Shrink-Fit: A Framework for Flexible Accelerator Sizing

Michael Lyons, Gu-Yeon Wei, and David Brooks
School of Engineering and Applied Sciences, Harvard University

Abstract—RTL design complexity discouraged adoption of reconfigurable logic in general purpose systems, impeding opportunities for performance and energy improvements [1]. Recent improvements to HLS compilers simplify RTL design and are easing this barrier. A new challenge will emerge: managing reconfigurable resources between multiple applications with custom hardware designs.

In this paper, we propose a method to "shrink-fit" accelerators within widely varying fabric budgets. Shrink-fit automatically shrinks existing accelerator designs within small fabric budgets and grows designs to increase performance when larger budgets are available. Our method takes advantage of current accelerator design techniques and introduces a novel architectural approach based on fine-grained virtualization. We evaluate shrink-fit using a synthesized implementation of an IDCT for decoding JPEGs and show the IDCT accelerator can shrink by a factor of 16x with minimal performance and area overheads. Using shrink-fit, application designers can achieve the benefits of hardware acceleration with single RTL designs on FPGAs large and small.

1 Introduction

Hybrid processors containing general-purpose and FPGA circuits failed to gain popularity in mainstream computing due to the complexity of RTL design. High level synthesis (HLS) compilers are easing this problem, leading to a new reconfigurable logic resource management challenge: general purpose computers run multiple applications at once, and each application will want to program their accelerators on the system's reconfigurable fabric. Fitting all of these accelerators may not be possible. In addition, resource budgets offered by today's FPGAs vary widely, and the range of resources will grow wider with future technology improvements. Manually redesigning accelerators for every possible resource budget is intractable, so the solution must automatically satisfy a wide range of reconfigurable resource budgets with a single design.

In this paper, we propose a novel method to automatically shrink existing accelerator designs within small fabric budgets and grow designs to increase performance when larger budgets are available. We call this method *shrink-fitting*, named after industrial methods to tightly combine parts after manufacturing. Our shrink-fit approach allows the operating system to fit accelerators onto an FPGA *after* implementing RTL. The approach uses virtualization, a technique to reuse hardware for multiple tasks. Shrink-fit differs from previous virtualization techniques which virtualize an accelerator as a single unit [2], [3]. Shrink-fitting works at a finer grain by virtualizing the components that make up each accelerator. This finer grain requires more flexibility and lower setup delays than the DMA transfers used in previous approaches, so we extend the "accelerator store" memory resource proposed in our previous work [4] with new shrink-fit virtualization operations.

2 Resizing Accelerators With Virtualization

Shrink-fitting takes advantage of the module-based design style used to implement most accelerators. Designers typically build hardware by reusing smaller "module" components, programming additional copies of the same module for multiple tasks [5]. A sampling of three popular OpenCores accelerators from different domains [6], [7], [8] confirms that each replicate

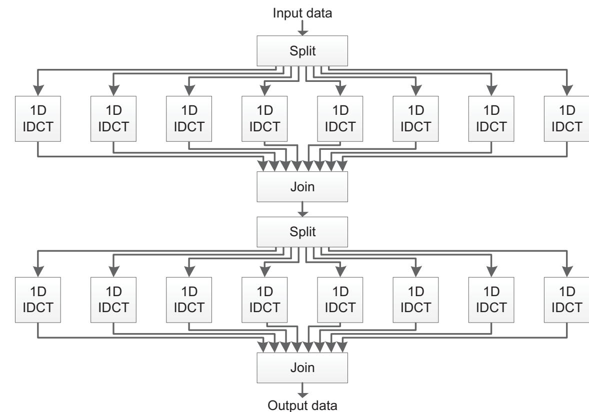

Fig. 1. **2D 8x8 IDCT accelerator data flow**

modules, from 4x to 883x. Using shrink-fit virtualization, a single copy of a module can be used for several tasks, reducing FPGA resource demands.

As an example, we introduce a 2D IDCT accelerator. The inverse discrete cosine transform (IDCT) is a signal processing algorithm used for JPEG image processing, and was used in previous work to demonstrate FPGA acceleration as part of a JPEG decoder [3]. The 2D IDCT accelerator used in this paper consists of sixteen copies of a 1D IDCT module (Figure 1).

Adding virtualization capabilities to a module allows a single instance of that module to replace several. Virtualization allows a module to impersonate other modules, so a virtualized 1D IDCT can act as any of the original sixteen found in the 2D IDCT. A single virtualized module can replace multiple modules by impersonating each in rapid succession.

Keeping with standard virtualization terminology, we refer to the virtualized modules programmed into the FPGA as "physical modules (PMs)" and the original modules they impersonate as "virtual modules (VMs)."

"Contexts" provide the link between PMs and VMs, telling PMs how to act as each VM. A context includes information about where to import and export data for processing, since different VMs may process different data. Contexts may also contain constant and dynamic data. When RTL designers use copies of the same module, each copy may have different constants or variable registers. By loading a context, a PM has all the information necessary to act as a VM, and can load another context, or "context switch," to act as a different VM. Note that PMs can only context switch to another VM of the same kind; an IDCT PM can only act as an IDCT VM and not as a Huffman encoder VM.

A virtualized accelerator can deploy multiple PMs to increase performance, provided the number of PMs does not exceed the number of VMs. The IDCT accelerator contains 16 VMs, therefore one to sixteen PMs can be programmed on the FPGA.

Through virtualization, an operating system can shrink-fit accelerators by increasing or decreasing the number of PMs programmed on the FPGA. The PM count is a trade-off: adding

Posted to the IEEE & CSDL on 5/9/2012
DOI 10.1109/L-CA.2012.7 1556-6056/12/$31.00 © 2012 Published by the IEEE Computer Society

Authorized licensed use limited to: Harvard Library. Downloaded on April 27,2022 at 12:51:46 UTC from IEEE Xplore. Restrictions apply.

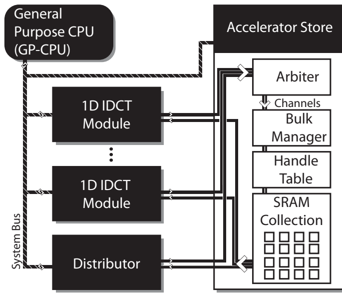

Fig. 2. **Accelerator store system architecture**

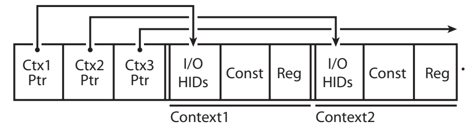

Fig. 3. **Context handle format** The handle contains HIDs for input and output FIFO handles, constants, and dynamic data.

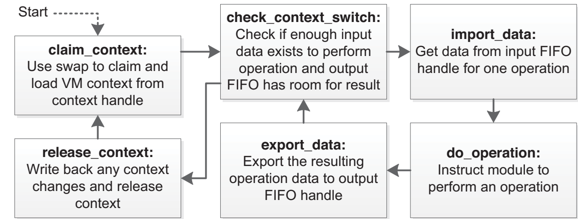

Fig. 4. **State machine for physical module wrapper**

additional PMs improves performance but requires additional reconfigurable logic. Implemented efficiently, this trade-off is linear: increasing the PM count 2x should also improve performance by 2x.

3 Module Virtualization Design

The system's architectural support for module virtualization must perform two functions: storing contexts and transferring data between VMs. We use the *accelerator store (AS)* to perform both of these tasks. The AS is a shared memory resource for accelerators, an alternative to private memories commonly found inside accelerators. Random access (RA) and FIFO memories in the AS are allocated to accelerators as needed at runtime.

We use RA memory in the AS to store contexts, and use AS FIFOs to transmit data between VMs. The AS exposes these memories to all PMs in the system via direct connection (Figure 2), so accesses to AS memory are usually satisfied within a few cycles. This prevents virtualization overheads from significantly affecting performance.

The AS uses *handles* to represent RA and FIFO memories. Each handle corresponds to a single RA or FIFO memory in the same way file handles represent data on a disk. PMs can access these handles by sending *handle requests* to the AS. Every handle request contains a *handle ID (HID)* number, which identifies a specific handle.

Sending data between virtual modules relies on the AS's support for FIFOs. In an unvirtualized accelerator, modules may physically connect their inputs and outputs to transmit data. A VM cannot use direct connections, because the VM is only active for short periods of time. To send data directly between virtual modules, both would need to be active at the same time. Instead, VMs share FIFO handles to transmit data and avoid this scheduling complexity. When the first VM completes an operation, it puts the result in a FIFO, and when the second VM becomes active, it gets the data from the same FIFO. As explained in our previous work [4], the accelerator store requires low area and power overheads, and performance overheads are less than 1%. The AS will not cause FPGA clock slowdowns since the AS supports 1 GHz+ clocks when synthesized by Design Compiler for a popular commercial 40nm technology. Using AS FIFOs, VMs can send data to each other even if they are not active at the same time without significant overheads.

Virtual modules also depend on the AS to store contexts. Each set of PMs copied from the same RTL design, or "module kind," maintains contexts for all of its VMs in a random access *context handle*. For example, the AS stores one context handle that contains contexts for all sixteen 1D IDCT VMs.

Each module kind is free to define the format of its context handle, but we propose a format suitable for most (Figure 3). We store each VM's context in a data record containing I/O handle IDs, constants, and modifiable data. The first words of the context handle form a table listing pointers to the context records; this allows for variable-length contexts. These records are loaded into the PM before performing computation.

Each PM loads contexts from its context handle. The PM first claims a context, ensuring two PMs never act as the same VM at the same time. This prevents PMs from scrambling input and output ordering. To claim a context, PMs swap the first context pointer in the context handle with null (zero). The swap will return a pointer to the context record from which the PM will load a context. The swap operation is necessary to prevent other PMs from loading the same context at the same time. If another PM tries to load the same context, it will load null and know the context is already in use. The PM will then try claiming the next context by swapping with the next context pointer in the context handle.

When a PM finishes acting as a VM, it writes back any changes to the context record. The PM then releases the context by writing the original context pointer back. Other PMs can load that context again.

To add virtualization capabilities to module designs, we introduce the *physical module wrapper (PMW)*. The PMW adds virtualization features to modules by communicating with the AS store. PMWs can be added to non-virtualized modules with little or no modification from the original design. The PMW, implemented in Verilog RTL, implements the state machine shown in Figure 4:

Once each module is virtualized, we identify the flow of data between VMs within the accelerator, as shown for the IDCT in Figure 1. We represent data flow as edges in a directed graph, linking VMs that produce data to those that consume the results. As previously discussed, we implement each link as a FIFO handle. VMs are connected by storing a common FIFO HID in both VMs's contexts.

Finally, the operating system programs as many PMs as will fit within FPGA fabric budgets.

Most accelerators divide or combine workloads for parallel processing. To ensure data ordering is correctly maintained, we first ensure every FIFO handle has only one source VM and one destination VM, preventing multiple PMs from accessing the same FIFO in a nondeterministic order. Second, we implement special *distributor* modules, inspired by StreamIt programming language primitives [9], to deterministically divide or combine data streams. The 2D IDCT uses distributors in two modes: *split* to divide one FIFO into several, and *join* to combine data

Authorized licensed use limited to: Harvard Library. Downloaded on April 27,2022 at 12:51:46 UTC from IEEE Xplore. Restrictions apply.

FPGA	Design	DSPs	Fabric util.	DSP util.
Arria	1D IDCT	No	601 (2.4%)	—
Arria	2D IDCT	No	9616 (38.0%)	—
Arria	1D IDCT	Yes	74 (0.3%)	4 (1.3%)
Arria	2D IDCT	Yes	1184 (4.7%)	64 (20.5%)
Cyclone	1D IDCT	No	901 (6.3%)	—
Cyclone	2D IDCT	No	14416 **(100.01%)**	—

TABLE 1
FPGA resource utilization for synthesized IDCT

from multiple FIFOs into one.

Many PMs may make AS requests at once, and the AS may not be able to handle all simultaneously. We use *channels* to indicate AS bandwidth: the number of channels corresponds to the number of handle requests an AS can accept per cycle. Systems with larger FPGAs can fit more PMs and will require an AS with additional channels. Further details and evaluation of AS configurations can be found in our previous work [4].

4 Shrink-Fit Evaluation

This section describes our experience adding shrink-fit support to an existing 2D 8x8 IDCT. We demonstrate a near-ideal performance-to-area ratio: shrinking an accelerator to half of its original size roughly halves its performance. We show shrink-fit has low area overheads as well, always less than 1% of system area. For the evaluation, we assume the shrink-fit system will utilize a hybrid processor containing general purpose and FPGA cores, such as the Intel E600C processor. The E600C consists of a fabricated "hard" Atom core and a "mid-range" sized Altera Arria II GX FPGA core.

We started by building a 2D 8x8 IDCT without shrink-fit support. 1D IDCT modules were implemented in C and compiled to Verilog RTL using AutoPilot. The resulting 1D IDCT requires 275 cycles to perform one operation. Fully pipelined and parallelized, the 2D 8x8 IDCT is capable of processing one 2D 8x8 IDCT every 275 cycles as well.

To evaluate different classes of FPGA cores, we synthesized the IDCT (in Quartus II 10.1) for the "mid-range" Arria II GX FPGA used by the Intel E600C processor and for the "low-cost" Cyclone IV GX FPGA processor. Table 1 shows the resources consumed on both FPGAs. The low-cost Cyclone cannot fit the 2D IDCT without the help of shrink-fitting. The Arria, however, can fit the full IDCT but requires a significant portion of resources (20.51% of DSPs or 38.01% of ALMs). A hybrid processor running multiple applications with hardware accelerators would require shrink-fitting to run the accelerators simultaneously. In both cases, shrink-fitting allows one 2D IDCT design to utilize a range of FPGA resources with varying workload requirements.

We then implemented system support for module virtualization (the AS, distributor, and PMW) directly in Verilog. Early in the design phase, we decided these architectural components should be hard implementations not programmed into FPGA fabric. These components offer common functionality to all PMs and hard fabrication will reduce logic area costs. The downside to hard fabrication is reduced flexibility, however results presented in Section 4 suggest these virtualization components can be intelligently provisioned in advance with abilities proportional to FPGA resources. This provisioning is primarily achieved by configuring two parameters: channel count and PM-to-AS connections. We sweep these parameters in the following section to evaluate our shrink-fit solution.

First, we investigate how the number of PMs in the system and the choice of channel count affects performance overheads. Second, we investigate how these two parameters affect the

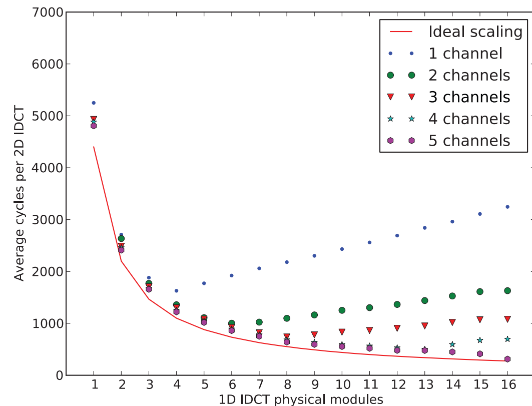

Fig. 5. **Shrink-fit 2D 8x8 IDCT performance sweep:** The red line illustrates the ideal inverse relation between shrink factor and performance (2x shrinking results in half of the performance).

area overhead incurred to support shrink-fit. As a result, a system designer could intelligently weigh performance and area overheads to decide the number of channels to provision in their hybrid processor's AS.

All measurements are obtained from Verilog RTL and use test vectors generated using the IJG JPEG decoder. We run Design Compiler at ultra effort for a popular commercial 40nm process, the same technology generation as mid-range FPGAs found in the E600C hybrid GPCPU+FPGA processor (Arria II). We scale 40nm figures up to 60nm to match the low-cost FPGAs Cyclone IV.

4.1 Shrink-fit performance overhead

In an ideal system, a shrink-fit accelerator's performance is inversely related to the resource reduction. If the 2D IDCT were shrink-fitted by 2x from sixteen PMs to eight PMs, the ideal system's performance would be reduced by half. Figure 5 compares the ideal performance (red line) of a shrink-fit 2D IDCT to our actual performance using ASes provisioned with varying amounts of channels (bandwidth).

Additional channels are necessary to keep pace with increasing numbers of PMs. A single channel AS achieves near-ideal performance for three to four 1D IDCT PMs, and three channels can support near-ideal performance for up to eight PMs. This observation motivates the need for different AS configurations to go with different classes of hybrid GPCPU+FPGAs. Low-cost processors contain less reprogrammable fabric, have room for fewer PMs, and one to three channels is sufficient. The larger number of PMs that fit on a mid-range FPGA necessitate an AS configuration of three to five channels. Using this classification, the relative area overheads of the AS remain low, since smaller processors will require a smaller AS.

In addition to reducing performance, contention can also increase bandwidth requirements. Contention reduces VM throughput, which leads to increased context switching, thus increasing bandwidth requirements even further. Fewer larger modules would result in larger step sizes, whereas more smaller modules would raise context switching overheads and reduce performance. The IDCT's range of one to sixteen modules achieves a good balance by fitting in flexible 6.25% increments.

FIFO handles can be sized as low as 64 bytes with no additional performance impact. The minimum size required by 1D IDCTs is 32 bytes, the minimum size of a 1D IDCT dataset. Under the minimum size configuration, performance

Authorized licensed use limited to: Harvard Library. Downloaded on April 27,2022 at 12:51:46 UTC from IEEE Xplore. Restrictions apply.

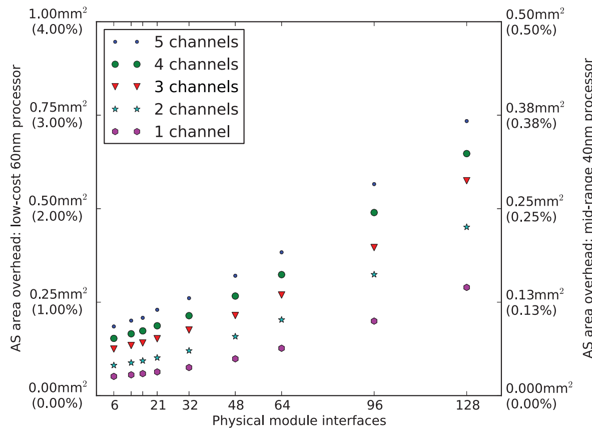

Fig. 6. **Shrink-fit AS area sweep:** The sweep shows absolute and relative AS area overheads for a low-cost $25mm^2$ processor fabricated at 60nm and a mid-range $100mm^2$ processor fabricated at 40nm.

suffers since each PM is required to context switch after every operation. The 2D IDCT requires 36 handles: one for each link in Figure 1 and one context handle for the 1D IDCT module class. An additional 1.125 KB to increase FIFO handles to 64 bytes is a worthwhile price for near-ideal performance.

4.2 Shrink-fit area overhead

If shrink-fit area overheads are low, multiple shrink-fit accelerators may be able to use more FPGA resources than fewer non-fitting accelerators. We compare AS area implemented as a hard core in two processor configurations: a standard $100mm^2$ die size representing a mid-range processor and a $25mm^2$ die for a low cost processor. In accordance with existing FPGA processor resources and the results of our performance evaluation, we assume low-cost hybrid processors will use 1-3 channels and a 60nm technology; mid-range processors will use 3-5 channels and a 40nm technology. Shrink-fit overheads can be kept to less 1% of chip area as discussed below.

For AS configurations ranging from one to five channels of bandwidth and supporting up to 128 PMs, the system needs AS interfaces for 6-21 PMs to support the 2D IDCT. We sweep up to 128 PMs to show the AS can scale to support multiple shrink-fit accelerators running simultaneously.

Area overheads for both low-cost and mid-range processors remains below 1% for all configurations (Figure 6). The low-cost processor can scale to 32 PMs and 3 channels with less than 1% of chip area. FPGA synthesis results of the IDCT suggest support for additional PMs is unnecessary. The mid-range AS with five channels can support up to 128 PMs using less than 0.4% of chip area, which synthesis results suggest is sufficient.

The distributor and physical module wrapper are both too small to significantly affect chip area. Each distributor measures $0.0034mm^2$ at 60nm and $0.0017mm^2$ at 40nm. Each PMW measures $0.0023mm^2$ at 60nm and $0.0011mm^2$ at 40nm.

5 Related Work

Previous work proposed management of reconfigurable resources for hybrid GPCPU+FPGA processors, but used software execution to handle accelerators that did not fit on FPGA fabric instead of shrink-fitting [10].

C-Cores introduced an approach for ASIC designs that automates software-hardware codesign using virtualization-like state management for isolated accelerators [11]. C-Cores targeted isolated accelerators rather than the finer-grained module approach necessary for accelerator resizing.

Prior work has investigated maximizing performance when FPGA resources are not known at design time. To improve the selection of accelerators to program at runtime, several works statically profiled accelerators [12], [13]. These approaches increase the pool of available accelerators by using HLS optimizations to create multiple versions of each accelerator. These approaches depend on significant computation to synthesize and profile hundreds of accelerator variations, memory to store each accelerator variation's bitfile, and accelerators which can be reliably profiled. As accelerators design complexity increases, their behavior will be less predictable and reduce profiling efficacy.

A related approach [14] performs profiling at runtime and periodically reprograms FPGA fabric to use different accelerators as workloads change. This requires dynamic reprogramming, which is known to require minimum delays of tens of milliseconds, and several seconds if reprogramming the entire FPGA [15].

Shrink-fit does not require synthesizing or profiling numerous accelerator variations before or during runtime, and predictable performance leads to less, if any, dynamic reprogramming. But if desired, shrink-fit can complement existing techniques for even greater benefit.

6 Conclusion

We introduced a new method to shrink-fit accelerators within reconfigurable fabric budgets. Our approach, based on fine-grained virtualization, leverages the accelerator store architecture and current module-based RTL design. We introduced a synthesized system containing an IDCT for accelerating JPEG decoding that can be shrunk down to one-sixteenth of its original size, and demonstrated performance and area overheads were low. These early results show shrink-fitting is a feasible path toward resolving future FPGA resource management challenges in mainstream computing.

References

[1] G. Stitt, "Are Field-Programmable Gate Arrays Ready for the Mainstream?" *IEEE Micro*, vol. 31, no. 6, Nov. 2011.
[2] G. Brebner, "A virtual hardware operating system for the Xilinx XC6200," *FPL*, pp. 327–336, 1996.
[3] J. Kelm and S. Lumetta, "HybridOS: runtime support for reconfigurable accelerators," *FPGA*, pp. 212–221, 2008.
[4] 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, Jan. 2012.
[5] D. M. Harris and S. L. Harris, "Digital Design and Computer Architecture," *Morgan Kaufmann*, pp. 167–168, 187, 233, 2007.
[6] 192 AES. opencores.org/project,systemcaes
[7] Reed Solomon. opencores.org/project,reed_solomon_decoder
[8] Elliptic Curve Group. opencores.org/project,ecg
[9] W. Thies and M. Karczmarek, "StreamIt: A language for streaming applications," *Compiler Construction*, Jan. 2002.
[10] M. Dales, "Managing a Reconfigurable Processor in a General Purpose Workstation Environment," *DATE*, vol. 1, Mar. 2003.
[11] S. Kumar *et al.*, "C-core: Using communication cores for high performance network services," *NCA*, Jan. 2005.
[12] J. R. Wernsing and G. Stitt, "Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *LCTES*, Apr. 2010.
[13] J. Cong, K. Gururaj, and G. Han, "Synthesis of reconfigurable high-performance multicore systems," *FPGA*, Feb. 2009.
[14] C. Huang and F. Vahid, "Transmuting coprocessors: dynamic loading of FPGA coprocessors," in *DAC*, Jul. 2009.
[15] V. Rana *et al.*, "Minimization of the reconfiguration latency for the mapping of applications on FPGA-based systems," in *CODES+ISSS*, Oct. 2009.

Authorized licensed use limited to: Harvard Library. Downloaded on April 27,2022 at 12:51:46 UTC from IEEE Xplore. Restrictions apply.