Multi-Accelerator System Development With The ShrinkFit Acceleration Framework

Michael J. Lyons
Harvard University

Gu-Yeon Wei
Harvard University

David Brooks
Harvard University

***Abstract*—This paper introduces the ShrinkFit accelerator framework, which simplifies the design of systems combining multiple accelerators. A single ShrinkFit system design can be deployed to FPGAs large and small, without time-consuming architectural parameter surveys. We describe four ShrinkFit accelerators implemented for an FPGA-based robotic bee brain prototype and demonstrate the flexibility of ShrinkFit with low performance overheads (under 10% on average) and low resource overheads (0-8% for accelerators and under 2% for hard logic blocks).**

I. Introduction

Using FPGAs, custom hardware accelerators can dramatically improve compute performance by targeting individual algorithms amenable to acceleration. These algorithms are typically explicitly parallel kernels, such as DCT and H.264 video decoding. Advances in high level language (HLL) synthesis tools show new promise in simplifying accelerator design: more accessible languages, such as C, can now be used to create and test hardware accelerators in less time. By reducing barriers to entry, these HLL tools can bring hardware accelerator design to a wider audience of developers. And with hybrid processors that combine FPGAs and general purpose (GP) cores [1], [2], programmers can package accelerator designs with applications, and improve application performance over software-only implementations [3]. This enables future systems to run multiple applications, each deploying and using one or more accelerators.

We are currently developing such a system to prototype the electronic "brain" of a flying robotic bee, called RoboBee. This hybrid processor prototype combines a general purpose Cortex-M0 core with accelerators on a Spartan-6 FPGA, and runs a bee application.

During the development of this prototype processor, we found that HLL tools were invaluable, easing the process of implementing accelerators. However, as much of the FPGA community has found [4], [5], [6], combining accelerators into a single system is difficult. Current approaches for designing systems containing multiple accelerators often use HLL tools to create multiple variants in order to find the optimal hardware design. This compute intensive approach explores many combinations of architectural parameters, such as pipeline depth, creating variants for different resource budgets [7]. Unfortunately, building systems with multiple accelerators designed in this way leads to several issues. First, selecting between the many variants of each accelerator is computationally intensive. Second, this process does not easily permit accelerators to share common resources, which may duplicate underutilized logic. Third, there is no way for accelerators to dynamically grow or shrink if resource budgets change. An alternative is to design all accelerators into the system as one combined accelerator, but this approach can further extend design time and is inflexible.

Further, most accelerated systems use shared bus approaches and DMA controllers (DMACs) which are difficult to scale with many accelerators. Shared bus approaches, such as AMBA [8] and Wishbone [9] connect several accelerators over a shared bus. This creates contention when multiple accelerators attempt to communicate over the same bus, so many systems use secondary and tertiary buses for slower accelerators. This may still not alleviate contention on the primary bus, leading to custom accelerator-to-accelerator connections [10].

To address these challenges we created *ShrinkFit*, an extensible framework that facilitates the design of multi-accelerator systems. As the name suggests, ShrinkFit enables accelerators to fit within small FPGA budgets when necessary, and expand to increased resources for additional performance. ShrinkFit also simplifies accelerator, system, and software design. Accelerator designers can implement accelerators once, without complex parameterizations or multiple implementations, and without designing custom interfaces to other accelerators. System design is also simpler, and does not require bus hierarchies or intimate knowledge of accelerator behavior. Software design is simpler as well, since no DMACs need programming. Simpler design for all three areas and resizable accelerators are necessary to manage many accelerators on hybrid systems, where workloads change frequently.

This framework not only applies to our specific RoboBee application, but to the needs of hybrid systems in general. By relying on virtualization [11], ShrinkFit can reduce the computational complexity of allocating resources to each accelerator, allow accelerators to share resources, and could be combined with dynamic reprogramming to support dynamic resizing. The implementation is based on an accelerator store [12], a scalable memory resource for accelerators to save and exchange data without shared buses or DMACs. To support ShrinkFit, we have added: new features to the accelerator store, a "slicer" component for managing data transfers between ShrinkFit accelerators, and an interface that simplifies adding ShrinkFit capabilities to existing accelerators. These capabilities rely on hard logic blocks, dedicated to ShrinkFit and assumed to be built into the FPGA, in the same way FPGAs currently contain dedicated RAM, DSP, and general purpose (GP) hard cores. We also introduce a software interface for building applications with ShrinkFit accelerators.

We demonstrate ShrinkFit's resizing capability and design simplifications with four ShrinkFit accelerators developed for our bee brain prototype. Experimental results show that ShrinkFit enables performance of individual accelerators and the bee application to scale linearly with available FPGA resources. Overheads are low as well: ShrinkFit hard logic block overheads require less than 2% of overall FPGA die area, FPGA resource overheads range from 0%-8%, and application performance overheads are under 10% on average.

II. Motivation

The RoboBees project, a large collaboration between many research groups, seeks to build a swarm of bee-sized, flying robots. Each bee must conserve energy while performing several visual algorithms. To minimize energy consumption while maximizing performance, the RoboBee's brain uses hardware accelerators as well as a general purpose Cortex-M0 processor.

To evaluate the performance requirements for each accelerator, we custom designed a helicopter brain prototype (HBP) circuit board that snaps onto a small helicopter. The HBP contains a low power Spartan-6 SLX150-1L FPGA, which offers the maximum resources within the helicopter's battery power and weight budgets.

To date, we have identified four accelerators useful for the project: image sharpening, edge detection, optical flow (OF), and discrete cosine transform (DCT). The HBP uses all four accelerators simultaneously to process images from the on-board camera, and each image is processed by the three tracks of the RoboBees application (Figure 1). We implemented logic for each of the accelerators using the Vivado C-to-RTL high level language (HLL) compiler [13].

The ShrinkFit framework allows us to take full advantage of our FPGA's resources, whether using a few accelerators early in development or after adding more over the course of the project. We designed ShrinkFit to generally address any FPGA system containing multiple accelerators, including hybrid processors. Throughout this paper, the RoboBee application and accelerators illustrate how ShrinkFit works and evaluate how well it performs.

III. Conceptual approach

To support ShrinkFit, designers decompose accelerators into smaller, reusable "modules" of logic. Once decomposed, these

978-1-4799-6492-5/14/$31.00 ©2014 IEEE

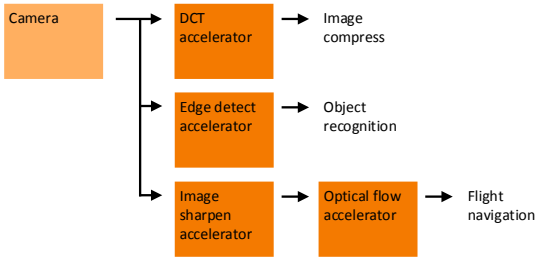


Fig. 1. The RoboBee application utilizes four accelerators. The brain prototype will use the results to compress images for offline analysis, recognize important objects, and avoid obstacles during flight.

modules can be combined to implement resizable accelerators via virtualization, and can be shared between accelerators.

A. Decomposition

ShrinkFit facilitates accelerator designs that are composed of smaller, reusable logic modules. To add ShrinkFit support to an accelerator, it must be decomposed into these reusable logic modules. Depending on the accelerator, different approaches may work best: some accelerators can be decomposed by pipeline stage, others by portions of each entry in a dataset (such as an image or matrix), or perhaps by subsequent entries in a dataset. For example, the image sharpening accelerator, used by the RoboBee brain, could be made up of multiple identical modules that each sharpens one region of an image. Assuming each image has sixteen regions, the accelerator requires sixteen sets of computations. One option would be to program all sixteen modules into the FPGA, each computing in parallel. In contrast, ShrinkFit enables the designer to resize the accelerator at any time by programming anywhere from one to sixteen modules into the FPGA, depending on performance and/or resource constraints. This resizability can shrink the image sharpen accelerator by 94% when only one module is used, but with at least a 16× increase in computation time. Ideally, this relationship between performance and resource utilization should be linear, i.e., doubling resources leads to double the performance.

The key concept behind ShrinkFit is that work performed by each module can be done without programming each module into the FPGA. Rather, the modules programmed into the FPGA need to be capable of doing the work of each module from the original design. To formalize how we decompose and design resizable accelerators, we introduce four terms:

Virtual modules (VMs) are the modules from an accelerator's original design. In the case of image sharpen, which has sixteen regions, the accelerator always contains sixteen virtual modules. VMs represent the work to be done, rather than the logic doing it.

Physical modules (PMs) are the actual logic blocks programmed into the FPGA. As few as one physical module can be programmed into the FPGA. More PMs can be added, but the number of PMs can never exceed the number of VMs. Programming fewer reduces FPGA resource utilization. Programming more PMs increases performance.

Module designs refer to the algorithm a PM or VM implements. PMs of the same module, such as DCT, use the same RTL and are identical, whereas PMs of different module designs are not interchangeable. A convolution PM cannot do the work of a DCT PM.

Module contexts contain the information a PM needs to act as a VM. Section III-B describes contexts in detail.

B. Module contexts

In all ShrinkFit accelerators, VMs are an abstraction representing the work to be done. It is the PM, logic programmed into the FPGA fabric, that performs the computations. To bridge the gap between work and logic, PMs use "contexts," which are blobs of data that instruct a PM how to act as a VM. Each VM has a corresponding context, and by loading it, a PM can act as its corresponding VM. For example, a convolution context contains the image region its VM corresponds to. When a convolution PM loads a context, it immediately knows which region of each image to process.

Because there may be fewer PMs programmed into the FPGA than VMs in the accelerator design, these PMs must routinely switch to perform the computation of different VMs within the accelerator. This process is known as "context switching," because a PM will do the work of one VM for a short period of time, then switch contexts to do the work of another VM. This approach ensures that the work of all VMs will be completed regularly no matter how many PMs are programmed into the FPGA.

C. Accelerator resource sharing

Because PMs do not necessarily belong to one accelerator or another, they can be shared between accelerators. For example, image sharpen and edge detect accelerators both use sixteen convolution VMs. If a system used both accelerators simultaneously, each convolution VM would have a corresponding context, resulting in a total of 32 convolution contexts. And because the system contains 32 convolution VMs, one to 32 convolution PMs could be programmed into the FPGA. However, if fewer PMs exist in order to reduce FPGA resource utilization, they would all take turns context switching into all 32 contexts, and do the work of all 32 convolution VMs in both accelerators. Rather than allocating different PMs to each accelerator, the accelerators share all PMs.

In rare cases, a system designer may wish to dedicate certain PMs to a single accelerator, rather than sharing PMs between accelerators. This is easily accomplished by creating two sets of contexts for the module design, and mapping some PMs to one set, and the remaining PMs to the other set.

IV. Framework Implementation

There are many ways to implement the conceptual approach described in the previous section as long as the implementation supports many PMs, enables PMs to context switch rapidly, and delivers PM input and output datasets quickly. With these requirements in mind, we present our implementation of the ShrinkFit concept.

Systems utilizing the ShrinkFit architecture include a general purpose core, several PMs, and ShrinkFit's hard logic blocks. Figure 2 provides a detailed illustration of how ShrinkFit is used for the RoboBees application. After PMs are programmed into the FPGA fabric, the general purpose core configures PMs and various ShrinkFit hard logic blocks use the system bus to perform reads and writes as if the general purpose core was reading and writing to memory.

All workload-specific circuits (physical modules) are implemented in reconfigurable FPGA fabric. Only generic logic (used by all workloads) is permanently fabricated as hard logic blocks. This philosophy matches current FPGA designs which include hard logic blocks for SRAM memories and commonly used DSP operations.

ShrinkFit depends on three types of hard logic blocks: the accelerator store, slicer, and ShrinkFit wrapper. The accelerator store provides a central location to store input and output data processed by accelerators as well as virtualization information, the slicer provides a resource for virtual modules to coordinate when processing the same pieces of data, and ShrinkFit wrappers manage virtualization tasks for each physical module. Each of these tasks apply to all accelerator designs, and each hard logic block is completely generic. They do not have any specializations for specific accelerators or workloads.

A. Accelerator store

To facilitate data storage and movement between PMs, we employ the accelerator store [12], which was originally proposed as a replacement for SRAMs located inside accelerators and as a method to transmit data between accelerators. We found that with some modifications, the accelerator store can be used to manage contexts and relay data between PMs. For maximum performance, accelerators often require fast access to input and output datasets. With this in mind, the accelerator store maintains direct connections with every PM programmed on the FPGA via ASPorts (Figure 2) to ensure low-latency communication.

1) Handles: Memory in the accelerator store (AS) is allocated using "handles." Similar to file handles in most operating systems, handles represent a block of memory in the AS. To access handles, VMs specify the handle's corresponding handle ID (HID) number. VMs can use handles to communicate: after the first VM stores data in the handle, the second VM can access it.

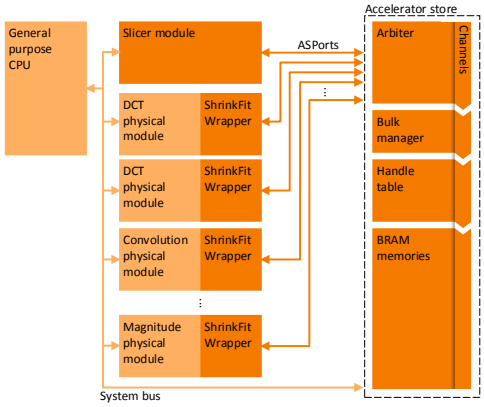


Fig. 2. The ShrinkFit framework consists of the accelerator store, slicer, and ShrinkFit wrappers. These components, darkened above, are permanently fabricated in the FPGA as hard logic blocks to obtain low die area overheads. All physical modules are programmed into the FPGA fabric as soft logic blocks. One or more general purpose cores may be implemented as hard or soft logic blocks.

The accelerator store provides two interfaces for accessing handles: FIFO queue and random access (RA). FIFO handles allow PMs to send individual words to each other, to be processed in order. This is helpful for short inputs or outputs that can fit in a single word, or as a method for VMs to relay commands to each other. RA handles, which support reads and writes at a specified address, are useful for storing larger datasets.

Handles are configured by the general purpose core in software, using the system bus to allocate handles before accelerators are started. However, if the system needs to modify the handle configuration, the same system bus interface can be used while the system is running. We currently allocate handles by hand, but future work could leverage memory allocators such as malloc to automate this process at runtime.

2) Bandwidth and arbitration: The accelerator store supports systems containing many PMs and each may make requests simultaneously. The AS has an arbiter to satisfy as many PM requests as possible, and reject any remaining requests. The arbiter uses a round-robin scheme to prevent starvation. If the request is rejected, the PM can try the request again on the next or later cycle.

We represent AS bandwidth in terms of "channels," which indicates the number of requests it can satisfy per cycle. An AS with one channel can satisfy one PM's request per cycle, a two channel AS can handle requests from two PMs in the same cycle, and so on. The AS, implemented in Verilog RTL, fully parameterizes the channel count and can be provisioned with more or fewer channels as desired. In our experience, provisioning more than three channels had negligible performance benefits.

3) AS architecture: Overall, there are three stages within the accelerator store. The first is arbitration, which takes requests from each accelerator and selects a subset based on a round robin selection scheme. In the second stage, the bulk manager and handle table decode accelerator requests into SRAM accesses. Finally, the decoded accesses are sent to a bank of small (2KB-4KB) SRAMs, the results of which are returned to accelerators.

Each accelerator has a direct connection to the accelerator store, and requests only add an additional cycle of latency (without contention). This low latency is critical for maintaining the high performance needed by accelerators. PM context switching would be especially affected by larger latencies, increasing the amount of time spent finding VMs and less time computing.

Additional details about the AS architecture are in [12].

4) Generic design: To support ShrinkFit, three features were added to the AS: bulk transfers, random bulk access, and swap operations. Bulk transfer support (sending multiple words with a single request) was added to reduce arbitration overheads and are already common in DMACs for the same reason. AS support for accelerators to specify custom addresses was also added for bulk transfers. This was particularly helpful when PMs accessed regions of an image: images are stored in raster order, so regions are not stored contiguously. We opted for a generic solution: to let accelerators specify any access pattern. Lastly, support for the swap operation (common on general purpose CPUs) was introduced. Swaps are critical for ShrinkFit context switching and applies to all workloads.

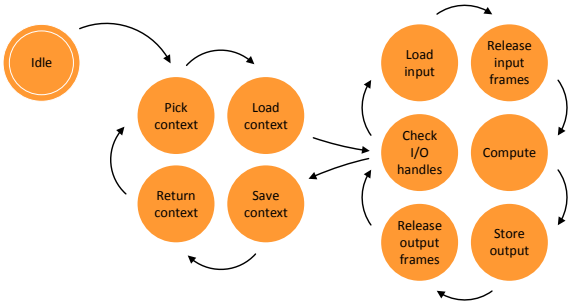


Fig. 3. Each ShrinkFit wrapper implements this state machine to context switch, load inputs, trigger computation, and store resulting outputs. Wrappers initially enter the idle state on startup.

B. Slicer module

The ShrinkFit framework includes a *slicer* module that interacts with the AS and PMs and ensures all VMs can properly access input data or store output data in RA handles. For example, the edge detect accelerator contains sixteen convolution VMs, which output images to sixteen magnitude VMs. To function correctly, all sixteen convolution VMs must each produce their regions of the output images before the magnitude VMs can consume them. In addition, the magnitude VMs must all consume these images before the convolution VMs can overwrite them with new output images. Because there are multiple VMs producing data and a different set of VMs consuming data, a FIFO handle would not suffice: as soon as one VM performed a get, the data would be lost to the other VMs. Instead, the slicer assists the VMs in ensuring that all VMs can reliably produce and consume data from RA handles.

To improve performance, the slicer supports varying amounts of buffering. Buffering is quantified in "buffer slots," which measure the number of entries that can be stored in the RA handle. For example, a handle with room for four images is sized to four buffer slots. By increasing the size of a handle to buffer more slots, a PM can batch more computation and improve performance.

Each buffer slot is in one of two states: produce or consume. When producing, one or more VMs write portions of the buffer slot. When all VMs finish producing, the buffer slot switches to the consume state. On each state switch, the slicer notifies the relevant VMs to resume computation.

If the handles are sized for multiple buffer slots, VMs can produce and consume simultaneously from the handle (but not the same buffer slot). One buffer slot may be in a produce state and the other in a consume state.

Ultimately, the slicer is responsible for tracking how many virtual modules have produced and consumed each buffer slot of data.

C. ShrinkFit wrapper

The ShrinkFit wrapper is a small hard logic block that connects a PM to the accelerator store's ASPort. The wrapper implements common ShrinkFit tasks, including context switching, loading inputs, storing outputs, and interacting with the slicer (Figure 3). We initially implemented this logic within each PM, but noticed we were duplicating the same logic across all PMs. Hence, we decided to refactor the common logic into a generalized hard logic block.

1) ShrinkFit wrapper contexts: The wrapper defines a common context handle structure (Figure 4). Each module design has its own RA context handle, with a context for each of its VMs. The first words in the context handle each correspond to a different VM, forming a context directory. Each of these directory entries contain the location

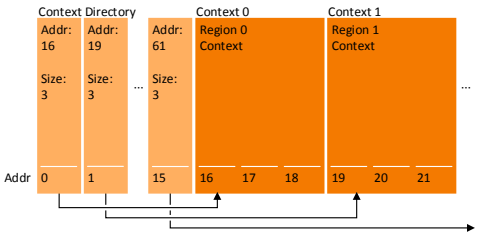


Fig. 4. ShrinkFit wrapper context handles contain a context for each VM. A context directory at the start of the handle identifies the locations of each context.

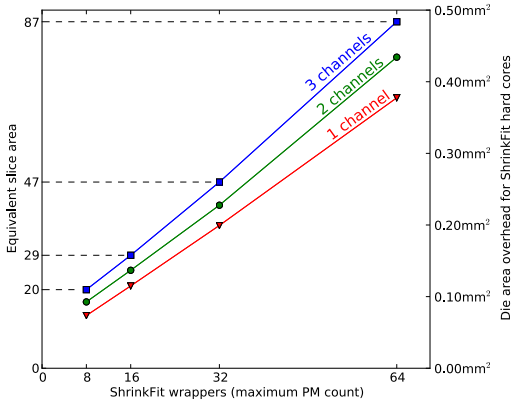


Fig. 5. ShrinkFit hard logic block die area overheads are low for systems with many accelerators and well-provisioned bandwidth. Hard logic blocks are synthesized for a commercial 40nm process technology, and area is expressed in mm^2 as well as normalized to the area of a Spartan-6 FPGA slice.

of the VM's context data and the length of the data. We chose this approach to support variable-length contexts, rather than hard coding context data lengths to a set size.

This context directory approach allows wrappers to quickly check if a context is claimed by another VM. A slicer tries to claim a handle by swapping the context's directory entry with an invalid entry. If the wrapper receives an invalid entry back, it knows another PM has already claimed this context, and tries to claim the next directory entry. If the PM receives any other value back, it knows it has successfully claimed the context. The wrapper then uses information in the directory entry to load the context. When the PM is done using the context, it saves any changes back to the context data, then writes the original directory entry back to the context directory. Although more complex context iteration schemes are certainly possible, we found that the ShrinkFit wrapper's current implementation resulted in excellent context switching performance in practice.

2) ShrinkFit wrapper input/output: After loading the context, the ShrinkFit wrapper first checks if input handles have enough data to consume, and if output handles have enough space to produce the results into. If checks fail, the context is returned and a new one is chosen. Otherwise, the wrapper begins loading from input handles.

D. ShrinkFit framework area costs

Because the ShrinkFit framework is generalized, rather than designed for a specific set of accelerators, it makes sense to hard code it into the die, rather than programming it into the FPGA. ShrinkFit also provides several opportunities for PMs to override automatic features if they desire a custom solution.

By utilizing hard logic blocks, the ShrinkFit framework has low area overhead. We synthesized all three hard logic blocks—AS, slicer modules, and ShrinkFit wrapper—using Design Compiler D-2010.03 for a commercial 40nm process. Assuming a three-channel AS with ASPorts for 64 PMs, which is more than sufficient to maximize RoboBee application performance (only 36 are needed), Figure 5 plots the die area overhead versus the number of ShrinkFit wrappers. For comparison to reconfigurable resources, we used a commercial 40nm memory compiler to find the area of an SRAM equivalent to the 32x512, dual-port BRAMs found in the Spartan-6 ($0.0276mm^2$), and used this to express the hard logic block area in terms of slices, the basic building block of FPGAs (this calculation is described in more detail in Section VI-A). Using the equivalent slice area in Figure 5, Section VII-D later shows that the ShrinkFit hard logic block die will consume less than 2% of the FPGA's die area in both small and large FPGAs.

V. Software Development

Developing applications requires the system designer to perform two tasks: decomposing accelerators and configuring the ShrinkFit hard logic blocks. To simplify the latter step, we implemented shrinklib, a software development kit (SDK) for the Python programming language.

A. Decomposing accelerators

Before any software can be implemented, the accelerators in the application design (Figure 1) must be decomposed into their corresponding VMs (Figure 6). For example, the edge detect accelerator decomposes into sixteen convolution VMs and sixteen magnitude VMs. This decomposition will also include handles to represent data connections, such as the images produced by convolution VMs and consumed by the magnitude VMs.

The application designer must decide how many buffer slots to allocate to each handle and calculate how many VMs produce and consume from each handle. Contexts for each module design are stored in RA handles, so handles must be allocated to store contexts as well.

B. Configure ShrinkFit hard logic blocks

The software program uses system bus reads and writes, just as it would to read from or write to memory, to configure ShrinkFit hard logic blocks in five steps:

1) Create handles (to connect VMs and store contexts)
2) Configure the slicer with the number of VMs that produce and consume each handle
3) Store contexts for each VM in context handles
4) Configure each PM's ShrinkFit wrapper
5) Start each PM's ShrinkFit wrapper

C. Shrinklib SDK

The shrinklib SDK includes routines to automate the application development steps after decomposition. Although shrinklib is currently implemented in the Python programming language, it simply performs system bus reads and writes, and is easily ported to other languages. We use the RoboBee application to show how shrinklib is used below.

The following code configures all DCT modules (steps 1-5):

```
dct_ctxt_handle = shrinklib.AsHandle(spi, hid=13,
  start_addr=0x0001F800, word_count=512)
virt_dct2_set = shrinklib.VirtDct2Set(spi=spi,
  slicer=slicer, physical_module_count=dct_pm_count,
  context_handle=dct_ctxt_handle)
virt_dct2_set.AddContexts(in_handle=camera_image_handle,
  out_handle=dct_coefs_handle)
virt_dct2_set.CommitInit()
virt_dct2_set.StartPhysicalModules()
```

The code to initialize the other four PM designs is almost identical.

Adding support to shrinklib for new PM designs is straightforward. The module designer only needs to write two methods for the new design. The first routine uses the system bus to configure each of the ShrinkFit wrappers for the module design, the second builds the context handle for each module design.

VI. ShrinkFit Module Evaluation

PMs are the building blocks of all ShrinkFit accelerators, and by extension, the applications that use them. Before evaluating the RoboBee application and the four ShrinkFit accelerators it uses, we consider the five PMs they are built from—convolution, magnitude, DCT, optical flow (OF) region, and OF merge. Specifically, we evaluate two aspects of the PMs. First, performance scales up linearly

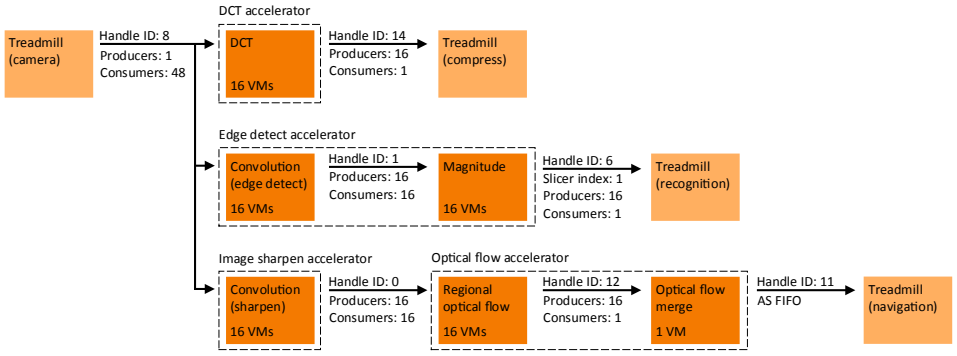
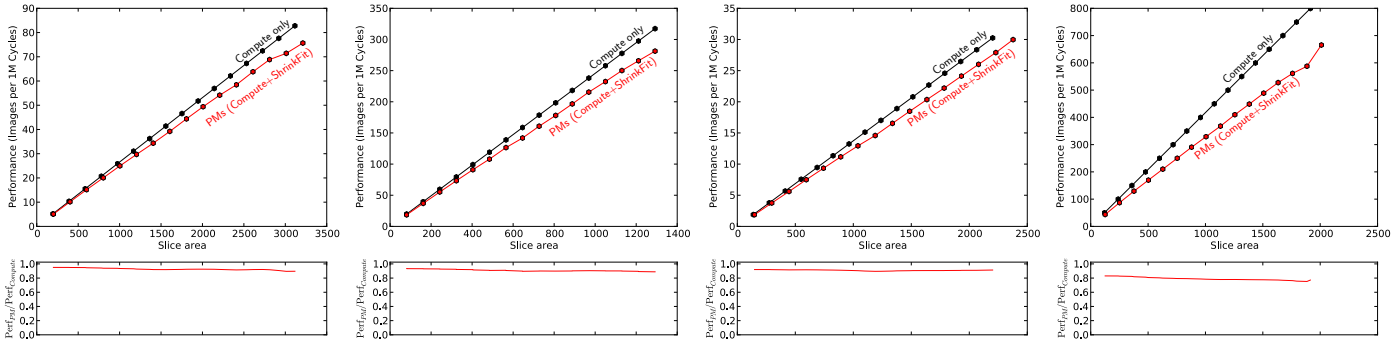


Fig. 6. Prior to implementing software, each ShrinkFit accelerator in the RoboBee application is decomposed into VMs and handles

	Slice area			*Registers*			*LUTs*			*DSPs*			*BRAMs*		
	Compute	*+PM*	Δ	*Compute*	*+PM*	Δ	*Compute*	*+PM*	Δ	*Compute*	*+PM*	Δ	*Compute*	*+PM*	Δ
Convolution	195	201	+3.0%	463	495	+6.5%	680	702	+3.1%	2	2	+0.0%	3	3	+0.0%
Magnitude	81	81	+0.0%	92	101	+8.9%	221	221	+0.0%	2	2	+0.0%	3	3	+0.0%
DCT	138	149	+7.8%	409	436	+6.1%	391	436	+10.3%	4	4	+0.0%	4	4	+0.0%
OF region	120	126	+5.0%	178	208	+14.4%	337	361	+6.6%	5	5	+0.0%	2	2	+0.0%
OF merge	261	261	+0.0%	772	772	+0.0%	863	864	+0.0%	4	4	+0.0%	5	5	+0.0%

TABLE I. **MODULE DESIGN FPGA RESOURCE OVERHEADS.** FPGA RESOURCE OVERHEADS (Δ) FOR ALL FIVE MODULE DESIGNS ARE LOW, FROM NONE TO +7.8%. "SLICE AREA" EQUALS THE SUM AREA OF FPGA PRIMITIVES (REGISTERS, LUTS, DSPS, AND BRAMS) RELATIVE TO THE AREA OF A SPARTAN-6 FPGA SLICE. EACH SLICE CONTAINS 16 REGISTERS AND 8 LUTS. DSPS AND BRAMS ARE EACH 4.95X THE AREA OF A SLICE.



(a) Convolution PMs (b) Magnitude PMs (c) DCT PMs (d) OF region PMs

Fig. 7. All four systems contain one to sixteen PMs of the same module design, demonstrating that ShrinkFit scales performance with FPGA resources, as well as low resource and performance overheads. Each configuration processes 100 images using a 3-channel AS. "Compute only" considers only the compute logic without ShrinkFit overheads. PM performance considers ShrinkFit glue logic resource and performance overheads, in addition to compute logic costs. Slice area considers the area consumed by registers, LUTs, DSPs, and BRAMs, relative to the area of a Spartan-6 slice. Secondary plots compare $Perf_{PM}/Perf_{Compute}$, the ratio between the two series.

with FPGA resources as more PMs are added. Second, regardless of whether a few or many PMs are programmed into the FPGA, performance overheads and resource overheads remain low.

A. ShrinkFit PM implementations

For each of the five RoboBee PM implementations we partition compute logic and ShrinkFit functionality into separate blocks. Compute logic, designed using the Vivado C-to-RTL compiler [13], only contains the logic required to perform the PM's computation and BRAM memories to hold input and output data. In other words, the compute logic blocks do not contain any optimizations for ShrinkFit interfacing or functionality. To add ShrinkFit support, each PM has additional "glue logic" to connect the compute logic block to a ShrinkFit wrapper. Since the wrapper performs most of the tasks related to ShrinkFit, the glue logic simply connects the wrapper to the corresponding PM and takes care of any special cases. For example, the convolution module may produce outputs to one handle if performing image sharpening, or two handles if performing edge detect. Convolution glue logic guides the wrapper to accommodate this choice properly.

In all five RoboBee PM designs, glue logic resource requirements are small compared to their corresponding compute logic blocks, between 0.0% and 7.8% (Table I). These low resource overheads are largely thanks to the ShrinkFit wrapper, which implements the majority of ShrinkFit functionality as a hard logic block. In order to compare the glue logic and compute logic resource costs, we synthesized each PM using Xilinx ISE 14.4 for the Spartan-6 FPGA used in the HBP, with and without glue logic. Like most FPGAs, the Spartan-6 contains four basic primitives: registers and lookup tables (LUTs), DSP blocks optimized for addition and multiplication (the same MAC blocks used by DSP processors), and BRAM blocks for efficiently storing large datasets. Registers and LUTs are contained within slices, of which hundreds if not thousands exist in the FPGA.

In addition to counting these primitives (register, LUT, DSP, and BRAM) in each design, we calculate a "slice area" resource cost combining primitives into a single metric. This metric packs registers and LUTs into slices (16 registers and 8 LUTs per slice) to obtain a slice count, and determines the die area of DSPs and BRAMs relative to the area of a slice (DSPs and BRAMs consume the area of 4.95 slices, according to PlanAhead). Total slice area is the sum of slices used by registers and LUTs, combined with the relative slice area of DSPs and BRAMs.

In the case of magnitude and OF merge modules, glue logic adds no overheads. This is due to underutilized slices in compute logic blocks, containing too few registers or LUTs to completely pack slices. The wrapper logic is able to utilize the unused primitives without adding additional slices.

B. Evaluation methodology

To analyze scalable PM performance and overheads, we analyzed four system design scenarios, each limited to only use one out of the four module designs: DCT, convolution, magnitude, and OF region. We do not analyze OF merge, the fifth module design, because only one can be used per optical flow accelerator. For each system, we considered different PM counts, ranging from the minimum (1) to the maximum (16). These systems also contain an accelerator store with three channels of bandwidth and four buffer slots per handle.

Our performance analysis relies on ModelSim 10.1, which performs cycle-accurate simulation of all runs using synthesizable RTL for all hard logic blocks and PMs. Each system utilizes a "treadmill" testing module, ensuring repeatable and error-free tests. The treadmill module injects pre-recorded camera images into the system in the same order and verifies PM outputs using corresponding checksums. Performance is determined by measuring the number of cycles required to process 100 images. To evaluate ShrinkFit performance overheads, we compare each PM's performance with the compute logic block's performance. We also record the subset of cycles spent utilizing the compute logic block. This measurement reveals the theoretical maximum performance of the PM without any ShrinkFit overheads (Table II). This represents the maximum performance for any alternative system interconnect using the compute logic in each PM as it does not consider any ShrinkFit overheads. Dividing actual ShrinkFit system performance by this theoretical maximum ($Perf_{PM}/Perf_{Compute}$) reveals the performance overhead of the ShrinkFit system.

We also verified that our implementations of the ShrinkFit framework RTL and PM RTL correctly synthesize and work in the HBP hardware. However, the HBP platform could not be used for thorough performance analysis because the off-the-shelf Spartan-6 FPGA does not include ShrinkFit hard logic blocks. Instead, all ShrinkFit features were programmed in as soft logic blocks, consuming much more of the FPGA's resources. This limited the number of PMs that could be added and introduced additional overheads that would not occur with ShrinkFit hard logic blocks.

C. PM performance scalability

Since all ShrinkFit accelerators are built with PMs, it is important to ensure they work well individually within the framework before combining them to compose accelerators for different applications. Hence, we evaluate the performance scalability of individual PM implementations first. Figure 7 plots the performance versus resource utilization (slice area) for the four resizable types of PM designs. For each of the PMs, the plots show how performance scales for "Compute only" and "PMs." "Compute only" data points correspond to compute logic without any other resource or performance overheads. This is a highly optimistic upper bound which does not consider the costs of context switching, loading input data to process, or storing the resulting output data. However, many of these overheads would be present whether or not ShrinkFit is used. For example, inputs and outputs will always need to be loaded. The "PMs" data points include these performance and soft logic resource overheads in order to evaluate how PMs perform in an actual ShrinkFit system. Each data point in Figure 7 represents a system with a different number of PMs. The points at the far left represent a system with a single PM. Proceeding to the right, each consecutive point uses more FPGA resources to add an additional PM. This continues until reaching the rightmost point, representing a system utilizing a maximum 16 PMs. The corresponding $Perf_{PM}/Perf_{Compute}$ plots show how overheads scale again with respect to resource utilization for all four PMs.

The results demonstrate that all of the PMs successfully achieve a linear performance-to-resource relation, with some minor exceptions. For each PM added to the system, performance roughly increases by a nearly constant factor. As the number of PMs increase, the slightly lower slope can be attributed to contention in the accelerator store. The $Perf_{PM}/Perf_{Compute}$ plots better illustrate this trend. While a slight downward slope can be seen for all PMs, these plots verify that ShrinkFit overheads are consistently low even as more PMs are added. The larger overheads for OF region are due to large inputs (regions of an image) and short execution times (Table II). OF region spends a considerable portion of its time loading inputs, a delay that would occur even if system architectures other than ShrinkFit were to be used. For this reason, and considering that the average $Perf_{PM}/Perf_{Compute}$ is still high at 78.96%, the PM implementation of OF region, and all other module designs, is sufficient for the goals of ShrinkFit.

	Cycles per image region			
	Average	*Minimum*	*Maximum*	*Max-Min* Δ
Convolution (sharpen)	13,520	12,665	14,403	12.1%
Convolution (edge detect)	12,079	11,315	12,867	12.1%
Magnitude	3151	2600	3250	20.0%
DCT	33,061	33,061	33,061	0.0%
OF region	1251	1173	1331	11.9%
OF merge	530	530	530	0.0%

TABLE II. **COMPUTE LOGIC BLOCK PERFORMANCE.** SHRINKFIT IS NOT AFFECTED BY BLOCK DELAY VARIABILITY.

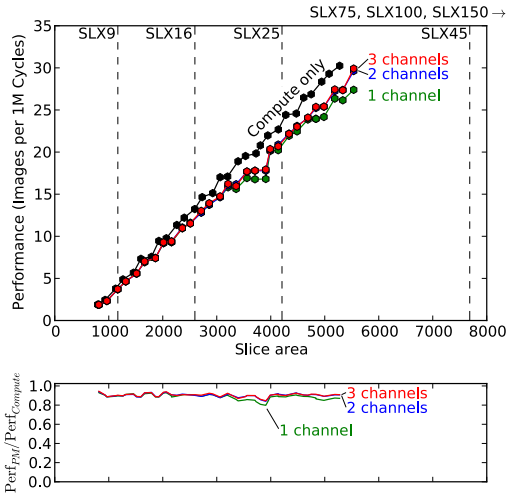


Fig. 8. When running full applications, AS bandwidth needs are low: two channels of AS bandwidth is sufficient for the RoboBee application, and one channel performance is only slightly less. Systems with varying PM counts and one, two, or three channels of AS bandwidth are considered. FPGA resources for Spartan-6 FPGA models are noted at the top.

VII. ROBOBEE APPLICATION EVALUATION

Given that PMs can scale performance with FPGA resources, and do so with low area and performance overheads, we now consider the case where PMs are combined to form four accelerators and implement the RoboBee application shown in Figure 6. The system should continue to scale performance with FPGA resources while adding more PMs into the system. The following evaluation not only demonstrates this is achievable, but that overheads also remain low. Further, results demonstrate that less channel bandwidth is necessary to support the RoboBee application than some PMs require when considered individually. Finally, we investigate the role of buffering in regards to performance.

All runs of the RoboBee application follow the same testing approach as with single module evaluations. The treadmill module is used to inject test images and verify output checksums. For each test, we measure performance as the number of cycles required to completely process 100 camera images. All test results are obtained from cycle accurate RTL simulations. In addition, we implemented a system using one PM of each of the module designs and deployed it to the RoboBee brain prototype FPGA. We successfully ran the application on the FPGA prototype, verified error-free functionality, and validated that on-FPGA and simulation results matched.

A. Application evaluation overview

We first consider ShrinkFit systems processing the RoboBee application. Like the previous figure, Figure 8 plots performance versus slice area for "Compute only" and three "PM" implementations with different AS bandwidth assumptions (1, 2, and 3 channels). Again, "Compute only" data points only include compute logic resource

and performance costs and do not include ShrinkFit overheads due to context switching, loading input data, or storing output data. Each point in the plot represents a different set of PMs. Points on the left side represent the minimum set of PMs, one each of the five module designs (convolution, magnitude, DCT, OF region, and OF merge) required by the application. This configuration requires the fewest FPGA resources possible. Each point to the right adds an additional PM to the system. Unlike the previous evaluations that considered a single module design, the system designer must decide which of four PMs to add to the system (because only one OF merge VM is used for the optical flow accelerator, there is no benefit from adding additional OF merge PMs). To decide which of the four PMs to add, we use each PM's compute-only average performance (Table II) as an approximation for the PM's performance and use these approximations to exhaustively calculate the expected performance of each possible system configuration. We believe this estimation is close enough to make good decisions, due to the previous section's findings that PM performance overheads are low. From the roughly 130,000 possible PM permutations, the highest performing configurations were progressively chosen for each subsequent point with increasing slice area up to the maximum number of PMs that maximize the overall performance of our application. This exhaustive search required less than one second on a typical desktop computer. More efficient search strategies are certainly possible, but the exhaustive approach performed well in practice for our workload.

The plot clearly shows that ShrinkFit again enables performance to scale up with increasing FPGA resource utilization. The somewhat jagged data points in the plot is an artifact of the different resource and performance characteristics of the PMs. Each module design's PM implementation consumes different FPGA resources and requires different amounts of time to complete their operation. In other words, the performance gained and resources consumed by adding a PM varies between module designs. In addition, applications may chain VMs in series, and adding an additional PM will only improve performance until it alleviates the critical path, shifting it to a different PM design. Although the point-to-point relation is jagged, the application's overall trend continues to be roughly linear.

These results verify that performance overheads are low when processing our application across the full range of slice area, as seen by the $Perf_{PM}/Perf_{Compute}$ plot. Although there is slightly more variance in this ratio than with single module evaluations, it is still relatively flat. In addition, the ratio is always high, on average at 90.34% for systems with a three channel AS, indicating performance overheads are low.

B. Bandwidth impact

Experimental results in Figure 8 additionally reveal that bandwidth has less of an effect on performance than for single module systems. Some of the module designs in the single module evaluation required an AS with three channels to achieve high performance. However, when considering the full application, three channels only improve performance over two channels by 0.20% on average, an insignificant difference. Even one channel performance is only 1.72% less on average than with three channels.

Bandwidth needs for the application are lower than for single modules because needs are determined by the slowest modules, not the fastest. OF region PMs require up to three channels when large numbers of PMs are present since computation is so fast that loading input images (utilizing bandwidth) requires a significant portion of the PM's cycles. But because the OF region PMs are so fast, they are rarely on the critical path when slower PMs, such as DCT, are present. As a result, the application never needs to program enough OF region PMs to require three channels.

These results also demonstrate the efficacy of PM sharing between ShrinkFit accelerators. Both edge detect and image sharpen accelerators make use of convolution PMs. These PMs are not partitioned to one accelerator or the other, rather, all PMs rapidly switch between both accelerators. Further, the lowest resource configurations use a single convolution PM, shared by both accelerators. Despite switching between both accelerators, the system achieves high performance.

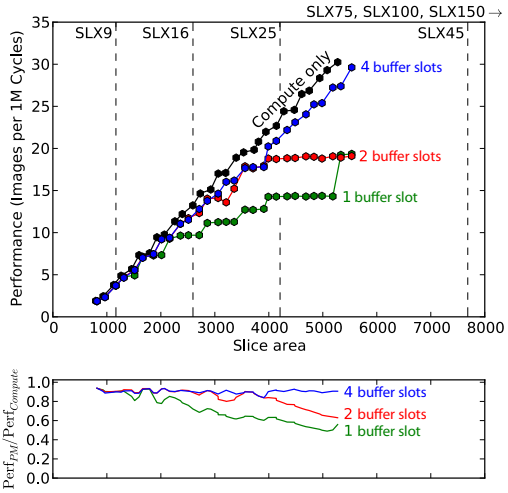


Fig. 9. Buffering is essential for high performance with many PMs: four buffer slots per handle is necessary to scale performance with upper PM counts. Systems with varying PM counts and one, two, or four buffer slots per handle are considered. FPGA resources for Spartan-6 FPGA models are noted at the top.

ShrinkFit quickly identifies when the system achieves maximum performance and adding additional PMs would waste resources. We do not consider PM configurations that consume 16x more resources than the minimum sized configuration, because maximum performance is achieved well before this point. Like bandwidth, performance is dictated by the slowest module limiting the critical path. Therefore, when the slowest module programs its maximum number of PMs into the FPGA logic (sixteen DCT PMs in the RoboBees application) the critical path cannot be reduced by adding PMs from other module designs. Once the maximum number of PMs for a module design have been programmed into the FPGA, it is quickly apparent that there is no need to consume more FPGA resources with other PMs. In the case of the RoboBees application, this occurs with the following PM counts: 16 DCT PMs, 13 convolution PMs, 2 magnitude PMs, 1 OF region PM, and 1 OF merge PM. By quickly identifying this maximum performing configuration, ShrinkFit prevents programming additional, unnecessary logic.

C. Buffering impact

In contrast to bandwidth insensitivity, Figure 9 shows buffering has a significant effect on performance as PM count grows. For this experiment, we used systems with the same PM selections as in the bandwidth evaluation. However, instead of varying bandwidth, we sized each handle to hold either one, two, or four buffer slots. In systems with low PM counts, $Perf_{PM}/Perf_{Compute}$ remains high regardless of buffer size, indicating low performance overheads. However, as PM counts rise, buffering less than four buffer slots constrains performance and lowers $Perf_{PM}/Perf_{Compute}$.

Buffer size has significant impact on performance due to pipelining effects between modules connected in series. For example, if a handle is sized for a single buffer slot, that buffer slot can only be used to produce or consume at any given time. Therefore, for modules on the critical path, only half may be actively processing at any time. This effect is lessened for handles sized for two buffer slots, which allows connected VMs to produce and consume simultaneously. Still, it is unlikely that both VMs will complete producing and consuming at exactly the same time, and therefore one will stall while the other completes. Using handles with room for four buffer slots decouples modules in series, and improves pipelining performance accordingly. Because these limitations would apply to any accelerator based system, the experimental results show that buffering is important for any accelerator based architecture and does not apply solely to the ShrinkFit framework.

To obtain a direct comparison between "Compute only" and actual PM performance while investigating buffering, we do not

include the resource overhead of using different levels of buffering in Figure 9. "Compute only" assumes an unlimited amount of buffering, and because any accelerator based system would require amounts of buffering at least equivalent to ShrinkFit, omitting the resource cost results in the fairest comparison. Using four buffer slots rather than one requires an additional area equivalent to 159 Spartan-6 slices, and is only necessary to increase buffering when larger PM counts are used (and more resources are available on the FPGA). In practice, the cost of adding additional buffering consumes a few percentage points of resources, and would apply equally whether or not ShrinkFit was used.

D. Hard logic block area overheads

Smaller FPGAs, such as the Spartan-6 SLX9 can only fit systems with a few PMs, whereas the larger SLX75 can fit enough to obtain the maximum achievable performance [14] (Figures 8, 9). As such, the SLX9's ShrinkFit framework would need to provision support for fewer PMs, which would reduce the percentage of FPGA die area consumed by ShrinkFit hard logic blocks. If Spartan-6 processors were to include ShrinkFit hard logic blocks, we would propose provisioning 16 PMs for the SLX9, 32 PMs for the SLX16 and SLX25, and 64 PMs for larger Spartan-6 FPGAs. For flexibility, more ShrinkFit wrappers are provisioned than used for the RoboBee application. For the SLX75 and up, almost half of the 64 wrappers are unused and available for future expansion. Using data from Figure 5, the equivalent slice area of ShrinkFit hard logic blocks is small by comparison. With this provisioning scheme, ShrinkFit hard logic blocks require less than 2% of reconfigurable die area for small and large Spartan-6 FPGAs.

VIII. Related work

Several projects developed coarse-grained accelerators. In some cases, the research contribution is in optimizations made for a particular workload [15], in others it is analyzing hardware accelerator design using one accelerator as a test case [16].

Several works target multiple resource budgets by creating variants of the same accelerator. Cong, et al., use the Vivado C-to-RTL compiler to survey architectural parameters to create accelerator variants for different resource budgets [17]. As discussed in Section I, this approach leads to challenges when building multi-accelerator systems. Elastic computing manually designs multiple variants of the same accelerator using different algorithms [18]. This approach is difficult to scale as it depends on significant manual design and the existence of multiple implementations of the same algorithm.

Other works have investigated approaches to manage multiple accelerators in a single system. Dales, et al., introduced an approach for a hybrid FPGA+GP processor to switch between accelerators and software execution [19]. This work does not use accelerator variants or resize accelerators. FPMR adds hardware support for MapReduce algorithms to resize an accelerator [20]. This work is limited to the use of a single accelerator and algorithms which fit within MapReduce semantics. CHARM, DRP, and PipeRench use generic hard compute logic blocks instead of FPGA slices to create reconfigurable accelerators [21], [22], [23]. SHARC presents a streaming model using multiple accelerators and a context structure for configuring accelerators [24]. However, SHARC's model hard-codes data connections between accelerators, in contrast to ShrinkFit's use of the accelerator store to buffer intermediate data. SHARC's approach therefore prevents resizability and logic sharing between accelerators that is possible using ShrinkFit.

Previous works have used virtualization concepts to support hardware acceleration. Kalte, et al., use contexts to store accelerator state, but for pausing or relocating accelerators in FPGA fabric rather than resizing them [25]. C-Cores introduced an approach for ASICs that automates software-hardware codesign using virtualization-like state management but is limited to single accelerators [26]. AXR-CMP manages accelerators within a virtualized memory space and supports the general purpose CPUs virtual memory spaces. However, AXR-CMP does not virtualize accelerators, accelerators cannot automatically context switch, and general purpose CPUs are required to program many DMACs[27].

IX. Conclusion

Despite HLL tool improvements, combining accelerators in a single system was challenging. Our evaluation demonstrated that ShrinkFit addressed this problem, and is able to linearly scale performance with FPGA resources. We also demonstrated hard logic block die area overheads (less than 2%), FPGA resource overheads (0-7.8%), and performance overheads (less than 10% on average) were all low. With ShrinkFit, we were able to design four accelerators once, reuse logic between accelerators, and take advantage of small and large FPGAs.

Acknowledgment

This work was supported in part by the National Science Foundation (NSF) Expeditions in Computing Award #: CCF-0926148.

References

[1] Xilinx, "Zynq-7000 all programmable SoC," 2013. [Online]. Available: http://xilinx.com/zynq

[2] Intel, "Intel Atom processor E6x5C series," 2013. [Online]. Available: http://www.altera.com/devices/processor/intel/e6xx/proc-e6x5c.html

[3] G. Stitt, F. Vahid, and S. Nematbakhsh, "Energy savings and speedups from partitioning critical software loops to hardware in embedded systems," *TECS*, vol. 3, no. 1, Feb. 2004.

[4] C. Huang and F. Vahid, "Transmuting coprocessors: dynamic loading of FPGA coprocessors," in *DAC*, 2009.

[5] W. Fu and K. Compton, "An execution environment for reconfigurable computing," *FCCM*, pp. 149–158, 2005.

[6] J. Kelm and S. Lumetta, "HybridOS: runtime support for reconfigurable accelerators," *FPGA*, pp. 212–221, 2008.

[7] J. Cong, K. Gururaj, and G. Han, "Synthesis of reconfigurable high-performance multicore systems," *FPGA*, 2009.

[8] ARM, "AMBA Open Specifications," 2013. [Online]. Available: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[9] Richard Herveille, "SoC Interconnection: Wishbone," 2013. [Online]. Available: http://opencores.org/opencores,wishbone

[10] Texas Instruments, "OMAP5430 Block Diagram," p. 291, 2013. [Online]. Available: http://www.ti.com/lit/ug/swpu249x/swpu249x.pdf

[11] M. J. Lyons, G. Wei, and D. Brooks, "Shrink-fit: A framework for flexible accelerator sizing," *IEEE CAL*, vol. PP, no. 99, 2012.

[12] M. J. Lyons *et al.*, "The accelerator store: A shared memory framework for accelerator-based systems," *ACM TACO*, vol. 8, no. 4, pp. 1–22, 2012.

[13] Xilinx, "Vivado Design Suite," 2013. [Online]. Available: http://xilinx.com/vivado

[14] Xilinx, "Spartan product tables," 2013. [Online]. Available: http://www.xilinx.com/publications/matrix/Spartan_Series.pdf

[15] J. Koo, A. Evans, and W. Gross, "3-D brain MRI tissue classification on FPGAs," vol. 18, no. 12, 2009, pp. 2735–2746.

[16] R. Hameed *et al.*, "Understanding Sources of Inefficiency in General-Purpose Chips," in *ISCA 37*, Jun. 2010.

[17] J. Cong *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE TCADICS*, vol. 30, no. 4, pp. 473–491, 2011.

[18] J. R. Wernsing and G. Stitt, "Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *LCTES*, 2010.

[19] M. Dales, "Managing a reconfigurable processor in a general purpose workstation environment," *DATE*, 2003.

[20] Y. Shan *et al.*, "FPMR: MapReduce framework on FPGA," in *FPGA*, 2010.

[21] J. Cong *et al.*, "Charm: a composable heterogeneous accelerator-rich microprocessor," ser. ISLPED, 2012.

[22] Y. Hasegawa *et al.*, "An adaptive cryptographic accelerator for IPsec on dynamically reconfigurable processor," *FPT*, 2006.

[23] S. C. Goldstein *et al.*, "PipeRench: a coprocessor for streaming multimedia acceleration," in *ISCA*, 1999, pp. 28–39.

[24] S. Kestur *et al.*, "SHARC: A streaming model for FPGA accelerators and its application to Saliency," in *DATE 2011*, 2011, pp. 1–6.

[25] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," *FPL*, 2005.

[26] G. Venkatesh *et al.*, "Conservation cores: reducing the energy of mature computations," *ASPLOS*, 2010.

[27] J. Cong *et al.*, "Architecture support for accelerator-rich cmps," ser. DAC '12, 2012, pp. 843–849.