can support GAM with complex scheduling algorithms as well as workload isolation.
- Compared to the GAM implementation in ASICs [3], our implementation in a CPU core is easier to develop and update. System developers can design the complex scheduler in C code and only need to recompile their codes when performing an update.

Note that though we implemented GAM in a CPU core, the overhead of the accelerator management is still small compared to the execution time of an accelerator, as shown in Fig. 5 (see Section III-D for our prototyping settings). We also found that talking to GAM via its driver and OS has very marginal overhead (the region "switch to OS" in Fig. 5). This is very different from the OS overhead reported in [3]. The work in [3] uses fine-grained accelerator chaining (which requires heavy communication via OS). In addition, it estimates the OS overhead by running Solaris10 in a Simics/GEMS simulation, which seems to overestimate in some cases (e.g., resulting in OS activity >99.6% in some cases). As a result, it proposed new customized instructions to access accelerators. But this turned out to be unnecessary in our real hardware implementation.

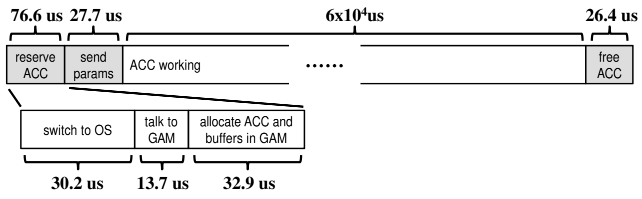


Figure 5: Little overhead (shadowed region) imposed by our architecture on accelerators. GAM = global accelerator manager.

D. Overall Architecture

We integrate all of our designs proposed in the previous sections, and prototype several instantiations of our architecture framework in commodity FPGAs. Our first phase of prototyping targets a Xilinx ML605 board with a Virtex-6 FPGA chip. Fig. 6 shows the overall architecture implemented in this FPGA. The architecture includes all of our platform-independent modules, such as IOMMU, DMACs, crossbar and global accelerator manager (GAM). It also includes platform-specific modules, such as Xilinx Microblaze CPU cores, MDM (Microblaze debug module to boot up OS core and GAM core), UART (as OS console interface), Ethernet (to download GAM driver and user applications with input data to run in OS), and Mailbox (as a communication IP between GAM core and GAM driver in OS core). The Xilinx development tool EDK for this FPGA chip allows us to have at most two "Microblaze" CPU cores in the FPGA. One CPU core is configured as a lightweight core for GAM. The other is configured to run Linux and user applications in the single-core mode. This can be extended to multicore when implemented in ASICs. We are also implementing our architecture in Zynq SoC with an ARM dual-core A9 processor in ASIC along with programmable logics.

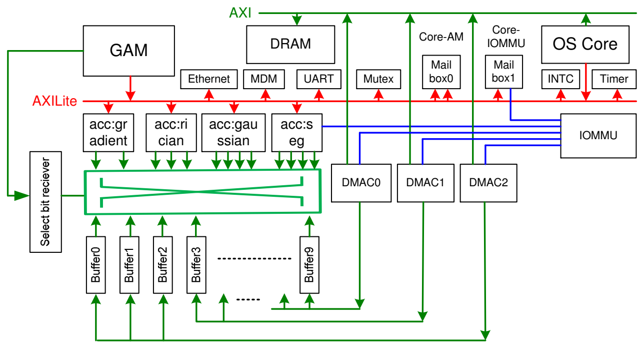


Figure 6: Overview of our PARC implemented in Xilinx Virtex-6 FPGA XC6VLX240T. GAM = global accelerator manager.

IV. Design Automation of PARC

During our prototyping, we found that due to the large scale and high heterogeneity of accelerator-rich architectures, their engineering cost increases significantly. In this section we discuss several technologies that we developed to reduce the human effort and speed up the design cycle of PARC.

A. Rapid Accelerator Designs

When there are hundreds of accelerators to be developed in an accelerator-rich architecture, in order to guarantee the time-to-market, we need to significantly improve the design productivity of each accelerator.

1) Integration With High-Level Synthesis Tools: Design productivity of accelerators can be improved by raising the level of design abstraction beyond register transfer level (RTL). High-level synthesis tools [11, 12] enable automatic synthesis of high-level, untimed or partially timed specifications (such as in C or SystemC) to low-level cycle-accurate RTL specifications for efficient implementation of accelerators.. As reported in [11], the code density can be easily reduced by 7-10x when moved to high-level specification in C, C++, or SystemC, and at the same time, resource usage can also be reduced by 11-31% in an HLS solution compared to a hand-coded design.

2) HLS-Compatible Accelerator Interface in C: HLS tools allow accelerator designers to develop standalone accelerators in C. When we target the design of accelerators that will be integrated in a system, we need to provide a standardized accelerator interface in HLS-compatible C as well. When an accelerator needs to communicate with the outside, it either receives function parameters from the global accelerator manager (GAM), or sends requests to IOMMU to transfer data between userspace and its SPMs, or accesses data in the shared SPMs which are assigned to the accelerator. Fig. 7 shows our standardized accelerator interface that fulfills these communication demands in a user-friendly way. As shown

```
 1 void AccGradient2D(
 2         volatile int* paramFIFO,
 3         volatile int* IOMMU_FIFO,
 4         volatile float SPM0[4096],
 5         volatile float SPM1[4096],
 6         volatile float SPM2[4096],
 7         volatile float SPM3[4096]
 8         )
 9 {
10     // read function parameters sequentially
11     int image_vaddr = *paramFIFO;
12     int height = *paramFIFO;
13     int width = *paramFIFO;
14     // main body starts here
15     ...
16     for( int i = 0; i < 4096; i++ )
17         SPM2[i] = SPM0[i] + SPM1[i];
18     ...
19 }
```

Figure 7: Our standardized accelerator interface in HLS-compatible C.

in lines 11-13 in Fig. 7, accelerator designers can fetch function parameters sequentially by reading the "paramFIFO." Then they can send data transfer requests by writing the "IOMMU_FIFO" in a similar way. After data are moved to the accelerator's SPMs, designers can access these SPMs just like accessing data arrays in C (line 17 in Fig. 7).

During our prototyping, we found that the data transfer requests sent to IOMMU involve many packet contents. It is not desirable to expose these contents to accelerator designers. We further package these contents and offer a user-friendly API in C to accelerator designers to send data transfer requests as shown in Fig. 8. This

```
IOMMU_request(
        flag, // read or write
        vaddr, // the virtual address of the target data array in host app
        buf_id, // SPM ID of the accelerator
        buf_addr, // target starting address in the SPM
        length, // transfer length
        copy, // the number of replications of the request
        stride, // the distrance of the starting virtual address of two adjacent replications
        buf stride, // the distrance of the starting SPM address of two adjacent replications
        );
```

Figure 8: The C-based API for an accelerator designer to send data transfer requests to IOMMU.

API is fully compatible with high-level synthesis, and its function calling will be mapped to push each flit in a request packet into the FIFO-based channel "IOMMU_FIFO" that is connected to IOMMU. The API allows accelerator designers to specify an arbitrary length of data to transfer according to the application demand. Since our DMACs do not go through the CPU cache, the data transfer will not be limited by the fixed data block size of cache line. Accelerator designers can call the API anywhere in their C codes and can prefetch data whenever they wish. The initial interface (without the shadowed lines in Fig. 8) limits one data transfer request on a continuous address space starting at "vaddr" with a size of "length." In many cases, designers may want to fetch a data region with discontinuous addresses, e.g., the shadowed region in Fig. 9. This motivates us to enhance the data transfer

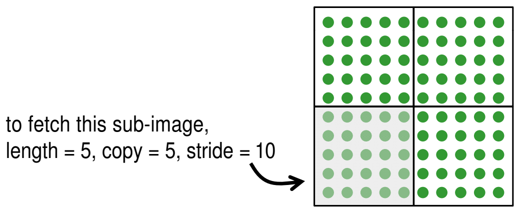
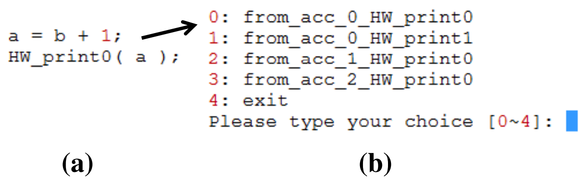


Figure 9: An example where designers want to fetch a subimage with discontinuous addresses in a 2D image.

protocol as shown in the shadowed lines in Fig. 8. Though the data region to be fetched contains discontinuous addresses, its regular shape can be defined by two extra variables "copy" and "stride" in addition to "length." This enhanced protocol not only saves designers the burden of sending many requests for a data region with discontinuous space, but also gives IOMMU opportunities to process the transfer of a data region in a more global way.

With our standardized accelerator interface, accelerators can be designed in pure C, synthesized by HLS tools to RTLs, and seamlessly fit into PARC. The standardization also provides accelerator designs full portability when the underlying physical hardware platform is changed.

3) Hardware Debug Support: In PARC, accelerators can be written in C and then are synthesized into RTL. But accelerators may manifest errors when running in the system. It is difficult to debug accelerators in a system since they have been mapped to hardware. As a further step in our system support for rapid accelerator design, we allow accelerator designers to have "printf"-like capability in their C codes to obtain debug logs after accelerator runs. As shown in Fig. 10(a), designers can call our API "HW_print" in their codes. We integrate the "HW_print"



```
a = b + 1;
HW_print0( a );
```

```
0: from_acc_0_HW_print0
1: from_acc_0_HW_print1
2: from_acc_1_HW_print0
3: from_acc_2_HW_print0
4: exit
Please type your choice [0~4]:
```

(a) **(b)**

Figure 10: Interface of our hardware debug support for accelerator designers. (a) An accelerator design in C with our API to dump debug logs. (b) Interface for designers to take a look at the dumped debug logs after accelerator runs in PARC.

API with the execution flow of the high-level synthesis tool "Vivado_HLS" [12] to generate an extra channel "log FIFO." Logs can be dumped through these channels to the configurable debug module in PARC, as shown in Fig. 11. The configurable debug module will be customized to provide an input port for each accelerator containing "HW_print" during our system generation. This module also contains on-chip memories to store the debug

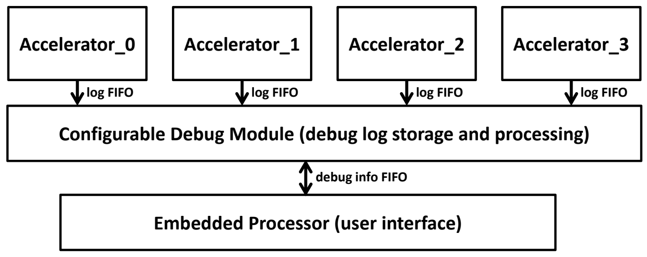


Figure 11: Architecture of our hardware debug support.

logs after preprocessing (it will only store the most recent ones if cannot fit into the memory size). We also reuse the CPU in PARC to provide a user interface for designers to take a look at the debug logs as shown in Fig. 10(b).

B. Virtualization of Accelerators in User Applications

Application programmers do not want their codes to be dependent on the physical configuration (such as quantity and utilization) of accelerators in an accelerator-rich architecture. Accelerators need to be virtualized from a user's applications, just like device virtualization in OS.

1) Object-Oriented Interface to Application Programmers: We implemented an API library so that an application programmer can use on-chip accelerators by calling member functions of the corresponding accelerator objects. The API library contains an GAM driver and an object-oriented user interface that can translate the accelerator demands from a user application into the communication packets to be sent to the global accelerator manager (GAM). Fig. 12 shows an example code of a user application using an accelerator. In line 1, the programmer declares an accelerator

```
 1 AccGradient2D acc0;
 2 if( acc0.reserve( ) )
 3 {
 4     acc0.start(
 5             image, // variable array
 6             256, 256 // image size
 7             );
 8     while( !( acc0.done( ) ) )
 9     {
10         wait( 1000 );
11     }
12     acc0.free( );
13 }
14 else
15 {
16 // accelerator not availabe, compute by CPU
17     CPU_gradient2D( image, 256, 256 );
18 }
```

Figure 12: An example code of a user application using an accelerator through our object-oriented APIs.

object "acc0." The accelerator class "AccGradient" is a child of the public class "Accelerator" that we developed. It corresponds with the accelerator kernel "Gradient" which is used to speed up many applications in the medical imaging domain [13]. Each type of accelerator kernel is mapped to a child class in our user interface. Then in line 2 of Fig. 12, after declaration of the accelerator object, the programmer tries to reserve a "Gradient" accelerator. The reservation may fail when all the "Gradient" accelerators existing in the system are occupied by other applications at the moment. The programmer may choose to wait for an available "Gradient" accelerator, or switch the computation task to CPU as shown in line 17 of Fig. 12. If the reservation succeeds, the programmer can start the accelerator, and send parameters of the computation task — just like calling a kernel function, as shown in line 4 of Fig. 12. Then the programmer can call the member function in line 8 to check whether the accelerator has finished its task or not. Last the programmer may free the accelerator so that other applications can use it as shown in line 12 of Fig. 12, or else may keep the accelerator reserved and assign another task to it.

2) Software Pipelining of Accelerators: An application may have multiple accelerators working in parallel. With our object-oriented interface, application programmers can easily write codes with software pipelining of accelerators as shown in Fig. 13. In

```
float images[N][256][256];
...
for( int i = 1; i < N; i++ )
{
    acc0.start( images[i], 256, 256 );
    acc1.start( images[i-1], 256, 256 );
    while( !( acc0.done( ) && acc1.done( ) ) )
    {
        wait( 1000 );
    }
}
...
```

Figure 13: An example code of software pipelining of accelerators.

the example code of Fig. 13, each image is processed by the accelerator "acc0" first and then by the "acc1". Note that we do not check the data coherence among accelerators used by programmers. Therefore, programmers need to guarantee that there is no data race condition when they call a set of accelerators that might be executed concurrently.

C. Automatic System Synthesis and Generation

The developer of an accelerator-rich architecture may want to add new accelerators to the system, or try different types of accelerators or different numbers of SPMs, etc. As described in the previous sections, we add auxiliary hardware modules (IOMMU, DMACs, crossbar and GAM) to the system. These hardware modules need to be customized so that they correspond to the developer's modification of the system. If this large-scale system with hundreds of accelerators is manually maintained, a huge amount of human effort will be needed. Parameterization of a system can save human effort, but it is not easily realized in PARC. For example, the crossbar topology can change considerably since its reoptimization will be invoked upon each architectural update. The communication channels among the new accelerators, the IOMMU, the GAM, etc., also need to be generated. Motivated by these challenges, and to save human effort, we implemented an architectural template rather than an ad hoc architecture, and we developed a fully automated flow of system synthesis and generation.

1) System Generation Interface to System Developers: Our flow of automatic system generation needs only a high-level system description file from a developer. Fig. 14 shows an example system configuration file. This is an "xml" file with several fields. In the

```
<system>

<ACCs>
    <acc type="gradient" num="2" num_params="5" path="acc/gra">
        <SPM size="4096" num="6"/>
    </acc>
    <acc type="rician" num="1" num_params="7" path="acc/ri">
        <SPM size="4096" num="12"/>
    </acc>
    <acc type="gaussian" num="2" num_params="7" path="acc/gau">
        <SPM size="4096" num="4"/>
    </acc>
    <acc type="segmentation" num="1" num_params="13" path="acc/seg">
        <SPM size="4096" num="8"/>
    </acc>
</ACCs>

<SharedSPMs size="4096" num="32"/>

<crossbar flex="3"/>

<debug on="1"/>

</system>
```

Figure 14: An example system configuration file created by a system developer to generate a system with four types of accelerators via our automated flow.

first field, "ACCs," in each entry the developer can specify the name of the accelerator to add, the duplication of this accelerator to put in the system, the number of its function parameters, the path of this accelerator design, and the number of SPMs needed by this accelerator. In the second field, "SharedSPMs," the developer can specify the total number of SPMs to put in the system. In the third field, "crossbar," the developer can tune the crossbar between accelerators and shared SPMs. In the fourth field, "debug," the developer can specify whether to turn on the hardware debug support or not (see details in Section IV-A3). As long as the developer gets this file prepared, he can invoke our flow for automated system generation and get all the RTLs that are ready for the back end process.

2) *Automation Flow:* Fig. 15 shows the overview of our fully automated flow of system synthesis and generation. This flow

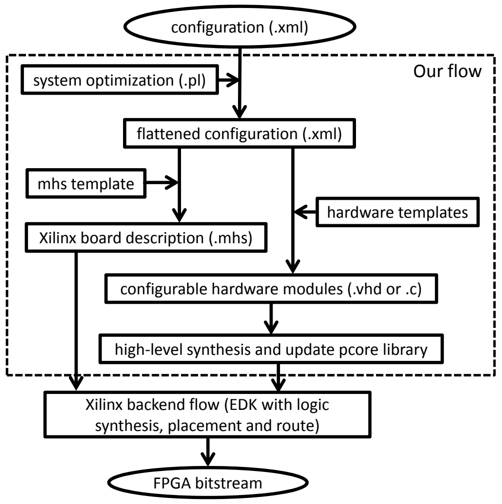

configuration (.xml)
system optimization (.pl)
Our flow
flattened configuration (.xml)
mhs template
hardware templates
Xilinx board description (.mhs)
configurable hardware modules (.vhd or .c)
high-level synthesis and update pcore library
Xilinx backend flow (EDK with logic synthesis, placement and route)
FPGA bitstream

Figure 15: Our fully automated flow of system synthesis and generation.

begins with the configuration file provided by the developer, and all following steps will be executed automatically upon a single "make" button. It is also seamlessly connected to a Xilinx back-end flow to implement the architecture prototype in the Xilinx FPGA. The first step of the flow is to apply system optimization to the developer's configuration to get a full system configuration. Then it is combined with our hardware templates to create the hardware modules that are customized to the developer's demand. If necessary, high-level synthesis tools will be called and the IP library will be updated. Depending on the target technology, e.g., ASIC at 45nm node, or Xilinx Virtex-6 FPGA in our case, the flow will go through the technology description (e.g., the Xilinx board description shown in Fig. 15) and will be seamlessly integrated with the back-end process (e.g., Xilinx EDK flow for bitstream generation shown in Fig. 15). Table I shows the engineering cost saved by our automatic system generation.

Table I: Design effort savings from adding 10 "segmentation" accelerators to our automation flow. We only need to add 3 lines of code in the input file of our flow, and >3000 lines of code will be automatically generated.

	components	# of code lines
developer input	configuration file	3
automatically generated	GAM	113
	IOMMU	200
	crossbar	2240
	system specification	542
	total	3095 (>1,000x)

To the best knowledge of the authors, this is the first work that provides automatic generation of accelerator-rich architectures.

V. Evaluation of Prototyping in Commodity FPGAs

A. Floorplan and Area Evaluation

Fig. 16 shows a floorplan of PARC mapped onto the Xilinx Virtex-6 FPGA XC6VLX240T. This is based on a system con-

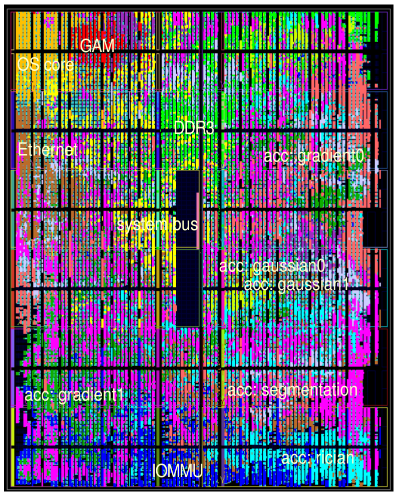


Figure 16: Floorplan of PARC implemented in Xilinx Virtex-6 FPGA.

figuration with six medical imaging accelerators [13] (two named "gradient," one named "rician," one named "segmentation," and two named "gaussian"), and 20 shared SPM banks. The overall system occupies 76% of the total logic resources in the Virtex-6 FPGA. The "crossbar" module is not shown in Fig. 16 since it is

Table II: Significant speed-up and energy savings of the accelerators in PARC.

		segmentation	gaussian	gradient + racian
Xilinx Microblaze Processor in FPGA @ 100MHz	runtime (s)	1.4e3	1.7e2	3.3e3
	energy (J)	546	66.3	129
Dual-Core ARM Cortex-A9 MPCore @ 800MHz	runtime (s)	0.597 (1x)	0.301 (1x)	0.862 (1x)
	energy (J)	0.299 (1x)	0.150 (1x)	0.431 (1x)
Accelerator in Our System in FPGA @ 100MHz	runtime (s)	0.056 (10.7x)	0.066 (4.6x)	0.060 (14.4x)
	energy (J)	0.123 (2.4x)	0.145 (1.0x)	0.132 (3.3x)
8-core Xeon Server E5405 @ 2GHz	runtime (s)	0.405 (1x)	0.109 (1x)	0.106 (1x)
	energy (J)	4.056 (1x)	1.064 (1x)	0.992 (1x)
Accelerator in Our System projected on 45nm ASIC	runtime (s)	0.014 (28x)	0.016 (6.6x)	0.015 (7.1x)
	energy (J)	0.010 (395x)	0.012 (87x)	0.011 (90x)

small (2.8% of total logic) and scattered. Note that we can have more accelerators if there is no resource limitation from the FPGA (i.e., run only the front end part of our flow). The primary purpose of this prototyping is to validate our system design and automation flow in real hardware.

B. Performance Gain and Energy Savings

Table II shows the performance gain and energy savings of the accelerators in PARC. All the computation tasks use the input data of a 128x128x128 3D image. Accelerator kernels "gradient" and "rician" are running together to form the application "denoise" in [13]. Power is measured by McPAT [14] for CPUs and Xilinx xPower for FPGAs. We observed a $>$10x speedup of accelerators running in FPGA-based PARC compared to the ARM processor in ASIC. Note that data paths in FPGAs have to go through lookup tables and routing switches for the sake of programmability, and therefore they are much longer than their ASIC counterparts. The evaluation results are also projected to ASICs based on the gap between FPGA and ASIC reported in [15]. If PARC is further implemented in ASIC, a $>$100x energy savings can be achieved.

VI. Conclusions

This work discusses an implementation study for a general framework of accelerator-rich architectures in an FPGA-based prototyping. We introduced several architecture improvements that address some important system-level design issues including data transfer between userspace and device memories, on-chip memory architecture, and hardware resource management. These improvements are critical when we migrate from CPU core-centric architectures to accelerator-centric architectures. We also developed an automated flow to enable rapid development of accelerator-rich architectures. We run our prototyping in real hardware, and experimental results show that our architecture can fully exploit the $>$100x energy benefits of accelerators.

VII. Acknowledgements

The authors would like to thank Chiyuan Ma for designing a Xilinx dual-port memory wrapper, and Peipei Zhou for experiments on the Xilinx AXI4 bus IP. The authors would also like to thank Xilinx for equipment donation and financial contributions. This work is partially supported by the Center for Domain-Specific Computing (CDSC), and the NSF grant CCF-0903541.

References

[1] J. Benson *et al.*, "Design, integration and implementation of the DySER hardware accelerator into OpenSPARC," in *International Symposium on High-Performance Computer Architecture (HPCA)*, no. Section 3, Feb. 2012, pp. 1–12.

[2] W. Qadeer *et al.*, "Convolution Engine : Balancing Efficiency and Flexibility in Specialized Computing," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[3] J. Cong *et al.*, "Architecture Support for Accelerator-Rich CMPs," in *Design Automation Conference*, 2012, pp. 843–849.

[4] H. Franke *et al.*, "Introduction to the wire-speed processor and architecture," *IBM Journal of Research and Development*, vol. 54, no. 1, pp. 3:1–3:11, Jan. 2010.

[5] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The Accelerator Store: A Shared Memory Framework For Accelerator-Based Systems," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, pp. 1–22, Jan. 2012.

[6] J. Cong *et al.*, "BiN: A Buffer-in-NUCA Scheme for Accelerator-Rich CMPs," in *International Symposium on Low Power Electronics and Design*, 2012, pp. 225–230.

[7] L. Seiler *et al.*, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, vol. 27, no. 3, p. 1, Aug. 2008.

[8] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," *International Symposium on Computer Architecture*, p. 37, 2010.

[9] J. Cong and B. Xiao, "Optimization of Interconnects Between Accelerators and Shared Memories in Dark Silicon," in *International Conference on Computer-Aided Design (ICCAD)*, 2013.

[10] N. Sun and C.-C. Lin, "Using the cryptographic accelerators in the ultrasparc t1 and t2 processors," *Sun BluePrints Online*, no. 819, 2007.

[11] J. Cong *et al.*, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[12] Xilinx, "Vivado High-Level Synthesis." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm

[13] A. Bui, J. Cong, L. Vese, and Y. Zou, "Platform characterization for Domain-Specific Computing," *Asia and South Pacific Design Automation Conference*, pp. 94–99, Jan. 2012.

[14] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *International Symposium on Microarchitecture*, 2009, pp. 469–480.

[15] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb. 2007.