Sharing, Protection and Compatibility for Reconfigurable Fabric with AmorphOS

Ahmed Khawaja, Joshua Landgraf, and Rohith Prakash, ***UT Austin;***
Michael Wei and Eric Schkufza, ***VMware Research Group;***
Christopher J. Rossbach, ***UT Austin and VMware Research Group***

https://www.usenix.org/conference/osdi18/presentation/khawaja

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Sharing, Protection, and Compatibility for Reconfigurable Fabric with AMORPHOS

Ahmed Khawaja[1], Joshua Landgraf[1], Rohith Prakash[1],
Michael Wei[2], Eric Schkufza[2], Christopher J. Rossbach[3]
[1]*The University of Texas at Austin* [2]*VMware Research Group*
[3]*The University of Texas at Austin and VMware Research Group*

Abstract

Cloud providers such as Amazon and Microsoft have begun to support on-demand FPGA acceleration in the cloud, and hardware vendors will support FPGAs in future processors. At the same time, technology advancements such as 3D stacking, through-silicon vias (TSVs), and FinFETs have greatly increased FPGA density. The massive parallelism of current FPGAs can support not only extremely large applications, but multiple applications simultaneously as well.

System support for FPGAs, however, is in its infancy. Unlike software, where resource configurations are limited to simple dimensions of compute, memory, and I/O, FPGAs provide a multi-dimensional sea of resources known as the FPGA *fabric*: logic cells, floating point units, memories, and I/O can all be wired together, leading to spatial constraints on FPGA resources. Current stacks either support only a single application or statically partition the FPGA fabric into fixed-size *slots*. These designs cannot efficiently support diverse workloads: the size of the largest slot places an artificial limit on application size, and oversized slots result in wasted FPGA resources and reduced concurrency.

This paper presents AMORPHOS, which encapsulates user FPGA logic in *morphable tasks*, or **Morphlets**. Morphlets provide isolation and protection across mutually distrustful protection domains, extending the guarantees of software processes. Morphlets can *morph*, dynamically altering their deployed form based on resource requirements and availability. To build Morphlets, developers provide a parameterized hardware design that interfaces with AMORPHOS, along with a *mesh*, which specifies external resource requirements. AMORPHOS explores the parameter space, generating deployable *Morphlets* of varying size and resource requirements. AMORPHOS multiplexes Morphlets on the FPGA in both space and time to maximize FPGA utilization.

We implement AMORPHOS on Amazon F1 [1] and Microsoft Catapult [92]. We show that protected sharing and dynamic scalability support on workloads such as DNN inference and blockchain mining improves aggregate throughput up to 4× and 23× on Catapult and F1 respectively.

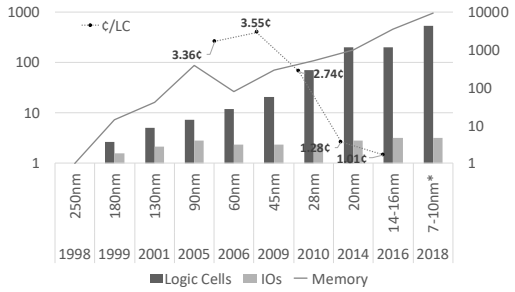

Figure 1: Cost per logic cell and relative density of memory and logic cells over time for FPGAs at each process node. Left and right axes show logic cells and memory density in log-scale relative to 250nm. The dotted line shows the cost per logic cell for the highest density FPGA at that node (in cents) where historical pricing was available [84]. The 14-16nm node introduced FinFETs, which greatly increase performance/W, so that the same application may use fewer logic cells. * Data for 7-10nm projected from [22].

1 Introduction

FPGAs offer compelling hardware acceleration in application domains ranging from databases [28, 59, 74], finance [54, 70], neural networks [115, 104], graph processing [36, 85], communication [57, 107, 27], and networking [53, 92, 27]. Over the last few decades, FPGA compute density has grown dramatically, cost per logic cell has dropped precipitously (Figure 1), and higher-level programming abstractions [60, 19, 32, 20, 65, 90] have emerged to improve programmer productivity. Cloud providers such as Amazon [1] are offering compute resources with FPGAs. However, system software has not

kept up. The body of research effort on FPGA OS support [77, 100, 99, 86, 45, 52, 29] and sharing [30] has yielded no first-class commodity OS support, and on-demand FPGAs from AWS and Microsoft support a single-application model.

Current proposals for FPGA sharing [30, 26, 42, 110, 63] partition a physical FPGA into a small number of fixed-size *slots* and demand-share them across user logic using hardware support for *partial reconfiguration* (PR). PR changes the configuration of FPGA fabric within a slot without perturbing the state of the rest of the FPGA. User logic is pre-compiled to a *bitstream* that targets the pre-defined slots, enabling a system to deploy user logic with low latency. A reserved partition of the fabric, or *shell*, implements library support. Fixed-slot designs have significant drawbacks in practice. Forcing applications to target fixed partitions unnecessarily constrains them: the size of the largest partition places an artificial limit on application size, and oversized partitions result in wasted FPGA resources and reduced concurrency.

We present a design and prototype of protected sharing and cross-platform compatibility for FPGAs called AMORPHOS. AMORPHOS enables applications to scale dynamically in response to load and availability, and enables the system to transparently change mappings between user logic and physical fabric to increase utilization. AMORPHOS avoids fixed-size slots and manages physical fabric in dynamically sized **zones**. Zones are demand-shared across *morphable tasks*, or **Morphlets**. A **Morphlet** is a new abstraction which forms a protection boundary and encapsulates user FPGA logic in a way that enables it to be dynamically scaled and remapped to the physical fabric. Morphlets express scalability dimensions and resource constraints using a **mesh**. A mesh is a map from feasible resource combinations to abstract descriptions of the logic. Meshes act as an intermediate representation (IR) that can be re-targeted at runtime to different hardware allocations, allowing the AMORPHOS scheduler to re-target Morphlets to available FPGA fabric. AMORPHOS caches dynamically generated bitstreams in a shared **registry** to hide the latency of re-targeting. AMORPHOS mediates Morphlet access to OS-managed resources through a **hull**, which hardens and extends a traditional shell design with access control and support for isolation. The hull also forms a canonical interface that enables Morphlets to be portable.

We prototype AMORPHOS on both Amazon F1 and Microsoft Catapult. Measurements show that AMORPHOS's abstractions provide both compatibility and protected sharing while dramatically improving utilization and throughput. We make the following contributions:

- A minimal set of OS-level abstractions and interfaces to enable mutually distrustful FPGA sharing and protected access to OS-managed resources.
- A compatibility layer that enables portability of FPGA code across Amazon F1 and Microsoft Catapult FPGA systems.
- Techniques that transparently transition between scheduling modes based on fixed and variable zones to increase utilization and throughput.
- Evaluation of a prototype showing AMORPHOS sharing support increases fabric utilization and system throughput up to 4× (Catapult) and 23× (F1) relative to fixed-slot sharing and non-sharing designs.

2 Background

Field Programmable Gate Arrays (FPGAs) are circuits that can be configured post-manufacture to implement custom logic. FPGAs may be deployed in a system in several ways:

Discrete. A FPGA can be used on its own without a processor. Network switches, for example [17], can be implemented this way to provide a programmable data plane.

System-on-chip. FPGAs may include one or more hard (in-silicon) processors [35, 16] tightly integrated with the FPGA. Logic in the FPGA can manipulate the processor and vice versa (e.g. FPGA logic may directly write into processor caches or manipulate memory controllers).

Bump-in-the-wire. FPGAs can be placed on an I/O pipeline to "transparently" manipulate data. For example, an FPGA may be integrated into a network card or memory and storage controller to provide line-rate encryption [8].

Co-processor/Offload. FPGAs can be I/O-attached (e.g. via PCIe) to offload compute. An application configures the FPGA to implement a hardware accelerator and sends data and requests to it like a co-processor. Many workloads targeting on-demand cloud FPGAs [1, 79], such as DNNs [83, 116], media transcoding [9], genomics [6], real-time risk modeling [87], and blockchain [105, 49] fall in this category. AMORPHOS *is designed for FPGAs deployed in the co-processor/offload configuration.*

2.1 Software versus Hardware

Writing Hardware. Hardware description languages (HDLs), such as Verilog [106] and VHDL [21], enable developers to configure the various resources on the FPGA *fabric*: interconnect, look-up-tables (LUTs), flip-flops, on-chip memory (block RAM), and "hard resources" (adders, DSPs, memory controllers, etc.). Unlike software, where

resource arrangement is abstracted away by the ISA, hardware gives developers explicit control over arranging and connecting resources in a flexible manner.

Building and deploying hardware. To be deployed on an FPGA, a design must be converted into a *bitstream*, a binary which configures the FPGA fabric. The bitstream is built from the HDL in two stages: First, *synthesis* converts and maps the HDL into a *netlist*, which describes how resources on the FPGA should be connected to implement design logic. Synthesis is similar to software compilation and usually takes on the order of minutes. Second, the *place-and-route* (PAR) step takes the netlist and attempts to route the design on the FPGA fabric. PAR is a constraint-solving problem which can take hours for a complex design. A bitstream takes 10s-100s of milliseconds to be loaded.

Sharing and reconfiguring hardware. Unlike software, which can be context switched by saving and restoring architectural state, context switching FPGA hardware at arbitrary points requires capturing the current state of the logic, as loading a new bitstream will reset that state. While mechanisms do exist, they are not universally supported [47] or are in their early stages [23], and are not supported in all AMORPHOS's target environments. Therefore, time-sharing must either be non-preemptive, or must forcibly revoke access to the FPGA, potentially at the cost of losing application state.

Partial Reconfiguration. Hardware support for *partial reconfiguration* [76] (PR) enables parts of an FPGA to be reconfigured *in situ* without impacting the live configuration or circuit state of other parts of the FPGA fabric. Use of the feature necessitates including partial reconfiguration logic along with the netlist during the place-and-route build phase, but does not otherwise impact the process in a fundamental way: the output is a bitstream that targets *a specific set of physical FPGA resources*. Partial reconfiguration can be faster because partial bitstreams are smaller. Because PR can allow an application to change without impacting the state of other applications, it is an attractive primitive for implementing context switching.

Scaling Hardware. Unlike software, which is scaled by increasing the number of cores or the number of operations executed per instruction (SIMD), hardware can scale by implementing what can be thought of as entirely new specialized instructions or algorithms. This enables FPGAs to provide energy-efficiency and evolvability that are difficult to achieve with fixed-function hardware like GPUs or TPUs [117, 46, 11]. For example, a deep neural network (DNN) can be implemented as thousands of independent 2-bit bitwise processors, rather than consuming the pipeline of a general purpose 64-bit processor.

2.2 FPGA OS and Sharing Support

On-demand FPGAs in the cloud, such as Amazon F1 [1], only enable coarse-grain sharing of a FPGA. F1 provides developers with SDKs for developing, simulating, debugging, and compiling hardware accelerators on-demand. FPGA designs are saved as Amazon FPGA Images (AFIs) and deployed to an F1 instance. The AWS Marketplace functions as a library of pre-built common AFIs. At deployment, an AFI is assigned the fabric of the entire FPGA: there is no support for sharing across protection domains. The lack of fine-grained sharing means that both the cloud provider and the user are locked out of the flexibility of the FPGA: once a user loads an AFI, Amazon must assume that the entire FPGA is being used by that AFI, even though the FPGA may be idle. Other than decommissioning the instance, the user has no way to release FPGA resources back to the cloud provider. As a result, workloads which need to conditionally or occasionally offload compute [97], or which cannot fully utilize the FPGA, may be unable to cost-effectively use cloud FPGAs.

Previous proposals have touched on OS-level concerns such as cross-application sharing [31, 109, 52], hardware abstraction layers [111, 61, 78, 62, 50, 80], and shared runtime support [45, 103, 37], or access from a virtual machine [88]. Theoretical aspects of spatial scheduling on FPGAs [43, 102, 108, 31], task scheduling in heterogeneous CPU-FPGA platforms [25, 102, 108, 44, 18], mechanisms for preemption [73], relocation [55], and context switch [72, 93] are well-explored. Access from an FPGA to OS-managed resources such as virtual memory [33, 15, 114, 77], file systems [100], and system calls [77, 100] has enjoyed the research community's attention as well. However, no first class OS support for FPGAs is present in modern commodity OSes and cloud FPGA platforms support a single application model.

Recent designs for FPGA sharing in datacenters [30, 26, 42, 110, 63] leverage partial reconfiguration to demand share fixed pre-reserved partitions of FPGA fabric among applications with bitstreams pre-compiled to target those partitions. AMORPHOS begins with a design of this form, extends it to enable cross-domain protection, and replaces the fixed slot restriction with elastic resource management to increase utilization and throughput.

3 Goals

AMORPHOS supports demand-sharing of FPGAs by mutually distrustful processes. AMORPHOS multiplexes fabric spatially by default, co-scheduling user logic from different processes, and falling back to time-sharing when

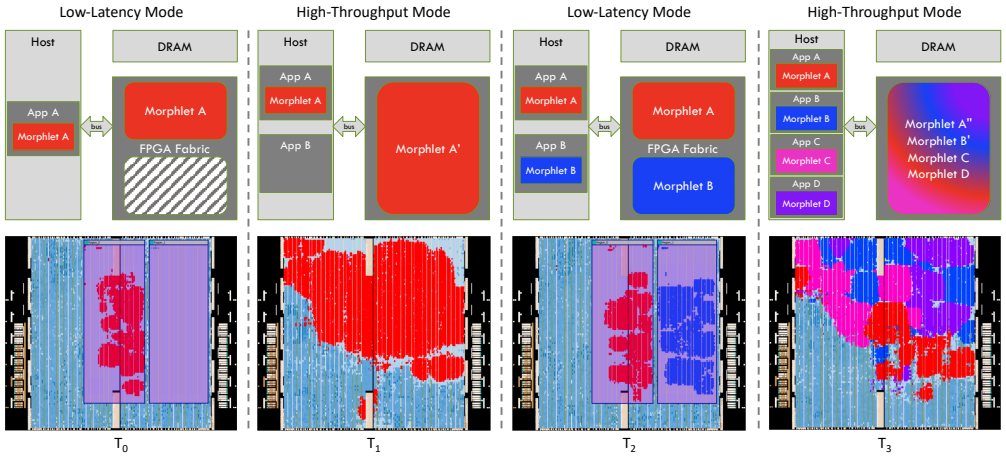

Figure 2: AMORPHOS managing a number of DNNWeaver (see §6) Morphlets. The top row depicts the host and FPGA state while the bottom row shows the corresponding chip layout on Catapult. At T_0, a single DNNWeaver Morphlet is placed on the FPGA. At T_1, AMORPHOS detects underutilization and transitions to high-throughput mode, giving the Morphlet more area. At T_2, another Morphlet is instantiated and AMORPHOS returns to low-latency mode. Finally, at T_3, 2 more DNNWeavers have been scheduled and AMORPHOS transitions to high-throughput mode to fit them all on the FPGA.

space-sharing is infeasible due to resource constraints. A critical design objective for AMORPHOS is avoiding the artificial constraints on inter- and intra-Morphlet scalability induced by a fixed-slot design. AMORPHOS enables individual applications to utilize additional fabric if available, and enables multiple applications to share the fabric to achieve higher aggregate utilization.

3.1 Programming Model

We target a programming model of HDL (hardware description language) over an abstract FPGA fabric. The primary tangible change from current HDL-FPGA programming models is the requirement for the developer to use virtual interfaces for communication with the host and access to on-board resources such as DRAM, network I/O, etc. Collectively, these interfaces form a mediation and compatibility layer called the hull, which encapsulates, hardens, and extends current vendor-specific *shells* [92, 1]

3.2 Isolation

AMORPHOS provides protection guarantees similar to those provided to processes in a software OS. Memory and I/O protection is enforced between Morphlets. Best effort performance isolation is provided based on resource allocation policy and scheduler hints. When FPGA resources are constrained, AMORPHOS dedicates an even share of I/O and memory bandwidth to each Morphlet, enforced by a hardware arbiter. AMORPHOS makes a best effort to allocate fabric fairly under contention by preferring spatial assignments that balance the resources allocated to each application, and time-slicing fairly when spatial sharing is unfeasible (see §4.2 for details). Extending these mechanisms to provide priority-proportional fairness is straightforward, but our prototype currently does not provide flexible software-exposed policies, which we leave as future work. Our current design avoids co-scheduling Morphlets which will interfere with each other through contention on the hardware based on scheduler hints.

AMORPHOS does not provide explicit protection against side channels. Side channels exist and are an active area of research where some mitigations now exist [94]. However, the attack surface for Morphlets is considerably smaller, as Morphlets enjoy exclusive access to all the FPGA hardware resources they use except interfaces to AMORPHOS itself, which are implemented with cross-domain isolation in mind. For example, special care is taken to zero out all signals on a Morphlet's interface if it is not the intended recipient of a transaction, which ensures the Morphlet can not monitor the address/data signals of other Morphlets.

3.3 Dynamic Scalability

A key goal of AMORPHOS is increasing utilization. When only a single application is on the FPGA, it should enjoy exclusive access to all resources it can actually use. When multiple Morphlets contend, if a feasible partitioning of the fabric accommodating them all exists, applications are mapped to shares of the fabric concurrently. If no feasible partitioning exists, the system falls back to time-sharing at coarse granularity. A key challenge to realizing this vision is very high latency (potentially hours or more) of place-and-route (PAR), which maps user-logic to physical fabric. Using partial reconfiguration (PR) to deploy appli-

cations avoids that latency, but constrains applications to fixed slots, giving up elasticity. Avoiding or hiding PAR latency without constraining logic to fixed slots is a primary design goal for Morphlets and the AMORPHOS scheduler. Furthermore, for Morphlets to take advantage of different size partitions, the programming model must provide a way for the developer to express scalability dimensions, valid configurations, and hints to the system to inform the scheduler.

While AMORPHOS's primary sharing strategy is spatial sharing, support for time-sharing is a *de facto* requirement to avoid starvation when the FPGA is contended. Preemptive time-slicing requires mechanisms for capturing, evacuating, and restoring state on the FPGA, and while some applicable mechanisms do exist (e.g. ICAP [47]) they are not universally supported, and state-capture remains an active research area [55, 72, 93, 73]. We opt for a non-preemptive context switch based on extensions to the programming that include a *quiescence interface*.

3.4 Motivating Example

Figure 2 shows a series of scheduling decisions taken by our system in response to applications requesting use of the FPGA. The top row depicts the state of the host and FPGA while the bottom row shows the corresponding chip layout on Catapult V1 FPGAs [92] (Altera Stratix V 5SGSMD5H2F35I3L). At time T_0, process A instantiates a Morphlet on the FPGA. To provide on-demand access at the lowest latency, it initially deploys A on fixed-size zone 1 using partial reconfiguration. At time T_1, AMORPHOS notices the resulting under-utilization and *morphs*. A's mesh is used to select a more performant netlist that uses as much of the FPGA as it can profitably consume, and full reconfiguration is used to give A all the resources not consumed by AMORPHOS itself. At time T_2, process B requests FPGA fabric. To serve that demand quickly, AMORPHOS *morphs* again, reinstating A in zone 1, and mapping B to zone 2. At some future time T_3, which represents the state after potentially many intervening events, four processes have requested FPGA access, and AMORPHOS has *morphed* by selecting netlists from each Morphlet's mesh to produce a single combined bitstream that co-schedules all. Utilization and throughput are improved by 2× compared to a fixed slot design.

4 Design

AMORPHOS introduces a number of new abstractions and interconnected components. A system overview is shown in Figure 3. User logic is encapsulated in **Morphlets**, a **zone** manager tracks allocatable area of physical FPGA fabric, and a scheduler manages the mapping between Morphlets and zones. To enable flexible mapping of Morphlets to zones, Morphlets encapsulate information to enable the scheduler to generate new bitstreams on demand, in the form of **meshes**. To hide the latency of PAR for dynamic re-targeting of Morphlets, the scheduler maintains a **registry** that caches (potentially combined) bitstreams that can be instantiated on a zone with low latency. AMORPHOS mediates Morphlet access to memory and I/O with a compatibility and protection layer called the **hull**.

4.1 Hull

The primary job of the **hull** is to provide cross-domain protection by mediating access to memory and I/O, and to enable compatibility by presenting Morphlets with canonical interfaces to interact with the rest of the platform. The hull coordinates with the scheduler by sending and monitoring quiescence signals (§4.3), disabling connections to zones of the FPGA currently being reprogrammed (§4.2), and connecting and initializing Morphlets after reprogramming is complete. The hull provides memory protection for on-board DRAM using segment-based address translation and manages peripheral I/O devices by implementing shared logic to interface with them, along with simple access mediation logic (e.g. rate-limiting for contended I/O). Finally, the hull exports interfaces to the host OS to configure access control and protection mechanisms, e.g. base and bounds registers for segments.

We expect that future FPGA platforms will provide some of this functionality, address translation in particular, in "hard IP," meaning it will be supported directly in silicon. Our current prototypes are forced to synthesize these functions from FPGA fabric.

4.2 Zones and Scheduling

The **zone** manager allocates physical FPGA fabric to Morphlets. Fabric not consumed by the hull forms a *global* zone, which can be recursively subdivided into smaller reconfigurable zones that can be allocated to different Morphlets. Our Catapult prototype supports two smaller zones within the global zone, each of which can be further subdivided into two. F1 hardware has considerably more resources, and could support a considerably larger number of zones with more levels of subdivision. However, F1 does not expose the partial reconfiguration feature, so our F1 prototype is forced to manage only a single global zone. Zones may be allocated to individual Morphlets or may accommodate multiple Morphlets simultaneously. When it is time to schedule a Morphlet, the job of the zone manager is to find (or create) a free reconfigurable zone matching the Morphlet's default bitstream. If a match is found, the Morphlet can be deployed on that zone di-

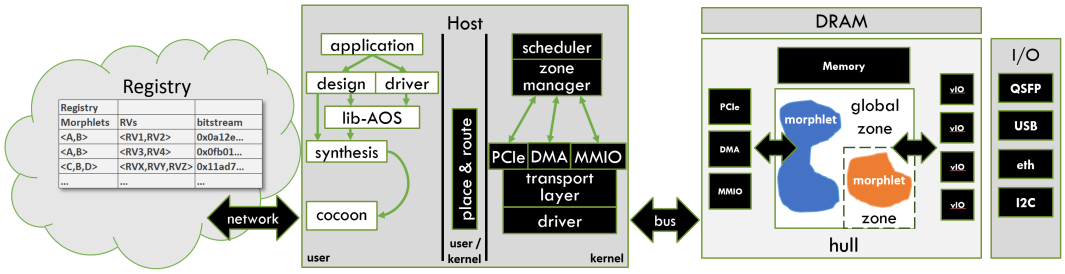

Figure 3: AMORPHOS design overview. FPGA Morphlets (applications) are synthesized by the user and given to AMORPHOS to be converted into bitstreams capable of being placed on the FPGA. The FPGA is split into a hull and multiple zones, in which Morphlets can be scheduled from cocoons. Access to memory and I/O from Morphlets is virtualized by the hull, which implements the logic to interface with the resources directly and to ensure proper access control. On the host side, communication to the Morphlet is virtualized through the lib-AMORPHOS interface.

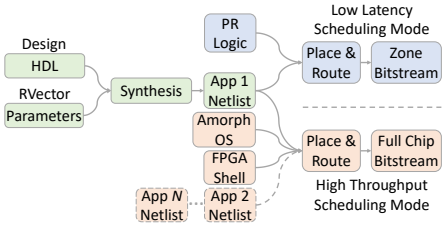

Figure 4: AMORPHOS Morphlet life cycle.

RVector	Cocoon

```
struct {
  bool optLatency;
  size_t minMem;
  size_t optMem;
  size_t memBw;
  size_t PCIeBw;
} RVector;
```

```
struct {
  bitstream default_bitstream;
  map<RVector, netlist> mesh;
} Cocoon;
```

Figure 5: Object model for Cocoons.

rectly. If one is not found, the zone manager must coalesce free (or reclaimed) zones to form a larger one, and inform AMORPHOS that it must re-target the Morphlet along with any other currently-running Morphlets to be deployed on the coalesced zone. In the limit, all Morphlets are deployed together on the global zone, maximizing aggregate utilization and individual application performance.

Zones play a key role in balancing scheduling latency against aggregate throughput because fixed zones and PR is better for fast deployment, while a larger zones with multiple Morphlets is better for utilization and throughput. AMORPHOS's scheduler supports two modes reflecting this tradeoff, *low-latency mode* and *high-throughput mode*, and transitions between those modes transparently based on demand.

In low-latency mode, reconfigurable zones enable Morphlet to be deployed almost instantly through partial reconfiguration with the Morphlet's default bitstream. The Morphlet's default bitstream targets one or more of the smaller zones and includes the partial reconfiguration logic required to enable it to use PR. PR-based scheduling also allows other Morphlets to continue uninterrupted. However, reconfigurable zones incur significant area overhead for the additional PR logic required and increase fragmentation of the FPGA fabric.

When the reconfigurable regions cannot accommodate the Morphlets of all applications concurrently, a *morph* operation occurs. The zone manager coalesces zones to form larger ones, eventually converging to the single global zone, and the scheduler enters high-throughput mode. To do so, it re-targets running Morphlets by running place-and-route to create a bitstream that includes logic for all of them and subsequently maps that bitstream to the target zone. When the global zone is the target, this requires reconfiguring the whole FPGA. However, the global zone can accommodate significantly more Morphlets because PR support fabric is freed, and fragmentation is eliminated by not restricting Morphlets to exclusive partitions of the FPGA. AMORPHOS hides the latency of place-and-route for *morph* operations by caching or pre-computing combined bitstreams targeting the global zone in a Morphlet registry. The registry's entries are bitstreams for "co-Morphlets" representing co-compiled combinations of Morphlets.

AMORPHOS also uses the *morph* operation for single Morphlets when the FPGA fabric is underutilized. Moving a Morphlet to a larger zone or the global zone puts significantly more resources at its disposal. The application can then use these resources to run faster. AMORPHOS targets applications in which Morphlets will likely run for an extended time, so the overhead of moving Morphlets to larger zones is amortized by the gains in aggregate throughput. The ability of a Morphlet to benefit from increasing resource shares is visible to AMORPHOS through the Morphlet's mesh, enabling AMORPHOS to avoid *morphing* when it is not performance profitable to do so.

4.3 Morphlets and Cocoons

While **Morphlets** are analogous to and extend the process abstraction, the AMORPHOS build toolchain produces **Cocoons** from HDL specifications targeting AMORPHOS, which are analogous to an application binary. In addition to the deployable bitstream produced by current FPGA build tools, Cocoons encapsulate abstract information about the Morphlet to enable stages of the build toolchain to be re-invoked dynamically to produce different bitstreams on demand. Dynamic re-targeting enables co-scheduling of multiple Morphlets on a zone or dynamic scaling of the fabric resources allocated to an individual Morphlet.

Figure 5 shows the contents of a Cocoon, and Figure 4 shows how the various stages in the build and deployment process interact with Cocoons to enable dynamic targeting. A cocoon's *default bitstream* targets a default zone on the device and can be deployed using PR. Its **mesh** encapsulates a constrained set of strategies for re-targeting the Morphlet's user logic. Concretely, a mesh is a map of abstract descriptions of the logic, or **netlists**, keyed by **RVectors**. An RVector describes a feasible combination of resources and scheduler hints for the corresponding netlist. The netlist acts as an intermediate representation (IR) which can, potentially in combination with netlists from other Morphlets, be used as input to place-and-route tools to produce new deployable bitstreams. The *default bitstream* is always used when the scheduler is in low-latency mode. When the scheduler is in high-throughput mode it may compare current system state against RVectors in the mesh to select an appropriate netlist. To deploy the dynamically chosen configuration, the scheduler can then produce the required bitstream or look it up in in the Morphlet registry (§4.5) to hide place-and-route latency.

RVectors. A RVector (Resource Vector) describes Morphlet resource constraints and utilization hints that cannot be derived from the netlist in the mesh. Important entries include Boolean valued hints for memory and PCIe usage which simplify connection to AMORPHOS FPGA-side interfaces, as well as optimal and minimal memory footprint and bandwidth estimates. Our experience implementing AMORPHOS is that hints regarding an application's bottleneck resources and access patterns are essential to guide co-scheduling. For example, this allows the hull to be optimized for lower memory access latency with some bandwidth trade-off. Note that low level FPGA-specific resources (e.g. number of LUTs, BRAMs, etc.) can be derived from a netlist and are not included in a RVector.

Quiescence Interface. Evacuating Morphlets from the FPGA is necessary when the enclosing process terminates

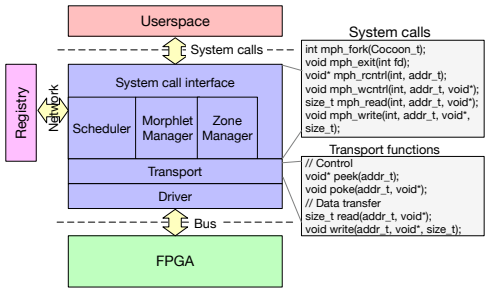

Figure 6: AMORPHOS host stack interfaces between user space and FPGA Morphlets.

or when the scheduler needs to reallocate a zone to another Morphlet. Rather than immediately removing the Morphlet (at risk of losing work) or attempting to capture and save a Morphlet's state (difficult with current hardware [98, 56]), the hull provides a quiescence interface to inform the Morphlet of the impending context switch. The Morphlet is then given an opportunity to enter a stable state and/or save its progress. A Morphlet informs AMORPHOS that it can be safety switched by asserting a *quiescence* signal through the hull. Unresponsive Morphlets are forcibly evacuated after a configurable time-out to avoid DoS. Our current design allows Morphlets to leave data in on-board memory in the absence of memory pressure from incoming Morphlets. Transparent swap in/out of a Morphlet's FPGA DRAM state is a straightforward operation; our current prototypes do not yet support it.

4.4 Host Stack/OS interface

AMORPHOS integrates with the OS in the Catapult stack and acts as a user-mode library for F1. The entire host stack is depicted in Figure 6. AMORPHOS's OS interface exposes system calls to manage Morphlets and enables communication between host processes and Morphlets. The interface provides APIs to load and evacuate Morphlets as well as to read and write data over the transport layer to FPGA-resident Morphlets.

4.5 Morphlet Registry

AMORPHOS dynamically transitions between low-latency and high-throughput scheduling mode, reflecting a fundamental latency/density tradeoff. To hide the latency of dynamic bitstream generation, AMORPHOS maintains a registry, a cache of precomputed bitstreams that contain deployable spatial sharing combinations of multiple Morphlets. For a large number of Morphlets, precomputing bitstreams for all possible combinations is impractical, particularly when combinations include duplicate Morphlets. We argue that a number of factors enable us to reduce the search space to a practical level. First, building

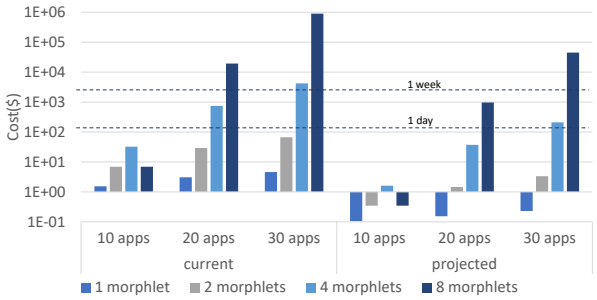

Figure 7: Cost of pre-compiling all possible combinations of Morphlets given varying numbers of deployable Morphlets and varying levels of concurrency. Cost is in dollars and reflects the cost of renting demand infrastructure from Amazon AWS to run the build toolchains. "Current" data are based on measurements with our present toolchain, while projected are scaled to assume a (conservative) 20× improvement in place-and-route performance based on [40, 39, 38].

combined Morphlets can occur in parallel. Second, reducing the latency of place-and-route is an active area, and recent research has produced order of magnitude reductions (20-70×), e.g based on GPUs [40] or other parallel resources [39, 38]. Third, Morphlets can be grouped by popularity or according to hints encoded in RVectors to bound the the number of choices, and sharing densities need not be maximized to achieve multiplicative improvements in throughput and utilization.

Figure 7 shows the cost in dollars for AWS infrastructure to pre-compile all possible combinations of Morphlets for varying numbers of Morphlets and concurrency levels using current tools and using future tools whose performance is projected based on [40, 38, 39]. Compile times are derived from our own benchmark builds. [1] The dotted lines correspond to a day and a week of compute time on 20 VMs. The AWS marketplace, at the time of this writing, offers only 18 FPGA applications [4]. From this pool, all possible co-schedules of 4 Morphlets can be computed in under a day for $100 in computation time. Faster future build tools and careful grouping to reduce the search space can increase utilization further. For example, if co-locatable Morphlets are partitioned in groups of 20, all densities of up to 8 can be precomputed in a handful of days for $1,000. The registry need not eliminate lookups or maximize density to significantly improve utilization.

5 Implementation

We implement AMORPHOS on Amazon F1 FPGA cloud instances[1, 2] and the Microsoft Catapult open research platform [92], available at TACC [5].

[1] Concretely, a single instance of DNNWeaver can be compiled for F1 in 103 minutes. The second and third instances bring that to 118 minutes, while 8 instances can be co-compiled in 157.

Catapult and F1 both support shells to provide three basic forms of platform library support: 1) a bulk host-FPGA data transfer interface, 2) a control interface to manage FPGA applications, and 3) interfaces to on-board DRAM. Catapult and F1 expose these functions with different levels of abstraction. Catapult supports packetized bulk data transfers, a register interface for control signals, and a simple FPGA-side memory read/write interface with independent ports. F1's shell exports AXI4 [3] interfaces to encapsulate these three functional areas. AMORPHOS's interface must encapsulate both Catapult and F1 interfaces, as well as implement address translation for memory protection and I/O access mediation.

The AMORPHOS hull exposes 1) Control Register (CntrlReg) for Morphlet management, 2) Simple PCIe for bulk data transfer, and 3) a AMORPHOS Memory Interface (AMI) supporting 64-byte read/write transactions. Morphlets written to these interfaces are portable across Catapult and F1. AMORPHOS transparently manipulates address bits so Morphlets believe they have full control of memory. OS-programmable BARs (base-address registers) are used to control and protect what regions of memory are accessible to different Morphlets. In addition to memory protection, AMORPHOS provides each Morphlet with a virtual address space and abstracts away the 1-to-1 port-to-channel mappings imposed by F1 and Catapult shells. Virtual address spaces are striped across all memory channels. The number of co-resident Morphlets, memory access ports per Morphlet, and number of memory channels are parameters for the hull. Furthermore, the hull is modular and incurs no overhead for unused interfaces on a target FPGA platform.

Logic structures, such as FIFOs, are fundamental building blocks for FPGA application designers. AMORPHOS provides an FPGA-agnostic wrapper, HullFIFO, that exposes a high level interface to efficiently map to low-level primitives on both F1 and Catapult.

5.1 Catapult

Catapult divides FPGA fabric into a *shell* and user-logic called a *role*. The Catapult shell interface to memory is two 64-byte wide read/write ports over *disjoint* address spaces. AMORPHOS adopts the 64-byte transaction size but virtualizes the interfaces for multiple co-resident Morphlets using segment-based address translation and buffering to support application-level read-modify-write operations.

To enable AMORPHOS to use partial reconfiguration to manage zones we add a PR controller and PR wrapper. The PR controller streams in PR bitstream data from the PCIe bus and transfers it to a PR IP module (vendor-provided Intellectual Property logic block) which uses it

to reconfigure the zone fabric. I/O to each Morphlet is routed through the PR wrapper, which handles driving the Morphlet inputs and disconnecting the Morphlet outputs during PR. This safeguards the application and prevents spurious I/O during the programming process.

5.2 F1

F1 features a *shell* and a user application as *Custom Logic (CL)*. F1 features twice as many memory channels as Catapult and requires the CL to instantiate additional memory controllers if more than one memory channel is needed. AMORPHOS handles instantiating the memory controllers and is parameterized to scale itself to handle additional memory channels. The F1 shell features many different PCIe interfaces, some for DMA type transfers between the host and some lower throughput for management/-control of the CL. PCIe and Memory on F1 are exposed over AXI4 interfaces, which are more complex than the interfaces on Catapult. This complexity is abstracted away from the Morphlet and implemented in our hull. The hull sits on top of an unmodified F1 shell.

5.3 Multiplexing AMORPHOS Interfaces

Large numbers of concurrent Morphlets can stress AMORPHOS's internal FPGA-side subsystems. Each Morphlet requires the same set of interfaces (CntrlReg, Memory, and PCIe). Routing and connections to all of them is complicated by the fact that I/O pads for each can be (and are on F1/Catapult hardware) on different edges of the physical FPGA, which stresses place and route tools by complicating the routing problem and increasing congestion. Designing AMORPHOS's multiplexing logic to anticipate scale can mitigate some, but not all, of the problem. An initial design used multiple flat multiplexers to distribute interface signals to each Morphlet, but we found that, despite plenty of available fabric, they could not scale past 4 concurrent Morphlets in most cases.

Our current design implements a pipelined binary tree to route the CntrlReg signals. The tree-distribution network enables us to add pipeline stages, making it easier to meet timing while reducing the fanout of large data buses. The benefit is a substantial improvement to the scale at which AMORPHOS can route interfaces to concurrent Morphlets. The trade-off is minimal additional latency: 1 additional cycle for each layer, easily tolerable for CntrlReg, which is a low-bandwidth control interface.

Our current implementation takes a different approach with memory. Rather than scale the memory subsystem to provide N Morphlets with access to M memory channels for an arbitrary number of Morphlets, AMORPHOS uses flat multiplexing with up to 8 Morphlets and statically partitions the memory channels across *groups* of Morphlets at sharing densities above 8. This policy enables us to use a single-level of multiplexing and provide access to all channels for all Morphlets at lower densities but avoids the complexity and latency of an additional tree network at high densities. The tradeoff is that Morphlets are restricted to using a subset of DRAM channels, which does not alter the capacity of their memory share but does reduce the bandwidth available to them. Memory systems perform better when they manage fewer access streams (assuming sequential access) because back-to-back operations from a single stream enable optimizations that are not feasible between operations from different streams. The design decision enables much higher densities as it improves routability: a group of Morphlets only need to route to a subset of the memory channels. Our experience is that memory bandwidth contention determines the upper bound on scalability for Morphlets which share DRAM. Contention occurs at lower levels of concurrency than the levels that require strict group-based DRAM channel partitioning, so optimizing DRAM access for high sharing density is unlikely to provide substantial benefits.

5.4 Host Stack

AMORPHOS provides a host stack which interfaces with userspace applications, implemented as an OS extension in our Catapult prototype, and as a user-mode library for F1. The host stack comprises a system call interface, FPGA Morphlet manager and scheduler, zone manager, and transport layer that encapsulates the control and bulk transfer interfaces described above (§5). The interface and stack structure are illustrated in Figure 6. Control signals and reading/writing data are passed through the syscall interface to the transport layer. Morphlet allocation, scheduling hints, and tear down are redirected to the Morphlet scheduler and zone manager.

The host system call interface for Catapult is implemented as a service which supports the transport layer by wrapping the Catapult driver and library stack. The service associates Morphlets with file descriptors, exporting read and write operations on them, and communicates with the scheduler to monitor the active state of executing Morphlets or request quiescence.

6 Evaluation

AMORPHOS runs on both a Mt Granite FPGA board in the Catapult V1 cloud platform [92], containing an Altera Stratix V GS running at 125 MHz with two 4 GB DDR3 channels, and an Amazon F1 cloud instance [1], using a Xilinx UltraScale+ VU9P running at 125 MHz with four 16 GB DDR4 channels. Both platforms are connected over a PCIe bus and support build tools we adapt to build AMORPHOS and our benchmarks, summarized in Table 1.

Program	Description
DNNWeaver	Convolutional neural network
MemDrive	Memory streaming
Bitcoin	Bitcoin hashing accelerator
DFADD	Double-precision addition
DFMUL	Double-precision multiplication
DFSIN	Double-precision Sine function
MIPS	Simplified MIPS processor
ADPCM	Adaptive differential pulse codec
GSM	Linear predictive coding analysis
JPEG	JPEG image decompression
MOTION	Motion vector decoding
AES	Advanced encryption standard
BLOWFISH	Data encryption standard
SHA	Secure hash algorithm

Table 1: Benchmarks used to evaluate AMORPHOS

Catapult Benchmark	Logic Cells	Registers	BRAM Bits
DNNWeaver	39,994	108,640	387,840
MemDrive	2,449	1,488	570,496
Bitcoin	42,171	60,257	21,408
blowfish	20,581	24,082	810,850
gsm	20,910	24,716	5,552
mips	17,672	19,981	657,574
dfmul	17,759	20,586	0
aes	23,900	28,366	689,630
motion	25,178	26,734	687,366
dfadd	18,043	21,014	662,694
sha	17,772	21,380	788,806
adpcm	22,840	29,837	663,654
jpeg	42,243	40,327	1,116,312
dfsin	26,742	32,572	663,805
F1 Benchmark	**LUTs**	**Flip Flops**	**BRAM Bits**
DNNWeaver	4,924	4,773	339,968
MemDrive	1,136	930	0
Bitcoin	40,106	46,191	0

Table 2: FPGA resource utilization by Morphlet type broken down by resource type as reported by each platform.

Benchmarks. We evaluate benchmarks that cover three important categories for FPGA applications, defined by whether they are *memory-bound*, *compute-bound*, or *dynamic resource bound*. Morphlets are *compute-bound* when low-level FPGA resources such as LUTs, BRAMs, etc. are limited. Morphlets are *memory-bound* when off-chip memory bandwidth or latency constrains their performance. Morphlets are *dynamic resource bound* when they can be mapped to the fabric in ways that represent different points along their roofline model [82], meaning they can be *memory-* or *compute-bound*. Our Bitcoin Morphlet (based on [12]) is *compute-bound*. It is parameterized to replicate hashing units and can scale to consume most of the on-board FPGA fabric. Additional instances of functional units increase logic utilization limiting the maximum size/throughput of the Morphlet. Applications that are *memory-bound* usually have a low compute-to-memory ratio and directly benefit from additional off-chip memory bandwidth. Streaming applications (e.g. in database [75] or search [112]) access large amounts of data, often discarding much of it or doing minimal compute per datum. To represent a range of such applications, we wrote a custom Morphlet called MemDrive (MemD) that can be configured on the host side post-synthesis to generate different memory traffic patterns and read/write ratios, along with operations such as fills, reductions, and ECC checks.

Many applications can be configured to take advantage of either additional logic or additional memory bandwidth, corresponding to different points along their roofline model. To represent this class, we evaluate DNNWeaver [96], an open source DNN design framework that can be used to synthesize models from a description of a specific network topology. The user controls the number of functional units and data buffer sizes, translating to variable demand for on- and off-chip resources. We instantiate DNNWeaver with an 8-layer LeNet [71] topology.

To increase benchmark diversity, we include a number of benchmarks that perform many useful non-trivial functions that do not fully utilize the fabric or memory bandwidth. We use the LegUp [7] high level synthesis (HLS) environment to generate 11 Morphlets (a subset of CHStone[48]). LegUp applications use memory by composing it from BRAMs when needed, rather than off-chip DRAM, so they do not contend for DRAM bandwidth. However, as many FPGA applications (DNNs included) are optimized to ensure their working set fits in on-chip BRAMs to minimize off-chip memory access, we believe they are representative.

Metrics. We report resource utilization and performance measured by throughput. The build tools for each platform break down resource utilization into logic, registers/flip-flops, and BlockRAMs. Morphlets are instrumented with cycle counters to measure the runtime on the FPGA when each is running. End-to-end execution time is measured from the host side. Performance for MemDrive is reported as memory throughput (bytes/cycle). Bitcoin performance is reported as normalized hash throughput with the baseline being a fully unrolled and pipelined instance of the application (to the maximum the open source code permitted), producing a full block hash per cycle. DNNWeaver performance is reported as normalized throughput, where the baseline is the number of cycles required for input data to run through all network layers and complete inference.

We evaluate AMORPHOS with 14 different benchmarks, listed in Table 1. The logic, register, and memory utilization of these benchmarks is listed for both Catapult and (partially for) F1 in Table 2.

Table 3 shows increases in utilization and system

Catapult Configuration	# ALMs	Utilization	Sys. Throughput
1 Bitcoin	63,973	1.00x	1.00x
2 Bitcoin	93,908	1.47x	2.00x
4 Bitcoin	141,139	2.21x	4.00x
1 DNNWeaver	92,619	1.45x	1.00x
2 DNNWeaver	134,972	2.11x	2.00x
4 DNNWeaver	154,956	2.42x	3.31x
1 DNN, 1 MemD	92,135	1.44x	1.41x
2 DNN, 2 MemD	148,249	2.32x	0.80x
1 DNN, 1 BTC	112,010	1.75x	2.00x
2 DNN, 2 BTC	140,635	2.20x	3.68x
1 DNN, 1 BTC, 2 MemD	96,994	1.52x	1.86x
2 BTC, 2 MemD	95,936	1.50x	2.77x
F1 Configuration	**# LUTs**	**Utilization**	**Sys. Throughput**
1 MemD	68,885	1.00x	1.00x
2 MemD	89,161	1.29x	1.67x
4 MemD	100,773	1.46x	1.37x
8 MemD	127,530	1.85x	0.78x
1 Bitcoin	104,851	1.52x	1.00x
4 Bitcoin	229,482	3.33x	4.00x
8 Bitcoin	484,879	7.03x	8.00x
1 DNNWeaver	90,118	1.31x	1.00x
4 DNNWeaver	129,925	1.89x	3.94x
8 DNNWeaver	187,839	2.73x	7.80x
16 DNNWeaver	294,290	4.28x	14.80x
32 DNNWeaver	397,580	5.78x	23.22x

Table 3: Morphlet configurations run in AMORPHOS with corresponding ALM/LUT (logic) usage, relative system utilization improvement, and relative system throughput.

throughput that are made possible by co-scheduling Morphlets using AMORPHOS. Utilization is measured as ALM (adaptive logic module) or LUT (lookup table) usage relative to the smallest configuration on each platform, 1 Bitcoin for Catapult and 1 MemDrive for F1. System throughput is reported as the sum of each Morphlet's normalized throughput, relative to a single instance of that Morphlet. In only two cases does co-scheduling Morphlets result in reduced system throughput, both of which involve multiple MemDrive Morphlets, which interfere significantly with other memory-dependant Morphlets. In the best cases, co-scheduling Morphlets results in 7.03x increased utilization and 23.22x increased throughput.

6.1 CHStone

We evaluate CHStone benchmarks to illustrate generality and to demonstrate that useful accelerators can be co-scheduled at high density with AMORPHOS to increase throughput. We find that the upper bound on density for all is determined by AMORPHOS's ability to route control interfaces to them, which translates to an upper bound of 8 on our Catapult prototype. Because the LegUp compiler implements memory with BRAM, rather than by connecting to on-board DRAM, the CHStone workloads only shared resource is the CntrlReg interface. Absent any source of contention, they scale linearly to the upper bound when co-scheduled as Morphlets on AMORPHOS.

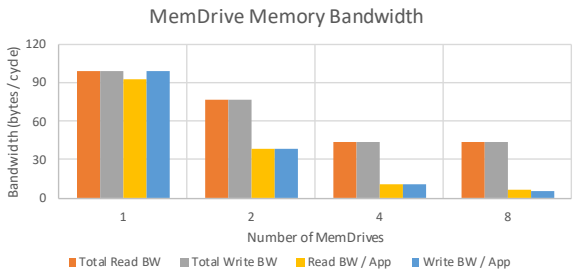

Figure 8: Total and per-MemDrive memory bandwidth for different numbers of Morphlets running in AMORPHOS on Catapult.

We do not report measurements on our F1 prototype as they illustrate the same phenomenon.

6.2 MemDrive

We study contention between *memory-bound* Morphlets using MemDrive, which stresses memory bandwidth. AMORPHOS's 64-byte read/write interface maps well to Catapult, but does not support burst transactions (one transaction returning multiple data payloads), which is necessary to achieve high read throughput on F1's AXI interface to memory. While we were able to achieve peak write-bandwidth on F1 and observe contention due to multiple applications running concurrently, we were unable to saturate read-bandwidth. In future work, our intention is to introduce burst detection and dynamically coalesce memory requests.

Catapult's memory system has a theoretical bandwidth of of 128 bytes/cycle. Experiments on our Catapult prototype show that the total achievable memory bandwidth is roughly 100 bytes/cycle for writes and 90 bytes/cycle for reads. We ran MemDrive in AMORPHOS and directly on the Catapult system to confirm that our virtualization layer incurs no bandwidth loss. Figure 8 shows the per-Morphlet and total system read/write bandwidth when running 1–8 Morphlets of MemDrive in AMORPHOS. Total system bandwidth decreases as the number of co-resident Morphlets rises from 1 up to 4, and saturates from 4 to 8. On F1, we observed similar contention when scaling from 1 to 8 MemDrive Morphlets. The RVector of each Morphlet provides hints to AMORPHOS's on-FPGA memory scheduler, enabling it to manage contention fairly, and improve effective memory bandwidth (e.g. by batching memory requests) or minimize latency for Morphlets that are latency sensitive. MemDrive is not latency sensitive, but its ability to saturate memory has implications for the memory scheduler, which must take care to ensure that latency sensitive Morphlets such as DNNWeaver are not impacted by that saturation.

6.3 DNNWeaver

Table 3 shows how DNNWeaver scales when instantiating multiple Morphlets. We see that aggregate throughput increases with more Morphlets, but contention causes the deviation from perfect linear-scaling to increase with increasing co-resident Morphlets. Both Catapult and F1 theoretically have enough bandwidth to support up to 4 and 32 DNNWeaver Morphlets, respectively. Contention for the memory system manifests as an increase in memory latency for DNNWeaver. We further show this contention in Table 3 by pairing DNNWeaver with MemDrive. Since DNNWeaver performance can suffer if it is paired with a *memory-bound* Morphlet, encoding a Morphlet's sensitivity to memory bandwidth/latency in the RVector is useful for the AMORPHOS scheduler.

6.4 Bitcoin

Up to 4 and 8 Bitcoin Morphlets can be co-resident on Catapult and F1 respectively. Table 3 shows that scaling is linear as Bitcoin only contends for on-chip resources, which are assigned during bitstream generation. The RVector for a Bitcoin-type application specifies that there is no runtime overhead except fabric resources. This would enable AMORPHOS to intelligently co-schedule Bitcoin with other Morphlets that make heavy use of memory but require much less fabric resources, such as MemDrive. *Compute-bound* Morphlets would be great for utilizing unused fabric as they can scale with available logic resources without hurting the performance of *memory-bound* Morphlets. We show this in Table 3 by pairing Bitcoin with DNNWeaver and MemDrive.

6.5 Density Limits

To determine the limits on sharing density we co-schedule as many concurrent Morphlets as possible, manually manipulating the build process where necessary to achieve higher density. While AMORPHOS can achieve high levels of concurrency this way, practically attainable and performance profitable levels are lower. High density co-scheduling of Morphlets stresses build tools because interfaces must be routed to each Morphlet. Avoiding routing congestion at higher densities require manipulation of the build tools. For example, configuring the build tools to focus on congestion rather than logic minimization spreads out the design and replicates logic, increasing area overheads. Routing is heuristic, so successfully meeting timing can depend on trying multiple random seeds. Such interventions are impractical to automate in an OS scheduler, and a production deployment of AMORPHOS would necessarily tolerate sharing densities below the maximum possible.

Morphlet	MaxPerf	MaxTools	Max
DNNWeaver	32	8	32
Bitcoin	8	4	8
MemDrive	2	8	32

Table 4: Limits on AMORPHOS F1 sharing density for DNNWeaver, MemDrive, and Bitcoin. The MaxPerf column indicates the level of Morphlet concurrency at which throughput is maximal. The MaxTools column indicates the maximum concurrency achievable without manual intervention in the build process. The Max column indicates the maximum level we attained with manual intervention in the build process. For example, DNNWeaver's maximal performance is achieved at 32 Morphlets, which is only achievable with manual effort; the build tool chain defaults achieve a maximum density of 8.

Limits on sharing density differ across workloads. Table 4 shows maximum densities on F1 when the upper bound is determined by best throughput, build transparency, or physical limits of the FPGA.

6.6 End-to-End Performance

To compare AMORPHOS against other FPGA sharing designs, we measure the time required to run 1-8 Bitcoin instances on Catapult using AMORPHOS in high-throughput mode, several slot-based approaches, and a no-sharing baseline. The performance of slot-based approaches is emulated by running AMORPHOS in low-latency mode, which uses PR to switch between zones of equal size. The performance of not sharing is emulated by running AMORPHOS with a single Morphlet. Since programming the whole FPGA using the Catapult tools takes a significant portion of time, we also emulate optimal full FPGA reconfiguration by adding a delay of 200ms, which is comparable to programming the whole FPGA via PR. The overhead of using AMORPHOS to emulate these approaches is negligible compared to application runtime, so we expect our results to be accurate for all approaches.

In high-throughput mode, AMORPHOS can fit 4 full-sized Bitcoin Morphlets on the FPGA: we assume that the registry is pre-populated with the required bitstream (see §4.5). When using fixed slots, only two Bitcoin instances can be co-resident. Since slots may not always be able to fit the largest version of Bitcoin, we emulate three different sizes of slots, which we refer to as small, medium, and large. The small slot can fit a quarter-speed variant of Bitcoin, the medium slot can fit a half-speed variant, and the large slot can fit the full-speed variant. In the no-sharing approach, a single full-sized Bitcoin Morphlet is instantiated.

Figure 9 reports the full system runtime of each approach. When running only a single Bitcoin Morphlet, AMORPHOS is comparable to both the no sharing and two large slot approaches. The smaller slot-based approaches limit the size of the application and already perform worse than AMORPHOS. With two Bitcoin Morphlets, only

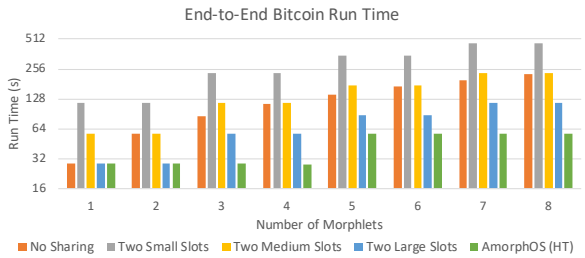

Figure 9: End-to-end runtime of Bitcoin executing under several different sharing schemes. Runtime is plotted logarithmically with lower runtimes being better.

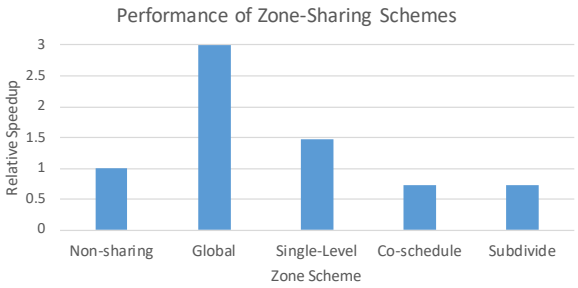

Figure 10: End-to-end performance of various zone-sharing schemes when executing 3 Bitcoin Morphlets.

AMORPHOS and two large slots are comparable. Finally, with 3 or more Bitcoin Morphlets, AMORPHOS is consistently able to attain higher logic densities and therefore better throughput than all other competing approaches. While AMORPHOS cannot always run in high-throughput mode as shown here, we expect AMORPHOS to maintain the same comparative advantage in the long run as it will only have to operate in low-latency mode until a high-throughput bitstream has been generated.

6.7 Hierarchical Zone Management

AMORPHOS can manage a zone in three ways. It can allocate the zone for exclusive use by a single Morphlet, co-schedule multiple Morphlets on it, or recursively subdivide it into two smaller zones. Subdividing top-level zones may be attractive if Morphlets do not fully utilize those zones or if Morphlet response time is more important than end-to-end run time. This flexibility gives rise to a policy space that trades off between density, performance, and registry overhead.

To characterize these trade-offs, we run three Bitcoin Morphlets on our Catapult prototype, in which AMORPHOS uses a single global zone or two top-level reconfigurable zones, each of which may be subdivided in two. We measure end-to-end execution time to completion for all three Morphlets, using a lower-bound baseline that does not share (**non-sharing**) and an upper-bound baseline that co-schedules all Morphlets on the global zone (**global**). We evaluate three different policies for managing the top-level zones. The first implements only a single-level of zone partitioning (**single-level**) with no co-scheduling within the zones. The second policy schedules combined Morphlets on zones without subdividing the zones (**co-schedule**). The third policy (**subdivide**) can subdivides the top-level zones. Registry entries for all combined bitstreams are pre-populated. For the **co-schedule** and **subdivide** cases, we morph the second two Morphlets by scaling them down to fit concurrently in a top-level reconfigurable zone, which reduces their throughput by a factor of 4, but allows us to run all three Morphlets in parallel.

Figure 10 shows end-to-end speedup relative to the non-sharing case for all policies. In this scenario, a single level of zone partitioning is the best option when co-scheduling on the global zone is not possible. This enables the first two Morphlets to run concurrently, providing additional concurrency that results in a performance gain relative to the no-sharing strategy. Both strategies for subdividing a top-level reconfigurable zone perform worse than the sequential case, for two reasons. First, performance is reduced by scaling them to fit a subdivided zone. Second, subdividing zones does not make all the underlying resources available to each subdivision. Additional PR logic is required for each, which consumes additional area and reduces routability.

Measurements of overhead for PR on Catapult FPGAs show that it grows linearly with density. Interconnect is the bottleneck resource, with 4% of global interconnect and 2% of global logic consumed by PR per Morphlet. While the 4% interconnect overhead can become a significant fraction of the allocatable fabric, density is primarily limited by fragmentation (we observe an average utilization loss of 16% in our workloads), which makes multiple levels of subdivision unprofitable for all but very small Morphlets on Catapult.

However, multiple levels of subdivision may be useful on F1 FPGAs, where the fraction of resources allocatable through PR zones is larger. F1 does not expose PR, so to predict sharing densities for PR-based subdivision on F1, we extrapolate assuming the same average 16% fragmentation per PR zone and 2% per-Morphlet overhead for PR logic. We assume that all fabric not consumed by AMORPHOS and PR logic can be divided evenly and allocated to Morphlets, but we impose a 90% upper bound on utilization per resource type, which is suggested by Xilinx to be the likely upper bound on UltraScale FPGAs [14]. This over-estimates utilizable fabric: other vendor guidance is more conservative [10] and our measured utilization does not exceed 70% for any workload.

Derived upper bounds on density for F1 hardware show that CLBs (Configurable Logic Blocks, which encapsulate multiple LUTs) are the limiting resource. We predict maximum sharing density with PR to be 16 and 4 for DNNWeaver and Bitcoin, respectively.[2] This suggests that zone subdivision will likely be possible and effective on F1. Our experience building AMORPHOS, however, is that subdividing zones increases the design complexity of hardware components and limits density unnecessarily by increasing fragmentation. In contrast, increasing density by co-scheduling Morphlets on a global zone can provide much higher densities with potentially higher effective deployment latency, but shifts much of the complexity to a software registry.

7 Related Work

FPGA programmability. Improving FPGA programmability is an active area largely characterized by efforts to enable programming with higher level languages, including C/C++ and other imperative languages [60, 19, 34, 69, 13, 18, 51, 68, 67], DSLs [32, 69, 95, 20, 81, 101, 66, 91], and even managed sequential languages such as C# [32] and map-reduce [95]. Progress in this area motivates our work, but is also orthogonal to it.

FPGA access to OS-managed resources. Prior work has explored exposing file systems [100] and the syscall interface [77, 100] to FPGAs. Much of this work has similar goals to our own, but we decided to focus on the exploration of cross-domain sharing and basic memory virtualization. A more mature AMORPHOS could clearly benefit from the rich body of work on memory virtualization for FPGAs [33, 15, 114, 77].

FPGA OSes. Previous work on FPGA OSes has focused on theoretical foundations for spatial sharing [43, 102, 108, 31], mechanisms for task preemption [73], relocation [55], context switch [72, 93], and scheduling of hardware and software tasks [25, 102, 108, 44]. While these explore ideas pertinent to OS primitives, end-to-end OS system-building was not their goal.

Extending current OS abstractions to FPGAs is another area of active research. ReconOS [77] extends a multi-threaded programming model to configurable SoCs that enables programmers to use "hardware threads" to transparently access OS-managed objects in the eCos [41] embedded OS. Hthreads [86] implements a similar hardware thread abstraction. Borph [100, 99] uses a *hardware process* abstraction to encapsulate FPGA logic in a process-like protection domain. Multi-application sharing for FPGAs is explored in [31, 109, 52], but some works restrict the programming model or design space [111], or do not tackle isolation and protection [31]. AMORPHOS differs by proposing new OS abstractions that differ from the existing CPU-oriented programming models.

MURAC [45] is the most closely related work to AMORPHOS. In MURAC, a process' logical address space encompasses all on-device resources that logically "belong to it", enabling the scheduler to support context switch using an ICAP (Internal Configuration Access Port). AMORPHOS takes a similar position on protection domains, but focuses on spatial scheduling and does not rely on hardware support for state capture.

FPGA Virtualization. Systems have been proposed that virtualize FPGAs with regions [88], tasks [89], processing elements [37], IPC-like communication primitives [80], and abstraction layers/overlays over diverse FPGA hardware [62, 50, 24, 61, 103]. Works virtualizing FPGAs in the cloud [30, 1, 79] share many of our core goals and tackle similar challenges. While these platforms use similar primitives to those of AMORPHOS, they typically restrict the programming and/or deployment model and do not support cross-domain sharing of FPGA fabric.

Overlays. FPGA overlays provide a virtualization layer to make a design independent of specific FPGA hardware [24, 113], enabling fast compilation times and low deployment latency [58, 64], at the cost of reduced hardware utilization and throughput. Like AMORPHOS, many overlays support some time-sharing and or spatial sharing. Overlays implement the same virtual architecture on different devices, they form a compatibility layer at the hardware interface. In contrast, AMORPHOS provides compatibility at the application-OS interface. Unlike Morphlets, overlays run on a virtual architecture, introducing overheads that limit utilization and performance.

8 Conclusion

This paper has described AMORPHOS, a design for FPGA protected sharing and compatibility based on abstractions that preserve existing programming models. AMORPHOS modulates between space- and time-sharing policies and isolates logic from different applications, enabling cross-cloud compatibility and dramatically improved throughput and utilization.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd Miguel Castro for their insights and comments. We acknowledge funding from NSF grant CNS-1618563.

[2] We do not predict density for MemDrive as it is bottlenecked by memory bandwidth at low densities.

References

[1] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. (Accessed on 09/27/2018).

[2] Amazon Web Services (AWS) - Cloud Computing Services. https://aws.amazon.com/. (Accessed on 04/30/2018).

[3] AMBA Specifications Arm. https://www.arm.com/products/system-ip/amba-specifications. (Accessed on 05/03/2018).

[4] AWS FPGA Marketplace. https://aws.amazon.com/marketplace/search/results?searchTerms=fpga. (Accessed on 09/14/2018).

[5] Catapult - Texas Advanced Computing Center. https://www.tacc.utexas.edu/systems/catapult/. (Accessed on 9/27/2018).

[6] Edico Genomes DRAGEN Platform. http://edicogenome.com/dragen-bioit-platform/. (Accessed on 5/2/2018).

[7] High-Level Synthesis with LegUp. http://legup.eecg.utoronto.ca/. (Accessed on 10/24/2017).

[8] Innova-2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf. (Accessed on 5/2/2018).

[9] Live Video Encoding Using New AWS F1 Acceleration NGCodec. https://ngcodec.com/news/2017/3/31/live-video-encoding-using-new-aws-f1-acceleration. (Accessed on 09/27/2018).

[10] Measuring Device Performance and Utilization: A Competitive Overview (WP496). https://www.xilinx.com/support/documentation/white_papers/wp496-comp-perf-util.pdf. (Accessed on 09/27/2018).

[11] Microsoft unveils Project Brainwave for real-time AI - Microsoft Research. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/. (Accessed on 10/21/2017).

[12] programism/Open-Source-FPGA-Bitcoin-Miner. https://github.com/programism/Open-Source-FPGA-Bitcoin-Miner. (Accessed on 10/24/2017).

[13] SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html. (Accessed on 09/27/2018).

[14] UltraScale Architecture: Highest Device Utilization, Performance, and Scalability (WP455). https://www.xilinx.com/support/documentation/white_papers/wp455-utilization.pdf. (Accessed on 09/27/2018).

[15] ADLER, M., FLEMING, K. E., PARASHAR, A., PELLAUER, M., AND EMER, J. Leap scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2011), FPGA '11, ACM, pp. 25–28.

[16] ALTERA. Cyclone V SoC Development Board Reference Manual. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/rm_cv_soc_dev_board.pdf. (Accessed on 5/2/2018).

[17] ALTERA. Integrating 100-GbE Switching Solutions on 28nm FPGAs. https://www.altera.com/en_US/pdfs/literature/wp/wp-01127-stxv-100gbe-switching.pdf. (Accessed on 5/2/2018).

[18] ANDERSON, E., AGRON, J., PECK, W., STEVENS, J., BAIJOT, F., SASS, R., AND ANDREWS, D. Enabling a uniform programming model across the software/hardware boundary. fccm'06, 2006.

[19] AUERBACH, J., BACON, D. F., CHENG, P., AND RABBAH, R. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 89–108.

[20] BACHRACH, J., VO, H., RICHARDS, B. C., LEE, Y., WATERMAN, A., AVIZIENIS, R., WAWRZYNEK, J., AND ASANOVIC, K. Chisel: constructing hardware in a scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012* (2012), pp. 1216–1225.

[21] BHASKER, J. *A Vhdl primer*. Prentice-Hall, 1999.

[22] BOHR, M. Moores Law Leadership. `https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/03/Mark-Bohr-2017-Moores-Law.pdf`. (Accessed on 09/27/2018).

[23] BOURGE, A., MULLER, O., AND ROUSSEAU, F. Automatic high-level hardware checkpoint selection for reconfigurable systems. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on* (2015), IEEE, pp. 155–158.

[24] BRANT, A., AND LEMIEUX, G. G. Zuma: An open fpga overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on* (2012), IEEE, pp. 93–96.

[25] BREBNER, G. J. A virtual hardware operating system for the xilinx xc6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers* (London, UK, UK, 1996), FPL '96, Springer-Verlag, pp. 327–336.

[26] BYMA, S., STEFFAN, J. G., BANNAZADEH, H., GARCIA, A. L., AND CHOW, P. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2014), FCCM '14, IEEE Computer Society, pp. 109–116.

[27] BYMA, S., TARAFDAR, N., XU, T., BANNAZADEH, H., LEON-GARCIA, A., AND CHOW, P. Expanding openflow capabilities with virtualized reconfigurable hardware. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 94–97.

[28] CASPER, J., AND OLUKOTUN, K. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays* (New York, NY, USA, 2014), FPGA '14, ACM, pp. 151–160.

[29] CHAI, Z., YU, J., WANG, Z., ZHANG, J., AND ZHOU, H. An embedded fpga operating system optimized for vision computing (abstract only). In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 271–271.

[30] CHEN, F., SHAN, Y., ZHANG, Y., WANG, Y., FRANKE, H., CHANG, X., AND WANG, K. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (New York, NY, USA, 2014), CF '14, ACM, pp. 3:1–3:10.

[31] CHEN, L., MARCONI, T., AND MITRA, T. Online scheduling for multi-core shared reconfigurable fabric. In *Proceedings of the Conference on Design, Automation and Test in Europe* (San Jose, CA, USA, 2012), DATE '12, EDA Consortium, pp. 582–585.

[32] CHUNG, E. S., DAVIS, J. D., AND LEE, J. Linqits: Big data on little clients. In *40th International Symposium on Computer Architecture* (June 2013), ACM.

[33] CHUNG, E. S., HOE, J. C., AND MAI, K. Coram: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2011), FPGA '11, ACM, pp. 97–106.

[34] COUSSY, P., AND MORAWIEC, A. *High-level synthesis: from algorithm to digital circuit*. Springer Science & Business Media, 2008.

[35] CROCKETT, L. H., ELLIOT, R. A., ENDERWITZ, M. A., AND STEWART, R. W. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[36] DAI, G., CHI, Y., WANG, Y., AND YANG, H. Fpgp: Graph processing framework on fpga a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2016), FPGA '16, ACM, pp. 105–110.

[37] DeHon, A., Markovsky, Y., Caspi, E., Chu, M., Huang, R., Perissakis, S., Pozzi, L., Yeh, J., and Wawrzynek, J. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems 30*, 6 (2006), 334–354.

[38] Dhar, S., Adya, S. N., Singhal, L., Iyer, M. A., and Pan, D. Z. Detailed placement for modern fpgas using 2d dynamic programming. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016* (2016), p. 9.

[39] Dhar, S., Iyer, M. A., Adya, S. N., Singhal, L., Rubanov, N., and Pan, D. Z. An effective timing-driven detailed placement algorithm for fpgas. In *Proceedings of the 2017 ACM on International Symposium on Physical Design, ISDP 2017, Portland, OR, USA, March 19-22, 2017* (2017), pp. 151–157.

[40] Dhar, S., and Pan, D. Z. Gdp: Gpu accelerated detailed placement. In *HPEC* (2018).

[41] Domahidi, A., Chu, E., and Boyd, S. Ecos: An socp solver for embedded systems. In *Control Conference (ECC), 2013 European* (2013), IEEE, pp. 3071–3076.

[42] Fahmy, S. A., Vipin, K., and Shreejith, S. Virtualized fpga accelerators for efficient cloud computing. In *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)* (Washington, DC, USA, 2015), CLOUDCOM '15, IEEE Computer Society, pp. 430–435.

[43] Fu, W., and Compton, K. Scheduling intervals for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on* (April 2008), pp. 87–96.

[44] Gonzalez, I., Lopez-Buedo, S., Sutter, G., Sanchez-Roman, D., Gomez-Arribas, F. J., and Aracil, J. Virtualization of reconfigurable coprocessors in hprc systems with multicore architecture. *J. Syst. Archit. 58*, 6-7 (June 2012), 247–256.

[45] Hamilton, B. K., Inggs, M., and So, H. K. H. Scheduling mixed-architecture processes in tightly coupled fpga-cpu reconfigurable computers. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on* (May 2014), pp. 240–240.

[46] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[47] Hansen, S. G., Koch, D., and Torresen, J. High speed partial run-time reconfiguration using enhanced icap hard macro. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* (2011), IEEE, pp. 174–180.

[48] Hara, Y., Tomiyama, H., Honda, S., Takada, H., and Ishii, K. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on* (2008), IEEE, pp. 1192–1195.

[49] Hari Krishnan, R., and Sai Saketh, Y. Cryptocurrency mining–transition to cloud.

[50] Huang, C.-H., and Hsiung, P.-A. Hardware resource virtualization for dynamically partially reconfigurable systems. *IEEE Embed. Syst. Lett. 1*, 1 (May 2009), 19–23.

[51] Inc, S. C. Carte programming environment, 2006.

[52] Ismail, A., and Shannon, L. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2011), FCCM '11, IEEE Computer Society, pp. 170–177.

[53] István, Z., Sidler, D., Alonso, G., and Vukolic, M. Consensus in a box: Inexpensive coordination in hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2016), NSDI'16, USENIX Association, pp. 425–438.

[54] Kaganov, A., Lakhany, A., and Chow, P. Fpga acceleration of multifactor cdo pricing. *ACM Trans. Reconfigurable Technol. Syst. 4*, 2 (May 2011), 20:1–20:17.

[55] KALTE, H., AND PORRMANN, M. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on* (Aug 2005), pp. 223–228.

[56] KALTE, H., AND PORRMANN, M. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on* (2005), IEEE, pp. 223–228.

[57] KAPITZA, R., BEHL, J., CACHIN, C., DISTLER, T., KUHNLE, S., MOHAMMADI, S. V., SCHRÖDER-PREIKSCHAT, W., AND STENGEL, K. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 295–308.

[58] KAPRE, N., AND GRAY, J. Hoplite: Building austere overlay nocs for fpgas. In *FPL* (2015), IEEE, pp. 1–8.

[59] KARA, K., AND ALONSO, G. Fast and robust hashing for database operators. In *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016* (2016), pp. 1–4.

[60] KHRONOS GROUP. *The OpenCL Specification, Version 1.0*, 2009.

[61] KIRCHGESSNER, R., GEORGE, A. D., AND STITT, G. Low-overhead fpga middleware for application portability and productivity. *ACM Trans. Reconfigurable Technol. Syst. 8*, 4 (Sept. 2015), 21:1–21:22.

[62] KIRCHGESSNER, R., STITT, G., GEORGE, A., AND LAM, H. Virtualrc: A virtual fpga platform for applications and tools portability. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2012), FPGA '12, ACM, pp. 205–208.

[63] KNODEL, O., AND SPALLEK, R. G. RC3E: provision and management of reconfigurable hardware accelerators in a cloud environment. *CoRR abs/1508.06843* (2015).

[64] KOCH, D., BECKHOFF, C., AND LEMIEUX, G. G. F. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL* (2013), IEEE, pp. 1–8.

[65] KOEPLINGER, D., DELIMITROU, C., PRABHAKAR, R., KOZYRAKIS, C., ZHANG, Y., AND OLUKOTUN, K. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 115–127.

[66] KOEPLINGER, D., DELIMITROU, C., PRABHAKAR, R., KOZYRAKIS, C., ZHANG, Y., AND OLUKOTUN, K. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 115–127.

[67] KOEPLINGER, D., FELDMAN, M., PRABHAKAR, R., ZHANG, Y., HADJIS, S., FISZEL, R., ZHAO, T., NARDI, L., PEDRAM, A., KOZYRAKIS, C., AND OLUKOTUN, K. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2018), PLDI 2018, ACM, pp. 296–311.

[68] LEBAK, J., KEPNER, J., HOFFMANN, H., AND RUTLEDGE, E. Parallel vsipl++: An open standard software library for high-performance parallel signal processing. *Proceedings of the IEEE 93*, 2 (2005), 313–330.

[69] LEBEDEV, I. A., FLETCHER, C. W., CHENG, S., MARTIN, J., DOUPNIK, A., BURKE, D., LIN, M., AND WAWRZYNEK, J. Exploring many-core design templates for fpgas and asics. *Int. J. Reconfig. Comp. 2012* (2012), 439141:1–439141:15.

[70] LEBER, C., GEIB, B., AND LITZ, H. High frequency trading acceleration using fpgas. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications* (Washington, DC, USA, 2011), FPL '11, IEEE Computer Society, pp. 317–322.

[71] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[72] LEE, T.-Y., HU, C.-C., LAI, L.-W., AND TSAI, C.-C. Hardware context-switch methodology for dynamically partially reconfigurable systems. *J. Inf. Sci. Eng. 26* (2010), 1289–1305.

[73] LEVINSON, L., MANNER, R., SESSLER, M., AND SIMMLER, H. Preemptive multitasking on fpgas. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on* (2000), pp. 301–302.

[74] LI, S., LIM, H., LEE, V. W., AHN, J. H., KALIA, A., KAMINSKY, M., ANDERSEN, D. G., SEONGIL, O., LEE, S., AND DUBEY, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 476–488.

[75] LIN, E. C., AND RUTENBAR, R. A. A multi-fpga 10x-real-time high-speed search engine for a 5000-word vocabulary speech recognizer. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays* (2009), ACM, pp. 83–92.

[76] LIU, M., KUEHN, W., LU, Z., AND JANTSCH, A. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on* (2009), IEEE, pp. 498–502.

[77] LÜBBERS, E., AND PLATZNER, M. Reconos: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst. 9*, 1 (Oct. 2009), 8:1–8:33.

[78] LYSECKY, R., MILLER, K., VAHID, F., AND VISSERS, K. Firm-core virtual fpga for just-in-time fpga compilation (abstract only). In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays* (New York, NY, USA, 2005), FPGA '05, ACM, pp. 271–271.

[79] MICROSOFT. Microsoft azure goes back to rack servers with project olympus, 2017.

[80] MISHRA, M., CALLAHAN, T. J., CHELCEA, T., VENKATARAMANI, G., GOLDSTEIN, S. C., AND BUDIU, M. Tartan: Evaluating spatial computation for whole program execution. *SIGOPS Oper. Syst. Rev. 40*, 5 (Oct. 2006), 163–174.

[81] MOORE, N., CONTI, A., LEESER, M., CORDES, B., AND KING, L. S. An extensible framework for application portability between reconfigurable supercomputing architectures, 2007.

[82] MURALIDHARAN, S., O'BRIEN, K., AND LALANNE, C. A semi-automated tool flow for roofline anaylsis of opencl kernels on accelerators. In *First International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'15)* (2015).

[83] NURVITADHI, E., VENKATESH, G., SIM, J., MARR, D., HUANG, R., HOCK, J. O. G., LIEW, Y. T., SRIVATSAN, K., MOSS, D., SUBHASCHANDRA, S., ET AL. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *FPGA* (2017), pp. 5–14.

[84] OCTOPART. Octopart historical pricing, 2017.

[85] OGUNTEBI, T., AND OLUKOTUN, K. Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2016), FPGA '16, ACM, pp. 111–117.

[86] PECK, W., ANDERSON, E. K., AGRON, J., STEVENS, J., BAIJOT, F., AND ANDREWS, D. L. Hthreads: A computational model for reconfigurable devices. In *FPL* (2006), IEEE, pp. 1–4.

[87] PELLERIN, D. Accelerated Computing on AWS. `http://asapconference.org/slides/amazon.pdf`, July 2017. (Accessed on 5/2/2018).

[88] PHAM, K. D., JAIN, A. K., CUI, J., FAHMY, S. A., AND MASKELL, D. L. Microkernel hypervisor for a hybrid arm-fpga platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on* (June 2013), pp. 219–226.

[89] PLESSL, C., AND PLATZNER, M. Zippy-a coarse-grained reconfigurable array with support for hardware virtualization. In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on* (2005), IEEE, pp. 213–218.

[90] PRABHAKAR, R., KOEPLINGER, D., BROWN, K. J., LEE, H., DE SA, C., KOZYRAKIS, C., AND OLUKOTUN, K. Generating configurable hardware from parallel patterns. *SIGOPS Oper. Syst. Rev. 50*, 2 (Mar. 2016), 651–665.

[91] PRABHAKAR, R., ZHANG, Y., KOEPLINGER, D., FELDMAN, M., ZHAO, T., HADJIS, S., PEDRAM, A., KOZYRAKIS, C., AND OLUKOTUN, K. Plasticine: A reconfigurable architecture for parallel paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 389–402.

[92] PUTNAM, A., CAULFIELD, A., CHUNG, E., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., HASELMAN, M., HAUCK, S., HEIL, S., HORMATI, A., KIM, J.-Y., LANKA, S., LARUS, J., PETERSON, E., POPE, S., SMITH, A., THONG, J., XIAO, P. Y., AND BURGER, D. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)* (June 2014).

[93] RUPNOW, K., FU, W., AND COMPTON, K. Block, drop or roll(back): Alternative preemption methods for RH multi-tasking. In *FCCM 2009, 17th IEEE Symposium on Field Programmable Custom Computing Machines, Napa, California, USA, 5-7 April 2009, Proceedings* (2009), pp. 63–70.

[94] SHAFIEE, A., GUNDU, A., SHEVGOOR, M., BALASUBRAMONIAN, R., AND TIWARI, M. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture* (New York, NY, USA, 2015), MICRO-48, ACM, pp. 89–101.

[95] SHAN, Y., WANG, B., YAN, J., WANG, Y., XU, N.-Y., AND YANG, H. Fpmr: Mapreduce framework on fpga. In *FPGA* (2010), P. Y. K. Cheung and J. Wawrzynek, Eds., ACM, pp. 93–102.

[96] SHARMA, H., PARK, J., AMARO, E., THWAITES, B., KOTHA, P., GUPTA, A., KIM, J. K., MISHRA, A., AND ESMAEILZADEH, H. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures* (2016).

[97] SIDLER, D., ISTVÁN, Z., OWAIDA, M., AND ALONSO, G. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 403–415.

[98] SIMMLER, H., LEVINSON, L., AND MÄNNER, R. Multitasking on fpga coprocessors. *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing* (2000), 121–130.

[99] SO, H. K.-H., AND BRODERSEN, R. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. *ACM Trans. Embed. Comput. Syst. 7*, 2 (Jan. 2008), 14:1–14:28.

[100] SO, H. K.-H., AND BRODERSEN, R. W. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, EECS Department, University of California, Berkeley, Jul 2007.

[101] SO, H. K.-H., AND WAWRZYNEK, J. Olaf'16: Second international workshop on overlay architectures for fpgas. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2016), FPGA '16, ACM, pp. 1–1.

[102] STEIGER, C., WALDER, H., AND PLATZNER, M. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers 53*, 11 (Nov 2004), 1393–1407.

[103] STITT, G., AND COOLE, J. Intermediate fabrics: Virtual architectures for near-instant fpga compilation. *IEEE Embedded Systems Letters 3*, 3 (Sept 2011), 81–84.

[104] SUDA, N., CHANDRA, V., DASIKA, G., MOHANTY, A., MA, Y., VRUDHULA, S., SEO, J.-S., AND CAO, Y. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2016), FPGA '16, ACM, pp. 16–25.

[105] TAYLOR, M. B. Bitcoin and the age of bespoke silicon. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (2013), IEEE Press, p. 16.

[106] THOMAS, D., AND MOORBY, P. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[107] TSUTSUI, A., MIYAZAKI, T., YAMADA, K., AND OHTA, N. Special purpose fpga for high-speed digital telecommunication systems. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors* (Washington, DC, USA, 1995), ICCD '95, IEEE Computer Society, pp. 486–491.

[108] WASSI, G., BENKHELIFA, M. E. A., LAWDAY, G., VERDIER, F., AND GARCIA, S. Multi-shape tasks scheduling for online multitasking on fpgas. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on* (May 2014), pp. 1–7.

[109] WATKINS, M. A., AND ALBONESI, D. H. Remap: A reconfigurable heterogeneous multicore architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43, IEEE Computer Society, pp. 497–508.

[110] WEERASINGHE, J., ABEL, F., HAGLEITNER, C., AND HERKERSDORF, A. Enabling fpgas in hyperscale data centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), Beijing, China, August 10-14, 2015* (2015), pp. 1078–1086.

[111] WERNER, S., OEY, O., GÖHRINGER, D., HÜBNER, M., AND BECKER, J. Virtualized on-chip distributed computing for heterogeneous reconfigurable multi-core systems. In *Proceedings of the Conference on Design, Automation and Test in Europe* (San Jose, CA, USA, 2012), DATE '12, EDA Consortium, pp. 280–283.

[112] WEST, B., CHAMBERLAIN, R. D., INDECK, R. S., AND ZHANG, Q. An fpga-based search engine for unstructured database. In *Proc. of 2nd Workshop on Application Specific Processors* (2003), vol. 12, pp. 25–32.

[113] WIERSEMA, T., BOCKHORN, A., AND PLATZNER, M. Embedding FPGA overlays into configurable systems-on-chip: Reconos meets ZUMA. In *ReConFig* (2014), IEEE, pp. 1–6.

[114] WINTERSTEIN, F., FLEMING, K., YANG, H.-J., BAYLISS, S., AND CONSTANTINIDES, G. Matchup: Memory abstractions for heap manipulating programs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 136–145.

[115] ZHANG, C., LI, P., SUN, G., GUAN, Y., XIAO, B., AND CONG, J. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 161–170.

[116] ZHANG, C., LI, P., SUN, G., GUAN, Y., XIAO, B., AND CONG, J. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2015), ACM, pp. 161–170.

[117] ZHAO, R., SONG, W., ZHANG, W., XING, T., LIN, J.-H., SRIVASTAVA, M. B., GUPTA, R., AND ZHANG, Z. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *FPGA* (2017), pp. 15–24.