Compiler-Driven FPGA Virtualization with SYNERGY

Joshua Landgraf
The University of Texas at Austin
Austin, Texas, USA
jland@cs.utexas.edu

Tiffany Yang
The University of Texas at Austin
Austin, Texas, USA
tiffanyyang@utexas.edu

Will Lin
The University of Texas at Austin
Austin, Texas, USA
wlsaidhi@utexas.edu

Christopher J. Rossbach
The University of Texas at Austin
VMware Research Group
Katana Graph
Austin, Texas, USA
rossbach@cs.utexas.edu

Eric Schkufza
Amazon
Palo Alto, California, USA
eric.schkufza@gmail.com

ABSTRACT

FPGAs are increasingly common in modern applications, and cloud providers now support on-demand FPGA acceleration in data centers. Applications in data centers run on virtual infrastructure, where consolidation, multi-tenancy, and workload migration enable economies of scale that are fundamental to the provider's business. However, a general strategy for virtualizing FPGAs has yet to emerge. While manufacturers struggle with hardware-based approaches, we propose a compiler/runtime-based solution called Synergy. We show a compiler transformation for Verilog programs that produces code able to yield control to software at *sub-clock-tick* granularity according to the semantics of the original program. Synergy uses this property to efficiently support core virtualization primitives: suspend and resume, program migration, and spatial/temporal multiplexing, on hardware which is available *today*. We use Synergy to virtualize FPGA workloads across a cluster of Altera SoCs and Xilinx FPGAs on Amazon F1. The workloads require no modification, run within $3-4\times$ of unvirtualized performance, and incur a modest increase in FPGA fabric utilization.

CCS CONCEPTS

• **Hardware → Hardware description languages and compilation**; *Reconfigurable logic and FPGAs*; • **Software and its engineering → Compilers**; **Operating systems**.

KEYWORDS

Compilers, FPGAs, Virtualization, Operating Systems

ACM Reference Format:
Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. 2021. Compiler-Driven FPGA Virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446755

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446755

1 INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) combine the functional efficiency of hardware with the programmability of software. FPGAs can exceed general-purpose CPU performance by orders of magnitude [13, 76] and offer lower cost and time to market than ASICs. FPGAs have become a compelling acceleration alternative for machine learning [16, 80, 86, 100], databases [12, 43, 61], finance [39, 58], graph processing [20, 69], communication [10, 38, 41, 76, 90], and image processing [68]. In data centers, FPGAs serve diverse hardware needs with a single technology. Amazon now provides F1 instances with large FPGAs attached [23] and Microsoft deploys FPGAs in new data center construction [64].

Virtualization is fundamental to data centers. It decouples software from hardware, enabling economies of scale through consolidation. However, a standard technique for virtualizing FPGAs has yet to emerge. There are no widely agreed upon methods for supporting key primitives such as *workload migration* (suspending and resuming a hardware program or relocating it between FPGAs mid-execution) or *multi-tenancy* (multiplexing multiple hardware programs on a single FPGA). Better virtualization support is required for FPGAs to become a mainstream accelerator technology.

Virtualizing FPGAs is difficult because they lack a well-defined interposable *application binary interface* (ABI) and state capture primitives. On CPUs, hardware registers are restricted to a small, static set and access to data is abstracted through virtual memory, making it trivial to save and restore state. In contrast, the state of an FPGA program is distributed throughout its reprogrammable fabric in a *program-* and *hardware-dependent* fashion, making it inaccessible to the OS. Without knowing how programs are compiled for an FPGA, there is no way to share the FPGA with other programs or to relocate programs mid-execution. FPGA vendors are pursuing hardware-based solutions to enable sharing by partitioning the device into smaller, isolated fabrics. However, lacking state capture primitives, this does not solve the fundamental problem and cannot support features like workload migration.

We argue that the right place to support FPGA virtualization is in a combined compiler/runtime environment. Our system, Synergy, combines a *just-in-time* (JIT) runtime for Verilog, canonical interfaces to OS-managed resources, and an OS-level protection layer to abstract and isolate shared resources. The key insight behind Synergy is that a compiler can *transparently* re-write Verilog code

to compensate for the missing ABI and explicitly expose application state to the OS. The core technique in Synergy is a static analysis to transform the user's code into a distributed-system-like *intermediate representation* (IR) consisting of monadic sub-programs which can be moved back and forth mid-execution between a software interpreter and native FPGA execution. This is possible because the transformations produce code that can trap to software at arbitrary execution points without violating the semantics of Verilog.

Synergy's first contribution is a set of compiler transformations to produce code that can be interrupted at *sub-clock-tick granularity* (§3) according to the semantics of the original program. This allows Synergy to support a large class of *unsynthesizable* Verilog. Traditional Verilog uses unsynthesizable language constructs for testing and debugging in a simulator: Synergy uses these to expose interfaces to OS-managed resources and to start, stop, and save the state of a program at any point in its execution. This allows Synergy to perform context switch and workload migration without hardware support or modifications to Verilog.

Synergy's second contribution is a new technique for FPGA multi-tenancy (§4). Synergy introduces a hypervisor layer into the compiler's runtime which can combine the sub-program representations from multiple applications into a single hardware program, which is kept hidden from those instances. This module is responsible for interleaving asynchronous data and control requests between each of those instances and the FPGA. In contrast to hardware-based approaches, manipulating each instance's state is straightforward, as the hypervisor has access to every instance's source and knows how it is mapped onto the device.

Synergy's final contribution is a compiler backend targeting an OS-level protection layer for process isolation, fair scheduling, and cross-platform compatibility (§5). Recent OS-FPGA proposals harden vendor *shells* and export interfaces for the application to assist the OS with state capture for context switch [46, 63]. A major obstacle to using these systems is the requirement that the developer implement those state capture interfaces. Synergy satisfies the state capture requirement transparently by using compiler analysis to identify the set of variables that comprise a program's state and emitting code to interact with state capture and quiescence interfaces. For applications which natively support such interfaces, Synergy can use these to dramatically reduce overhead for context switch and migration.

Our Synergy prototype extends the Cascade [78] JIT compiler and composes it with the AmorphOS [46] FPGA OS. We measure Synergy in real-world contexts that represent the heterogeneity of the data center. We show the ability to suspend and resume programs running on a cluster of Altera SoCs and Xilinx FPGAs running on Amazon's F1 cloud instances, to transition applications between the two, and to temporally and spatially multiplex both devices efficiently with strong OS-level isolation guarantees. This is done without exposing the architectural differences between the platforms, or requiring extensions to the Verilog language or modifications to the programs. We achieve performance within $3-4\times$ of unvirtualized code with a reasonable fabric cost.

```
 1: module Module(
 2:  input wire clock,
 3:  output wire[31:0] res
 4: );
 5:   wire[31:0] x = 1, y = x + 1;
 6:   reg[63:0] r = 0;
 7:   SubModule sm(clock);
 8:
 9:   always @(posedge clock) begin
10:     $display(r); // Prints 0,3,3,...
11:     r = y;
12:     $display(r); // Prints ?,2,2,...
13:     r <= 3;
14:     $display(r); // Prints ?,2,2,...
15:   end
16:
17:   always @(posedge clock) fork
18:     $display(r); // Prints ?,?,?,...
19:   join
20:
21:   assign res = r[47:16] & 32'hf0f0f0f0;
22: endmodule
```

Figure 1: A simple Verilog module. Verilog supports a combination of sequential and concurrent semantics.

2 BACKGROUND

Verilog [88] is one of two standard HDLs used to program FPGAs. VHDL [6] is essentially isomorphic. Verilog consists of *synthesizable* and *unsynthesizable* constructs. Synthesizable Verilog describes computation which can be lowered onto an FPGA. Unsynthesizable Verilog includes tasks such as print statements, which are more expressive and aid in debugging, but must be executed in software.

Verilog programs are declarative and organized hierarchically in units called *modules*. An example Verilog module is shown in Figure 1. The interface to a module is defined in terms of its input/output ports (`clock`, `res`). Its semantics are defined in terms of arbitrary-width wires (`x`,`y`) and registers (`r`), logic gates (e.g. `&`), primitive arithmetic (e.g. `+`), and nested sub-modules (`sm`). The value of a wire is functionally determined by its inputs (lines 5, 21), whereas a register is updated at discrete intervals (lines 6, 11, 13). For brevity, our discussion ignores Verilog's rules of type inference (`reg` may be demoted to `wire`). Synergy *does not*.

Verilog supports sequential and concurrent semantics. Continuous assignments (lines 5, 21) are scheduled when the value of their right-hand-side changes. Procedural blocks (lines 9–19) are scheduled when their guard is satisfied (e.g. `clock` changes from 0 to 1). The ordering of these events is undefined, and their evaluation is non-atomic. Any of the statements in a `fork/join` block may be evaluated in any order. Only a `begin/end` block is guaranteed to be evaluated sequentially. Procedural blocks can contain two types of assignments to registers: blocking (=) and non-blocking (<=). Blocking assignments are executed immediately, whereas non-blocking assignments must wait until all continuous assignments or control blocks are finished.

When used idiomatically, these semantics map directly onto hardware primtives: wires *appear* to change value instantly and registers *appear* to change value with the clock. However, unsynthesizable statements have no analogue. The print statement on line 18 is non-deterministic, it can be interleaved with any assignment in lines 10–14. So too is the first execution of lines 12 and 14, which can be interleaved with the assignment on line 5. While the assignment on line 11 is visible immediately, the assignment on line 13 is only performed after every block and assignment has been scheduled, thus the value 3 only appears the second time line 10 is executed.

2.1 Cascade

Cascade is the first JIT compiler for Verilog. Using Cascade, Verilog is parsed one line at a time, added to the user's program, and its side effects made visible immediately. This can include the results of executing unsynthesizable Verilog. While JIT compilation is orthogonal to Synergy, Cascade's runtime techniques are a fundamental building block. Cascade applies transformations to the user's program that produce code which can trap into the Cascade runtime at the end of the logical clock tick. These traps are used to handle unsynthesizable statements in a way that is consistent with Verilog's scheduling semantics, even during hardware execution. Synergy improves upon this to trap into the runtime at sub-clock-tick granularity according to the semantics of the original program and to enable context switch (§3).

Cascade uses the syntax of Verilog to manage programs at the module granularity. Its IR expresses a distributed system of Verilog *sub-programs*, each corresponding to a single module in the user's program. A sub-program's state is represented by a data structure known as an *engine*. Sub-programs start as low-performance, software-simulated engines and are replaced over time by high-performance FPGA-resident engines. Cascade retains the flexibility to relocate engines by imposing a constrained ABI on its IR, mediated by messages sent over the runtime's data/control plane. Relevant to our discussion is a subset of that ABI: `get/set` and `evaluate/update` messages. The `get/set` messages read and write an engine's inputs, outputs, and program variables. The `evaluate/update` messages request that an engine run until no more continuous assigns or procedural blocks can be scheduled, and latch the result of non-blocking assignments, respectively.

Unsynthesizable traps are placed on an ordered interrupt queue and evaluated between clock ticks, when changes to engine state have fixed-pointed and the program is in a consistent state. This limits support for unsynthesizable Verilog to output-only. For example, print statements can occur at any point in a program, but their side effects are only made visible between clock-ticks. There is no way to schedule an interrupt between the statements in a `begin/end` block, block on the result, and continue execution. Synergy removes these limitations.

2.2 AmorphOS

AmorphOS is an FPGA runtime infrastructure which supports cross-program protection and compatibility at very high degrees of multi-tenancy. AmorphOS allows hardware programs to scale dynamically in response to FPGA load and availability and can transparently change mappings between user logic and FPGA fabric to increase utilization by avoiding fragmentation. AmorphOS extends processes with *Morphlets*, an abstraction for FPGA-based execution. AmorphOS can spatially share an FPGA among Morphlets from different protection domains and falls back to time-sharing when space-sharing is infeasible. AmorphOS mediates OS-managed resources through a shell-like component called a hull, which provides an isolation boundary and a compatibility layer. This enables AmorphOS to co-locate several Morphlets in a single reconfigurable *zone* to increase utilization without compromising security. AmorphOS leaves the problems of efficient context switch, over-subscription, and support for multiple FPGAs mostly unsolved by relying on a programmer-exposed quiescence interface and a programmer-populated compilation cache.

AmorphOS's quiescence interface forces the programmer to write state-capture code (§1), which requires explicitly identifying live state. The interface is simple to support for request-response style programs such as DNN inference acceleration [80], but difficult, say, for a RISC core that can execute unbounded sequences of instructions. This can subject an OS-scheduler to arbitrary latency based on a program's implementation and introduces the need for forced revocation mechanisms as a fallback. Transparent state capture mechanisms which insulate the programmer from low-level details of on-fabric state are not supported.

3 VIRTUALIZATION PRIMITIVES

In this Section, we describe a sound transformation for Verilog that allows a program to yield control at sub-clock-tick granularity. This transformation allows Synergy to support the entire unsynthesizable Verilog standard from hardware, including `$save` and `$restart`, the two primitives which are necessary for supporting workload migration. We frame this discussion with a file IO case study. While file IO is not necessary for virtualization, it provides a clear perspective from which to understand the transformation. Moreover, supporting file IO in hardware enables a more expressive programming environment in which applications have access to OS-managed resources through canonical hardware-independent interfaces. We leave a discussion of other applications which can benefit from the ability to yield control at the sub-clock-cycle granularity (say, step-through debuggers) to future work.

3.1 Motivating Example: File I/O

Consider the program shown in Figure 2, which uses unsynthesizable IO tasks to sum the values contained in a large file. The program opens the file (line 4) and on every clock tick, attempts to read a 32-bit value (line 9). When the program reaches the end-of-file, it prints the running sum and returns control to the host (lines 10-12). Otherwise, it adds the value to the running sum and continues (line 14). While this program is simple, its structure is typical of applications that perform streaming computation over large data-sets.

The key obstacle to supporting this program is that the IO tasks introduce data-dependencies within a single clock-tick. The end-of-file check on line 10 depends on the result of the read operation on line 9, as does the assignment on line 14. Because the semantics of these operations involve an interaction with the file system, we

```
 1: module M(
 2:   input wire clock
 3: );
 4:   integer fd = $fopen("path/to/file");
 5:   reg[31:0] r = 0;
 6:   reg[127:0] sum = 0;
 7
 8:   always @(posedge clock) begin
 9:     $fread(fd, r);    // TASK 1
10:     if ($feof(fd))    // FEOF 1
11:       $display(sum); // TASK 2
12:       $finish(0);     // TASK 3
13:     else
14:       sum <= sum + r;
15:   end
16: endmodule
```

Figure 2: Motivating example. A Verilog program that uses unsynthesizable IO to sum the values in a large file.

$$
\begin{aligned}
\mathcal{S}(\texttt{fork } s_1 \ldots s_n \texttt{ join}) &\Rightarrow \texttt{begin } s_1 \ldots s_n \texttt{ end} \\
\mathcal{S}(\texttt{begin } s_1 \ldots s_n \texttt{ end}) &\Rightarrow \mathcal{S}(s_1) \ldots \mathcal{S}(s_n) \\
\mathcal{S}\begin{pmatrix}\texttt{always } @(\epsilon_1)\ s_1 \\ \ldots \\ \texttt{always } @(\epsilon_n)\ s_n\end{pmatrix} &\Rightarrow \begin{array}{l}\texttt{always } @(\epsilon_1 \texttt{ or } \ldots \texttt{ or } \epsilon_n) \\ \quad \texttt{if } (\mathcal{G}(\epsilon_1))\ \texttt{begin } \mathcal{S}(s_1)\ \texttt{end} \\ \quad \ldots \\ \quad \texttt{if } (\mathcal{G}(\epsilon_n))\ \texttt{begin } \mathcal{S}(s_n)\ \texttt{end}\end{array} \\
\mathcal{S}(s) &\Rightarrow s \\
\mathcal{G}(\texttt{posedge x}) &\Rightarrow \texttt{\_\_pos\_x} \\
\mathcal{G}(\texttt{negedge x}) &\Rightarrow \texttt{\_\_neg\_x} \\
\mathcal{G}(\texttt{x}) &\Rightarrow \texttt{\_\_any\_x}
\end{aligned}
$$

Figure 3: Transformations used to establish the invariant that procedural logic appears in a single control statement.

must not only pause the execution of the program mid-cycle while control is transferred to the host, but also block for an arbitrary amount of time until the host produces a result. Our solution is to transform the program into a state machine which implements a coroutine style semantics. While a programmer could adopt this idiom, the changes would harm both readability and maintainability.

3.2 Scheduling Transformations

Synergy uses the transformations sketched in Figure 3 to establish the invariant that all procedural logic appears in a single control statement. Any `fork/join` block may be replaced by an equivalent `begin/end` block, as the sequential semantics of the latter are a valid scheduling of the former. Also, any nested set of `begin/end` blocks may be flattened into a single block as there are no scheduling constraints implied by nested blocks. Next, we combine every procedural control statement in the program into a single statement called *the core*. The core is guarded by the union of the events that guard each individual statement. This is sound, as Verilog only allows disjunctive guards. Next, we set the body of

$$
\begin{aligned}
\delta(\texttt{x}) &\Rightarrow \begin{array}{l}\texttt{reg \_\_px;} \\ \texttt{always @(posedge \_\_clk)} \\ \quad \texttt{\_\_px <= x;}\end{array} \\
\mathcal{D}(\texttt{posedge x}) &\Rightarrow \begin{array}{l}\delta(x) \\ \texttt{wire \_\_pos\_x = !\_\_px \& x;}\end{array} \\
\mathcal{D}(\texttt{negedge x}) &\Rightarrow \begin{array}{l}\delta(x) \\ \texttt{wire \_\_neg\_x = \_\_px \& !x;}\end{array} \\
\mathcal{D}(\texttt{x}) &\Rightarrow \begin{array}{l}\delta(x) \\ \texttt{wire \_\_any\_x = \_\_px != x;}\end{array} \\
C(\texttt{always @}(\mathcal{E})\ s) &\Rightarrow \begin{array}{l}\forall_{\epsilon \in \mathcal{E}}\ \mathcal{D}(\epsilon) \\ \texttt{reg[31:0] \_\_state;} \\ \texttt{reg[31:0] \_\_task;} \\ \texttt{always@(posedge \_\_clk)} \\ \quad Lower(\mathcal{M}(s))\end{array}
\end{aligned}
$$

Figure 4: Transformations used to convert the core control statement into a form compatible with the Synergy ABI.

the core to a new `begin/end` block containing the conjunction of the bodies of each individual block. This is sound as well, as sequential execution is a valid scheduling of active procedural control statements. Finally, we guard each conjunct with a name-mangled version of its original guard (e.g. `__pos_x`, details below) as all of the conjuncts would otherwise be executed when the core is triggered. We note that these transformations are sound even for programs with multiple clock domains.

3.3 Control Transformations

The transformations in Figure 4 modify the control structure of the core so that it is compatible with the Cascade ABI. Recall that the Cascade ABI requires that all of the inputs to an IR sub-program *including clocks* will be presented as values contained in `set` messages which may be separated by many native clock cycles on the target device. Thus we declare state to hold the previous values of variables that appear in the core's guard, and wires that capture their semantics in the original program (e.g. `__pos_x` is true whenever a `set` message last changed `x` from false to true. We also declare new variables (`__state` and `__task`) to track the control state of the core, and whether a system task requires the attention of the runtime. Finally, we replace the core's guard by a posedge trigger for the native clock on the target device (`__clk`).

3.4 State Machine Transformations

The body of the core is lowered onto a state machine with the following semantics. States consist of as many synthesizable statements as possible and are terminated either by unsynthesizable tasks or the guard of an `if` or `case` statement. A new state is created for each branch of a conditional statement, and an SSA-style phi state is used to rejoin control flow.

A compiler has flexibility in how it chooses to lower the resulting state machine onto Verilog text. Figure 5 shows one possible implementation. Each state is materialized as an `if` statement that performs the logic associated with the state, takes a transition, and sets the `__task` register if the state ended in an unsynthesizable

```
 1: module M(
 2:   input wire __clk,
 3:   input wire[5:0] __abi
 4: );
 5:   reg __pclock;
 6:   always @(posedge __clk)
 7:     __pclock <= clock;
 8:   wire __pos_clock = !__pclock & clock;
 9:
10:   reg[31:0] __state = 5;
11:   reg[31:0] __task = `NONE;
12:   always @(posedge __clk)
13:     if (__pos_clock)
14:       {__task,__state} = {`TASK_1, 1};
15:     if ((__state == 1) && __cont)
16:       __task = `NONE;
17:       __state = __feof1 ? 2 : 4;
18:     if ((__state == 2) && __cont)
19:       {__task,__state} = {`TASK_2, 3};
20:     if ((__state == 3) && __cont)
21:       {__task,__state} = {`TASK_3, 5};
22:     if ((__state == 4) && __cont)
23:       __sum_next <= sum + r;
24:       {__task,__state} = {`NONE, 5};
25:     if ((__state == 5) && __cont)
26:       {__task,__state} = {`NONE, 5};
27:
28:   wire __tasks = __task != `NONE;
29:   wire __final = __state == 5;
30:   wire __cont = (__abi == `CONT) |
31:                 (!__final & !__tasks);
32:   wire __done = __final & !__tasks;
33: endmodule
```

Figure 5: The motivating example after modification to yield control to the runtime at the sub-clock-tick granularity.

statement. Control enters the first state when the variable associated with the original guard (`__pos_clock`) evaluates to true, and continues via the fall-through semantics of Verilog until a task is triggered. When this happens, a runtime which is compatible with the Cascade ABI can take control, place its results (if any) in the appropriate hardware location, and yields back to the target device by asserting the `__cont` signal. When control enters the final state, the program asserts the `__done` signal, indicating that there is no further work to do be done. Collectively, these steps represent the compute portion of the `evaluate` and `update` requests required by the ABI.

3.5 Workload Migration

With these transformations, support for the `$save` and `$restart` system tasks is straightforward. Both can be materialized as traps triggered by the value of `__task` in a runtime compatible with the Cascade ABI. The former prompts the runtime to save the state of the program through a series of `get` requests, and the latter

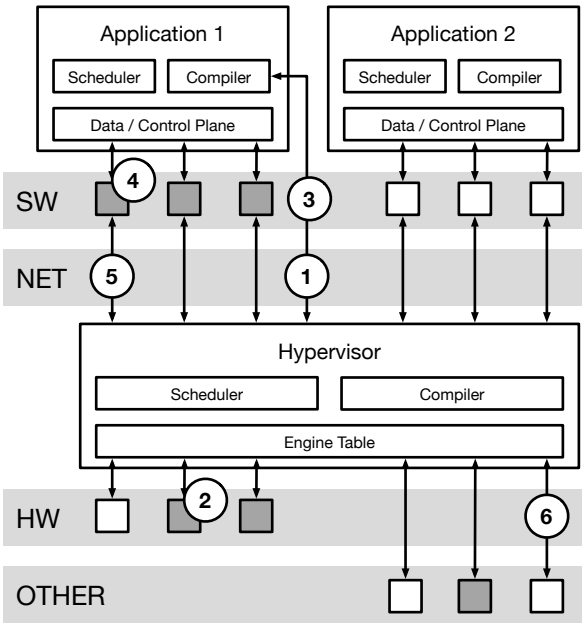

Figure 6: The SYNERGY virtualization layer. Sub-programs from multiple applications are combined on one target.

prompts a sequence of `set` requests. Either statement can be triggered in the course of normal program execution, or via an eval statement. Once a program's state is read out, it can be suspended, migrated to another physical machine if necessary, and resumed.

4 HYPERVISOR DESIGN

In this section we describe SYNERGY's support for the two primary forms of hardware multiplexing: *spatial* (where two programs are run simultaneously on the same fabric) and *temporal* (where two programs share resources using time-slice scheduling). SYNERGY provides an indirection layer that allows multiple runtime instances to share a compiler at the hypervisor layer.

4.1 Program Coalescing

Figure 6 shows a sketch of SYNERGY during an execution in which two applications share a single hardware fabric. In addition to the scheduler and data/control plane introduced in § 2, we have called out the compilers associated with both the runtime instance running those applications, and the SYNERGY hypervisor. These compilers are responsible for lowering a sub-program onto a target-specific engine that satisfies Cascade's distributed-system ABI.

The compiler in the runtime instance connects to the hypervisor ①, which runs on a known port. It sends the source code for a sub-program over the connection, where it is passed to the native hardware compiler in the hypervisor, which produces a target-specific implementation of an engine and places it on the FPGA fabric ②. The hypervisor responds with a unique identifier representing the engine ③ and the runtime's compiler creates an engine which remains permanently in software and is configured with the

unique identifier ④. The resulting engine interacts with the runtime as usual. However, its implementation of the Cascade ABI is simply to forward requests across the network to the hypervisor ⑤ and block further execution until a reply is obtained.

The key idea that makes this possible is that the compiler in the hypervisor has access to the source code for every sub-program in every connected instance. This allows the compiler to support multitenancy by combining the source code for each sub-program into a single monolithic program. Whenever the text of any sub-program changes, the combined program is recompiled to support the new logic. Whenever an application finishes executing, all of its sub-programs are flagged for removal on the next recompilation. The implementation of this combined program is straightforward. The text of the sub-programs is placed in modules named after their unique hypervisor identifier. The combined program concatenates these modules together and routes ABI requests to the appropriate module based on their identifier. By isolating both sub-program code and communication, the FPGA fabric can be shared securely.

The overhead of the Synergy hypervisor depends primarily on the application. While regular communication can become a bottleneck, optimizations [78] can reduce the ABI requests between the runtime and an engine to a tolerable level. For batch-style applications, fewer than one ABI request per second is required, and we are able to achieve near-native performance even for programs separated from the hypervisor by a network connection. In contrast, applications that invoke frequent ABI calls (e.g. for file I/O) will have overheads that scale with the frequency of interaction. While our discussion presents a hypervisor which compiles all of its sub-programs to FPGA fabric, this is not fundamental. The virtualization layer nests, and it is both possible and performant for a hypervisor to delegate the compilation of a sub-program to a second hypervisor ⑥, say if the device is full.

4.2 Scheduling State-Safe Compilation

The Synergy hypervisor schedules ABI requests sequentially to avoid resource contention. The one exception is compilation, which can take a very long time to complete. If compilation were serialized between ABI requests, it could render applications non-interactive. But scheduling compilation asynchronously leads to a key implementation challenge: changing the text of one instance's sub-programs requires that the entire FPGA be reprogrammed, a process which would destroy all connected instances' state. The solution is to schedule these destructive events when all connected instances are between logical clock-ticks and have saved their state.

Figure 7 shows the handshake protocol used to establish these invariants. Compilation requests are scheduled asynchronously ①, and run until they would do something destructive. The hypervisor then sends a request to every connected runtime instance ② to schedule an interrupt between their logical clock-ticks when they are in a consistent state ③. The interrupt causes the instances to send `get` requests to Synergy ④ to save their program state. When they have finished, the instances send a reply indicating it is safe to reprogram the device ⑤ and block until they receive an acknowledgement. Compilation proceeds after the final reply. The device is reprogrammed and the handshake finishes in the opposite fashion. The hypervisor informs the instances it is finished, they send `set` requests to restore their state on the target device and control proceeds as normal.

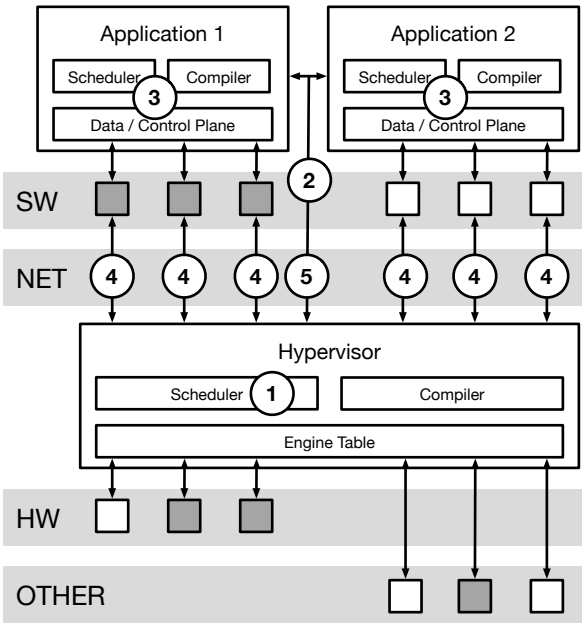

Figure 7: The handshake protocol used to establish state-safe interrupts in the Synergy's scheduler.

4.3 Multitenancy

Collectively, these techniques suffice to enable multitenancy. Spatial multiplexing is accomplished by combining the sub-programs from each connected runtime into a single monolithic program on the target device. Temporal multiplexing is accomplished by serializing ABI requests that involve an IO resource (say, a connection to an in-memory dataset) which is in use by another sub-program. Sharing preserves tenant protection boundaries using AmorphOS, which provides support for isolating sub-programs sharing the FPGA fabric (§2.2).

5 IMPLEMENTATION

Our implementation of Synergy comprises the hypervisor described in §4, compilation passes which enable sub-clock-tick granularity support for the unsynthesizable primitives described in §3, and both Intel and AmorphOS backends.

5.1 Intel Backends

Our implementation of Synergy extends Cascade's support for the DE10 Nano SoC to the full family of Intel devices that feature reprogrammable fabric and an ARM core. This describes a range of targets, including the high-performance Stratix 10. The core feature these targets share is that they support Intel's Avalon interface for memory-mapped IO. This allows us to lower the transformations described in §3 onto a Verilog module that converts reads and writes on the Avalon memory-mapped slave interface into ABI requests.

```
1: module Root();
2:   (* non_volatile *) reg[31:0] x;
3:   reg[31:0] y;
4:   always @(posedge clock.val)
5:     if (...) $yield;
6:   // Additional program logic ...
7: endmodule
```

Figure 8: The `$yield` task enables Synergy's quiescence interface. Volatile variables must be managed by the user.

Adding support for a new Intel backend amounts to compiling this module in a hardware context which contains an Avalon memory-mapped master whose control registers are `mmap`'ed into the same process space as the runtime or hypervisor. Compiling the logic for these interfaces can be expensive, so Synergy augments the Intel family of backends with a compilation cache similar to the one used by AmorphOS. This allows Synergy to transition gracefully from using Cascade's JIT interface for iterative development to using fast database lookup for mature virtualized applications that require rapid transitions to hardware execution. Unlike the AmorphOS backend described below, our DE10 backend does not yet support the AmorphOS protection layer.

5.2 AmorphOS Backends

Synergy uses a similar strategy for supporting multiple AmorphOS backends. We lower the transformations described in §3 onto a Verilog module implementing the AmorphOS CntrlReg interface. The module runs as a Morphlet inside the AmorphOS hull, which provides cross-domain protection and thus preserves tenant isolation boundaries. It also enables Synergy to take advantage of the large degree of multitenancy AmorphOS offers. The Synergy hypervisor communicates with the Morphlet via a library from AmorphOS. This makes adding support for a new AmorphOS backend as simple as bringing AmorphOS up on that target.

A key difference between the DE10 and F1 is the size and speed of the reprogrammable fabric they provide. Each F1 FPGA has 10× more LUTs and operates 5× faster than a DE10. This enables Synergy to accelerate larger applications, but also makes achieving timing closure challenging. Synergy adopts two solutions. The first is to pipeline access to program variables which are modified by `get/set` requests. For writes, Synergy adds buffer registers between the AmorphOS hull and the variables. For reads, Synergy builds a tree with the program's variables at the leaves and the hull at the trunk. By adding buffer registers at certain branches, this logic is removed from the critical timing path. The second solution is to iteratively reduce the target device frequency until the design does meets timing. This is automated by Synergy's build scripts, which can also preserve synthesis, placement, and routing data to help offset the cost of performing multiple compiles.

5.3 Quiescence Interface

AmorphOS provides a quiescence interface that notifies applications when they will lose access to the FPGA (e.g. during reconfiguration), allowing them to quiesce and back up their state accordingly.

Table 1: Benchmarks were chosen to represent a mix of batch- and streaming-style computation (marked ⋆).

Name	Description
adpcm	Pulse-code modulation encoder/decoder
bitcoin	Bitcoin mining accelerator
df	Double-precision arithmetic circuits
mips32	Bubble-sort on a 32-bit MIPS processor
nw⋆	DNA sequence alignment
regex⋆	Streaming regular expression matcher

Synergy supports this interface by handling the implementation of execution control and state management for developers. By default, all program variables are considered `non_volatile`, and will be saved and restored automatically.

For applications that implement quiescence, Synergy introduces an optional, non-standard `$yield` task, shown in Figure 8. Developers can assert `$yield` to signal that the program has entered an application-specific consistent state. When present, Synergy will only perform state-safe compilations at the end of a logical clock tick in which `$yield` was asserted. The use of `$yield` causes stateful program variables to be considered *volatile* by default. Volatile variables are ignored by state-safe compilations, making it is the user's responsibility to restore or reset their values at the beginning of each logical clock tick following an invocation of `$yield`. Users may override this behavior by annotating a variable as `non_volatile`.

6 EVALUATION

We evaluated Synergy using a combination of Altera DE10 SoCs and Amazon F1 cloud instances. The DE10s consist of a Cyclone V device [36] with an 800 MHz dual core ARM processor, reprogrammable fabric of 110K LUTs, 50 MHz clock, and 1 GB of shared DDR3 memory. Synergy's DE10 backend was configured to generate bitstreams using Intel's Quartus Lite Compiler and to interact with the DE10s' FPGA fabric via a soft-IP implementation of an Avalon Memory-Mapped master. The F1 cloud instances [23] support multiple Xilinx UltraScale+ VU9Ps running at 250 MHz and four 16 GB DDR4 channels. Synergy's F1 backend was configured to use build tools adapted from the F1 toolchain and to communicate with the instances' FPGA fabric over PCIe.

Table 1 summarizes the benchmarks used in our evaluation, a combination of batch and streaming computations. The ability to handle file IO directly from hardware made the latter easy to support, as developing these benchmarks amounted to repurposing testbench code designed for functional debugging. Benchmarks were compiled prior to running experiments to prime Synergy's bitstream caches. This was appropriate as Synergy's goal is to provide virtualization support for applications which have spent sufficient time in the compile-test-debug cycle to converge on a stable design.

We find that Synergy improves upon Cascade's performance. Despite targeting a 5× higher frequency on F1, implementing a more complex program transformation, and accounting for device frequency overheads, it still achieves a virtual clock frequency [78]

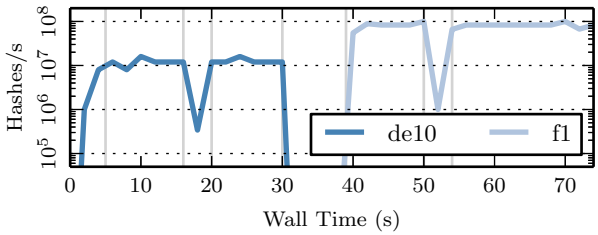

Figure 9: Suspend and Resume. `Bitcoin` is executed on a DE10 target, suspended, saved, and resumed on F1.

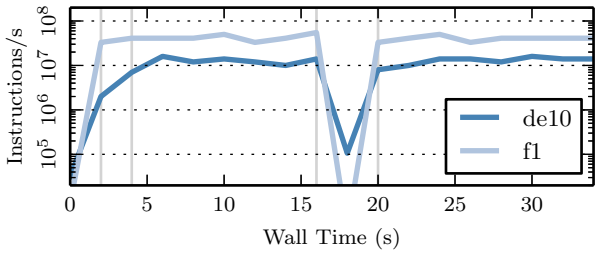

Figure 10: Hardware Migration. `Mips32` begins execution on one target and is migrated mid-execution to another.

within 3 – 4× of native unvirtualized performance and maintains a reasonable fabric cost. While non-negligible, these figures do not represent a lower-bound, and we expect further engineering to reduce them considerably.

6.1 Workload Migration

Figure 9 plots the performance of `bitcoin` as it is moved back and forth between software and hardware on two different target architectures. This workflow is typical of suspend and resume style virtualization. The application combines a block of data with a nonce, applies several rounds of SHA-256 hashing, and loops until it discovers a nonce that produces a hash under a target value.

The application begins execution in a new instance of Synergy and, after running briefly in software, transitions to hardware execution on a DE10 ($t = 5$) where it achieves a peak throughput of 16M nonces evaluated per second. At ($t = 15$) we emit a signal which causes the instance to evaluate a `$save` task. Control then transitions temporarily to software as the runtime evacuates the program's state. The application's throughput drops significantly during this window, but quickly returns to steady-state as control returns to hardware ($t = 22$). Synergy is then terminated ($t = 30$), and similar process is initiated on an F1 instance ($t = 39$). In this case, the instance evaluates a `restart` task to restore the context which was saved on the DE10 ($t = 50$). Due to the larger, higher performance hardware on F1, the program achieves a higher throughput (83M), but suffers from higher performance degradation during the `restart` as it takes longer to reconfigure.

Figure 10 plots the performance of a single-cycle 32-bit MIPS processor consisting of registers, a datapath, and on-chip memory.

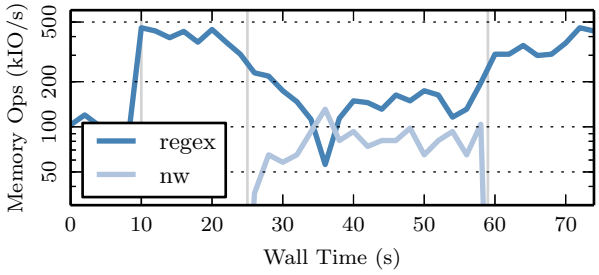

Figure 11: Temporal Multiplexing. `Regex` and `nw` are time-slice scheduled to resolve contention on off-device IO.

The CPU repeatedly randomizes and sorts an in-memory array, with execution transitioning between two FPGAs. The workload is typical of long-running batch computations which are coalesced to improve data center utilization.

The curves show two different execution contexts: one where the program is migrated between nodes in a cluster of DE10s, and one where it is migrated between F1 instances. The timing of key events is synchronized to highlight the differences between the environments. In both cases control begins in software and transitions shortly thereafter to hardware ($t = 2, 4$) where the targets achieve throughputs of 14M and 41M instructions per second, respectively. At ($t = 15$) we emit a signal which causes both contexts to evaluate `$save`/`$restart` tasks as the program is moved between FPGAs. A short time later ($t = 20$), performance returns to peak. Compared to the previous example and the same experiment run on some of the other benchmarks, the performance degradation during hardware/software transitions is more pronounced for `mips32`, with the virtual frequency temporarily lowering to 2K on F1. This is partially due to the large amount of state which must be managed by `get`/`set` requests compared to other benchmarks (the state of a MIPS processor consists of its registers, data memory, and instruction memory).

6.2 Multitenancy

Figure 11 plots the performance of two streaming-style computations on a DE10. Both read inputs from data files that are too large to store on-chip. The first (`regex`) reads in characters and generates statistics on the stream using a regular expression matching algorithm. The second (`nw`) reads in DNA sequences and evaluates how well they match using a tile-based alignment algorithm.

The regular expression matcher begins execution in a new instance of Synergy and, at time ($t = 10$), transitions to hardware where it achieves a peak throughput of 500,000 reads per second. At ($t = 15$), the sequence aligner begins execution in a second instance of Synergy. For the next few seconds, the performance of the matcher is unaffected. At ($t = 24$), the aligner transitions to hardware and the hypervisor is forced to temporally multiplex the execution of both applications, as they now contend on a common IO path between software and hardware. During the period where both applications are active ($t = 24 – 60$), the matcher's throughput drops to slightly less than 50%. This is due to the hypervisor's

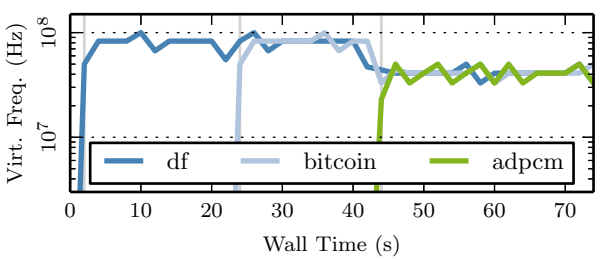

Figure 12: Spatial Multiplexing. `Bitcoin`, `df`, and `adpcm` are co-scheduled on one device without contention.

use of round-robin scheduling and the fact that the primitive read operations performed by the matcher (characters) require less time to run to completion than the primitive read operations performed by the aligner (strings).

At ($t = 60$), the sequence aligner completes execution, and the throughput for the matcher returns to its peak value shortly thereafter. Compared to previous examples, the time required to transition between performance regimes is slightly more pronounced. This is due to Synergy's use of adaptive refinement [78] to determine the time spent in hardware execution before yielding control back to the REPL. It takes several seconds after the aligner finishes execution for Cascade to adjust back to a schedule which achieves peak throughput while also maintaining interactivity.

Figure 12 plots the performance of some batch-style computations on an F1 instance. The first two applications read small inputs sets and transition to long-running computation before returning a result. The former (`df`) performs double-precision floating-point computations characteristic of numeric simulations, and the latter (`bitcoin`) is the miner described in §6.1. Compared to the previous examples, the results are unremarkable. Without resource contention, the hypervisor is able to run both in parallel. The applications begin software execution in separate instances of Synergy ($t = 0, 22$) and after transitioning to hardware ($t = 2, 24$) achieve a virtual clock rate [78] of 83 MHz. At ($t = 42$), another batch-style application that encodes and decodes audio data (`adpcm`) begins execution in a new instance of Synergy. While the hypervisor can run this application in parallel with the first two, lowering its application logic onto the F1 instance causes the resulting design to no longer meet timing at the peak frequency of 250 MHz. To accommodate all three applications, the global clock is set to 125 MHz, reducing their virtual clock frequencies to 41 MHz. The Synergy hypervisor hides the number of applications running simultaneously from the user. As a result, this can lead to unexpected performance regressions in our prototype. Future work can address this by running each application in an appropriate clock domain, with clock-crossing logic added automatically as needed.

6.3 Quiescence

Saving and restoring large volumes of state not only degrades reconfiguration performance (Figure 10) but also requires a large amount of device-side resources to implement (§ 6.4). Synergy's quiescence interface allows developers to signal when a program is quiescent and which variables are stateful at that time. We found that most of our benchmarks had a large number of volatile variables, including 99%, 96%, and 71% of `df`'s, `bitcoin`'s, and `mips32`'s state. For these applications, implementing quiescence resulted in an average LUT and FF savings of 50% and 15%, respectively. In our other benchmarks, 1/8 to 1/4 of the state was volatile. Implementing quiescence for them resulted in an average LUT and FF savings of 2% and 9%, respectively.

6.4 Overheads

There are two major sources of overheads in programs constructed by Synergy. The first are discrete, non-fundamental overheads resulting from how programs are virtualized in hardware in the Synergy prototype. Implementing the semantics of the original program with the ability to pause execution in the middle of a virtual clock cycle involves toggling the virtual clock variable, evaluating relevant program logic, and latching variable assignments. When these are done in separate hardware cycles, there is a minimum 3× performance overhead. This is an artifact of our implementation rather than a fundamental requirement and can be improved with future work on target-specific backends.

The second source of overheads comes from the state access and execution control logic added by Synergy, which has a less-obvious impact on designs targeting FPGA hardware. To evaluate these overheads, we compile our benchmarks under a number of different conditions and measure the resource usage and achieved device frequency. We perform these compilations using Vivado on F1, target AmorphOS's maximum frequency (250 MHz), and use the reported delay to determine the frequency achieved.

As a baseline, we compile our benchmarks natively on AmorphOS, providing an upper bound on resource and frequency overheads. We also simulate a Cascade on AmorphOS baseline by compiling our benchmarks without system tasks, which avoids overheads introduced by our new state machine transformations. Finally, we modified our benchmarks to implement the quiescence protocol, allowing us to estimate the savings of exposing reconfiguration to developers and establishing a lower bound on state access overhead. Due to the complexity of fully rewriting our benchmarks for these cases, we only focus on replicating overheads and not functionality.

Figures 13 and 14 show Synergy's FF and LUT usage is generally $2-4\times$ and $1-6\times$ native, respectively. For `adpcm` and `mips32`, the results exceed the height of the graph and have been labeled with the appropriate values. These two benchmarks are exceptional due to their use of large on-chip RAMs, which Vivado implements using FFs under Synergy instead of LUTRAMs. Creating RAMs out of FFs can also require additional LUTs to implement muxing logic. The `adpcm`* and `mips32`* results compare against AmorphOS using FFs for RAMs and show that Synergy's overheads are resonable under these conditions. Future work should enable state access transformations that preserve the memories in user designs, which would eliminate this problem. Overall, we find that Synergy's overheads are similar to Cascade's and that using quiescence annotations can provide savings of up to ~2×.

Figure 15 shows that Synergy does not reduce the design's operating frequency in most cases. However, `adpcm` is an exception, likely due to its use of system tasks from inside its complex control

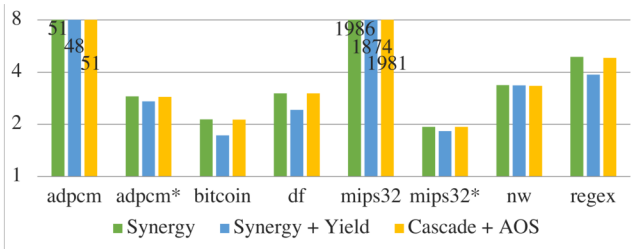

Figure 13: FF usage normalized to that of AmorphOS.

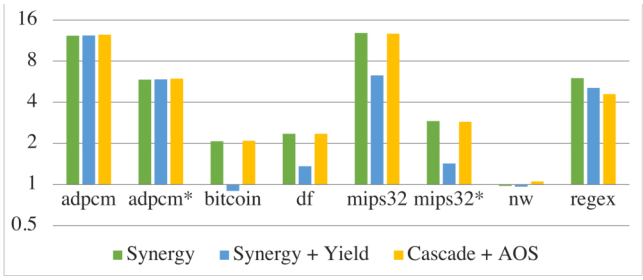

Figure 14: LUT usage normalized to that of AmorphOS.

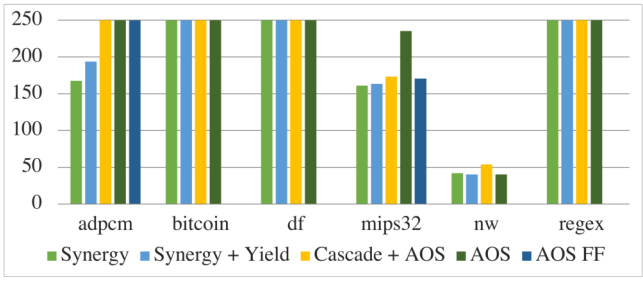

Figure 15: Design frequency achieved in MHz.

logic, which makes execution control much more expensive to implement. We see that Synergy's frequency overhead for `mips32` is almost entirely due to forcing the use of FFs to implement RAMs. When normalized against AmorphOS using FFs (AOS FF), Synergy was less than 6% slower despite supporting full state capture. We find that for `nw`, Synergy and the Cascade baseline achieve higher frequencies than native, likely due to `nw`'s complexity creating a higher-than-normal volatility in compiler outcomes. When combined with the previous 3× overhead, we find that Synergy's overall execution overhead is within 3 – 4× that of native.

While not shown in the data above, we tried compiling `adpcm` and `nw` with an anti-congestion strategy to see if it would help with their complex designs. We found that this improved both their frequencies under Synergy by 47%. With quiescence annotations, `adpcm` still improved 23% and `nw`, 37%. Applying the same strategy to `nw` under AOS only gave an improvement of 26%. This indicates that optimizing compiler strategies could be a great avenue for offsetting the costs of code transformations.

7 LIMITATIONS

Source Code. Synergy guarantees isolation between runtime instances with respect to hardware execution. However information leak between distrustful parties through source code and side channels is still possible. Leaks through sharing source could be mitigated with cryptographically secure channels, but Synergy would still function as a trusted party.

Compilation Cache. Synergy's backends rely on compilation caches to reduce overhead in production environments, as the alternative would be to transition back to software and wait through recompilation on virtualization events. Synergy uses techniques such as deterministic code generation to increase cache hit rates. As more applications use FPGAs, cache hit rates may drop and symmetry-breaking or speculative compilation may be needed to compensate.

8 RELATED WORK

FPGA OS and Virtualization. Primitives for FPGAs include sharing FPGA fabric [9, 14, 26, 50, 51, 93], spatial multiplexing [15, 28, 84, 91], context switch [59, 77], memory virtualization [1, 18, 62, 96], relocation [40], preemption [60], and interleaved hardware-software task execution [8, 30, 84, 91]. Core techniques include virtualizing FPGA fabric, including regions [71], tasks [73], processing elements [21], IPC-like communication primitives [66], and abstraction layers/overlays [7, 33, 48, 49, 85]

Extending OS abstractions to FPGAs is an area of active research. ReconOS [62] extends eCos [22] with *hardware threads* similar to Hthreads [70]. Borph [81, 82] proposes a *hardware process* abstraction. Previous multi-application FPGA sharing proposals [15, 37, 92, 94] restrict programming models or fail to provide isolation. OS primitives have been combined to form OSes for FPGAs [31, 62, 81, 82] as well as FPGA hypervisors [21, 66, 71, 73]. Chen et al. explore virtualization challenges when FPGAs are a shared resource [14]; AmorphOS [46] provides an OS-level management layer to concurrently share FPGAs among mutually distrustful processes. ViTAL [99] exposes a single-FPGA abstraction for scale-out acceleration over multiple FPGAs; unlike Synergy, it exposes a homogeneous abstraction of the hardware to enable offline compilation. The Optimus [63] hypervisor supports spatial and temporal sharing of FPGAs attached to the host memory bus, but does virtualize reconfiguration capabilities. Coyote [55] is a shell for FPGAs which supports both spatial and temporal multiplexing as well as communication and virtual memory management. While sharing goals with these systems, Synergy differs fundamentally from them by virtualizing FPGAs at the language level *in addition* to providing access to OS-managed resources.

FPGA Programmability and Compilation. HDLs, primarily Verilog [88] and VHDL [6], have served as the lowest abstraction level and least common denominator for programming FPGAs for decades. Improving FPGA programmability through higher level languages [3, 4, 19, 34, 47, 54, 56, 57, 65, 97], domain-specific languages [5, 17, 53, 57, 67, 75, 79, 83], or frameworks [4, 19, 47] is an active area of research, which while orthogonal, can greatly benefit from our contributions. AccelNet [27] supports fast in-network packet processing on top of Microsoft Catapult SmartNICs.

FlexNIC [44, 45] is a programmable networking device architecture to offload packet processing tasks. NICA [24] uses FPGA-based SmartNICs to accelerate network servers. Floem [72] is compiler to simplify offload development on SmartNICs.

FPGA compilers incur significant overhead. Synergy adapts many overhead-reduction techniques from the software domain such as eliminating redundant re-compilation [29, 74, 89], distributed build caching [2, 25, 32], and JIT compilation [87]. FPGA compilation can be further improved with a virtualization layer. Overlay-based virtualization [7, 42, 48, 49, 52, 95] abstracts away target-specific details and enables fast compilation and lower deployment latency. The approach reduces utilization and performance. Synergy and [78] work at this level, while AmorphOS [46] works at the application-OS boundary.

Hardware-Software Partitioning. The FPGA design cycle relies on developing a design in a high-fidelity simulator [35, 98] before compiling to hardware. Simulation incurs order of magnitude slowdowns compared to hardware execution, but is necessary for debugging. Many attempts have been made to bridge this performance gap, including systems which enable migration between different speed simulators [11], higher-level languages with partitioned runtime environments [4], and OS-managed communication channels [1]. Synergy builds on the approach taken by Cascade as it does not require explicit interface changes [1, 4] or hardware support [11].

9 CONCLUSION

FPGAs are emerging in data centers so techniques for virtualizing them are urgently needed to enable them as a practical resource for on-demand hardware acceleration. Synergy is a compiler/runtime solution that supports multi-tenancy and workload migration on hardware which is available *today*.

ACKNOWLEDGMENTS

We thank the PC and our shepherd Nate Foster for their insightful feedback. This research was supported by NSF grants CNS-1846169 and CNS-2006943.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix documents the requirements and instructions for setting up Synergy and how to reproduce the results of the experiments presented in our ASPLOS'21 paper. Code for our various backends can be found at:

- AWS F1: https://github.com/JoshuaLandgraf/cascade
- SW, DE10-Nano: https://github.com/eschkufz/cascade

A.2 Artifact Checklist

Checklist details: https://ctuning.org/ae/checklist.html

- **Algorithm**: Hardware-accelerated virtualized Verilog runtime
- **Program**: Assorted Verilog programs provided in our repos
- **Compilation**: GCC on Linux, Clang on macOS, AWS FPGA Dev AMI (includes Vivado) for AWS F1, Quartus Lite for DE10-Nano backend
- **Binary**: Our software is compiled from source. Vivado and Quartus binaries are provided through Amazon and Intel, respectively.
- **Data set**: Example data files provided with benchmarks
- **Run-time environment**: AWS F1 backend runs on AWS FPGA Dev AMI 1.7. DE10-Nano backend runs on Ubuntu 20 (VMs supported). SW backend can run on Ubuntu 20 or macOS 10.15.
- **Hardware**: AWS F1 instance or DE10-Nano kit.
- **Run-time state**: FPGA bitstreams cached for use in repeat execution.
- **Execution**: Benchmarks can run for minutes on hardware backends. Initial Quartus builds take ~20 minutes each; Vivado builds take ~2 hours, but large, timing-contrained builds can take several times that.
- **Metrics**: Synergy can profile virtual application frequency.
- **Output**: Profiling data is output to the console or a log file.
- **Experiments**: We provide a guide for environment setup, software installation, and experimental methodology for F1 instances.
- **How much disk space required?**: Our own software uses ~100MB. AWS FPGA Dev AMI uses 75GB with builds using ~1GB each. Quartus Lite uses 15GB with builds using 100s of MB each.
- **How much time is needed to prepare workflow?**: Software can be set up on Ubuntu or macOS in ~10 minutes. Setting up the AWS F1 and DE10-Nano environments can take 1-3 hours.
- **How much time is needed to complete experiments?**: Our F1 experiments can take 2 days, mostly for performing FPGA builds. Our DE10 experiments could take 2 hours.
- **Publicly available?**: Yes.
- **Code licenses?**: BSD-2.
- **Workflow framework used?**: Scripting via our software's library interface is supported.

A.3 Description

A.3.1 How to Access. Our code can be obtained by cloning the repositories linked in the abstract. The AWS F1 repo contains a preview of our F1 backend, which currently requires a manual setup process. The main repo is recommended for evaluating all other backends.

A.3.2 Hardware Dependencies. The AWS F1 backend requires an AWS f1.2xlarge instance, or f1.4xlarge for FPGA migration experiments. The DE10-Nano backend requires a DE10-Nano kit, available at http://de10-nano.terasic.com.

A.3.3 Software Dependencies. The AWS F1 backend requires the AWS FPGA Developer AMI, which includes all proprietary software needed. The DE10-Nano backend requires Intel's Quartus Lite software, which may require making a free account with Intel. All other dependencies are open-source and covered in our installation guides.

A.4 Installation

A thorough setup guide is available for the F1 backend: https://github.com/JoshuaLandgraf/cascade/blob/artifact/ARTIFACT.md. Otherwise, the README covers installation on Ubuntu and macOS, obtaining Intel's Quartus software, and setting up a DE10-Nano: https://github.com/eschkufz/cascade/blob/master/README.md.

A.5 Experiment Workflow

Our primary experiments consist of running several benchmarks on Synergy and demonstrating the ability to save/restore state, migrate execution across FPGAs, and share hardware resources. This can be accomplished from the command line, with directives specified dynamically through the REPL, or through scripting via the C++ library interface. Synergy tracks when these directives are executed as well as the virtual application's frequency and can print this information to the console or log it to a file.

A.6 Evaluation and Expected Results

In order to help simplify the process of running experiments, we provide several C++ programs that use Synergy's library interface to automate

experiment execution. These can be found in the artifact branch of the AWS F1 repo, in the experiments folder. The process of building and running these experiments is documented at https://github.com/JoshuaLandgraf/cascade/blob/artifact/experiments/README.md. We currently provide two versions of the experiments, one for native F1 execution, and one for software simulation. The expected results are displayed in this paper and can be compared with the performance data obtained by running the experiments.

A.7 Experiment Customization

Since Synergy is a runtime, it enables a wide variety of experiments beyond those shown in the paper. It is especially simple to look over the experiments provided and tweak them to run for longer, use different or more benchmarks, or execute more complex sequences of operations. Many of our benchmarks already have different top-level files that tweak the inputs, actions performed, or number of execution iterations. Users can also modify the provided benchmarks or run their own Verilog programs on Synergy.

REFERENCES

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/1950413.1950421

[2] K Aehlig et al. 2016. *Bazel: Correct, reproducible, fast builds for everyone.* https://bazel.io

[3] Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, Ron Sass, and David Andrews. 2006. Enabling a Uniform Programming Model across the Software/Hardware Boundary. FCCM 06.

[4] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. 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* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 89–108. https://doi.org/10.1145/1869459.1869469

[5] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. 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*. 1216–1225. https://doi.org/10.1145/2228360.2228584

[6] Jayaram Bhasker. 1999. *A VHDL primer*. Prentice-Hall.

[7] Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 93–96.

[8] Gordon J. Brebner. 1996. 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 (FPL '96)*. Springer-Verlag, London, UK, UK, 327–336. http://dl.acm.org/citation.cfm?id=647923.741195

[9] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. 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 (FCCM '14)*. IEEE Computer Society, Washington, DC, USA, 109–116.

[10] Stuart Byma, Naif Tarafdar, Talia Xu, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2015. Expanding OpenFlow Capabilities with Virtualized Reconfigurable Hardware. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. ACM, New York, NY, USA, 94–97. https://doi.org/10.1145/2684746.2689086

[11] Cadence. 2015. Cadence Palladium Z1 Enterprise Emulation Platform. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/palladium-z1-ds.pdf. (Accessed February 2020).

[12] Jared Casper and Kunle Olukotun. 2014. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2554688.2554787

[13] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture, In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture. https://www.microsoft.com/en-us/research/publication/configurable-cloud-acceleration/

[14] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (Cagliari, Italy) *(CF '14)*. ACM, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/2597917.2597929

[15] Liang Chen, Thomas Marconi, and Tulika Mitra. 2012. Online Scheduling for Multi-core Shared Reconfigurable Fabric. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Dresden, Germany) *(DATE '12)*. EDA Consortium, San Jose, CA, USA, 582–585. http://dl.acm.org/citation.cfm?id=2492708.2492853

[16] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. IEEE. https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/

[17] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *40th International Symposium on Computer Architecture*. ACM. http://research.microsoft.com/apps/pubs/default.aspx?id=198052

[18] Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An In-fabric Memory Architecture for FPGA-based Computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/1950413.1950435

[19] Philippe Coussy and Adam Morawiec. 2008. *High-level synthesis: from algorithm to digital circuit.* Springer Science & Business Media.

[20] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. 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* (Monterey, California, USA) *(FPGA '16)*. ACM, New York, NY, USA, 105–110. https://doi.org/10.1145/2847263.2847339

[21] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. 2006. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems* 30, 6 (2006), 334–354. https://doi.org/10.1016/j.micpro.2006.02.009

[22] Alexander Domahidi, Eric Chu, and Stephen Boyd. 2013. ECOS: An SOCP solver for embedded systems. In *Control Conference (ECC), 2013 European*. IEEE, 3071–3076.

[23] Amazon EC2. 2017. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[24] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. 345–362. https://www.usenix.org/conference/atc19/presentation/eran

[25] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 11–20.

[26] Suhaib A. Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom) (CLOUDCOM '15)*. IEEE Computer Society, Washington, DC, USA, 430–435.

[27] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, Berkeley, CA, USA, 51–64. http://dl.acm.org/citation.cfm?id=3307441.3307446

[28] W. Fu and K. Compton. 2008. Scheduling Intervals for Reconfigurable Computing. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*. 87–96. https://doi.org/10.1109/FCCM.2008.48

[29] K Funk et al. 2016. *icecream.* https://github.com/icecc/icecream

[30] Ivan Gonzalez, Sergio Lopez-Buedo, Gustavo Sutter, Diego Sanchez-Roman, Francisco J. Gomez-Arribas, and Javier Aracil. 2012. Virtualization of Reconfigurable Coprocessors in HPRC Systems with Multicore Architecture. *J. Syst. Archit.* 58, 6-7 (June 2012), 247–256. https://doi.org/10.1016/j.sysarc.2012.03.002

[31] B. K. Hamilton, M. Inggs, and H. K. H. So. 2014. 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*. 240–240. https://doi.org/10.1109/FCCM.2014.75

[32] Allan Heydon, Timothy Mann, Roy Levin, and Yuan Yu. 2006. *Software Configuration Management Using Vesta.* Springer.

[33] Chun-Hsian Huang and Pao-Ann Hsiung. 2009. Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems. *IEEE Embed. Syst. Lett.* 1, 1 (May 2009), 19–23. https://doi.org/10.1109/LES.2009.2028039

[34] SRC Computers Inc. 2006. Carte Programming Environment. http://www.srccomp.com/SoftwareElements.htm

[35] Intel. 2018. *Intel Quartus Prime Software.* https://www.altera.com/products/design-software/fpga-design/quartus-prime/download.html

[36] Intel. 2020. Cyclone V Device Handbook. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v2.pdf.

[37] Aws Ismail and Lesley Shannon. 2011. 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 (FCCM '11).* IEEE Computer Society, Washington, DC, USA, 170–177. https://doi.org/10.1109/FCCM.2011.48

[38] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16).* USENIX Association, Berkeley, CA, USA, 425–438. http://dl.acm.org/citation.cfm?id=2930611.2930639

[39] Alexander Kaganov, Asif Lakhany, and Paul Chow. 2011. FPGA Acceleration of MultiFactor CDO Pricing. *ACM Trans. Reconfigurable Technol. Syst.* 4, 2, Article 20 (May 2011), 17 pages. https://doi.org/10.1145/1968502.1968511

[40] H. Kalte and M. Porrmann. 2005. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on.* 223–228. https://doi.org/10.1109/FPL.2005.1515726

[41] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12).* ACM, New York, NY, USA, 295–308. https://doi.org/10.1145/2168836.2168866

[42] Nachiket Kapre and Jan Gray. 2015. Hoplite: Building austere overlay NoCs for FPGAs. In *FPL.* IEEE, 1–8.

[43] Kaan Kara and Gustavo Alonso. 2016. 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.* 1–4. https://doi.org/10.1109/FPL.2016.7577353

[44] Antoine Kaufmann, Simon Peter, Thomas Anderson, and Arvind Krishnamurthy. 2015. FlexNIC: Rethinking Network DMA. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland) *(HOTOS'15).* USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=2831090.2831097

[45] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS).* 15 pages. https://doi.org/10.1145/2872362.2872367

[46] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18).* 107–127.

[47] Khronos Group 2009. *The OpenCL Specification, Version 1.0.* Khronos Group. http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf

[48] Robert Kirchgessner, Alan D. George, and Greg Stitt. 2015. Low-Overhead FPGA Middleware for Application Portability and Productivity. *ACM Trans. Reconfigurable Technol. Syst.* 8, 4, Article 21 (Sept. 2015), 22 pages. https://doi.org/10.1145/2746404

[49] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. 2012. VirtualRC: A Virtual FPGA Platform for Applications and Tools Portability. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '12).* ACM, New York, NY, USA, 205–208. https://doi.org/10.1145/2145694.2145728

[50] O. Knodel, P. Lehmann, and R. G. Spallek. 2016. RC3E: Reconfigurable Accelerators in Data Centres and Their Provision by Adapted Service Models. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD).* 19–26. https://doi.org/10.1109/CLOUD.2016.0013

[51] Oliver Knodel and Rainer G. Spallek. 2015. RC3E: Provision and Management of Reconfigurable Hardware Accelerators in a Cloud Environment. *CoRR* abs/1508.06843 (2015). arXiv:1508.06843

[52] Dirk Koch, Christian Beckhoff, and Guy G. F. Lemieux. 2013. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL.* IEEE, 1–8.

[53] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16).* IEEE Press, Piscataway, NJ, USA, 115–127. https://doi.org/10.1109/ISCA.2016.20

[54] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018).* ACM, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[55] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[56] James Lebak, Jeremy Kepner, Henry Hoffmann, and Edward Rutledge. 2005. Parallel VSIPL++: An open standard software library for high-performance parallel signal processing. *Proc. IEEE* 93, 2 (2005), 313–330.

[57] Ilia A. Lebedev, Christopher W. Fletcher, Shaoyi Cheng, James Martin, Austin Doupnik, Daniel Burke, Mingjie Lin, and John Wawrzynek. 2012. Exploring Many-Core Design Templates for FPGAs and ASICs. *Int. J. Reconfig. Comp.* 2012 (2012), 439141:1–439141:15. https://doi.org/10.1155/2012/439141

[58] Christian Leber, Benjamin Geib, and Heiner Litz. 2011. High Frequency Trading Acceleration Using FPGAs. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL '11).* IEEE Computer Society, Washington, DC, USA, 317–322. https://doi.org/10.1109/FPL.2011.64

[59] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai, and Chia-Chun Tsai. 2010. Hardware Context-Switch Methodology for Dynamically Partially Reconfigurable Systems. *J. Inf. Sci. Eng.* 26 (2010), 1289–1305.

[60] L. Levinson, R. Manner, M. Sessler, and H. Simmler. 2000. Preemptive multitasking on FPGAs. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on.* 301–302. https://doi.org/10.1109/FPGA.2000.903426

[61] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. 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* (Portland, Oregon) *(ISCA '15).* ACM, New York, NY, USA, 476–488. https://doi.org/10.1145/2749469.2750416

[62] Enno Lübbers and Marco Platzner. 2009. ReconOS: Multithreaded Programming for Reconfigurable Computers. *ACM Trans. Embed. Comput. Syst.* 9, 1, Article 8 (Oct. 2009), 33 pages. https://doi.org/10.1145/1596532.1596540

[63] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland).

[64] Microsoft. 2017. Microsoft Azure Goes Back To Rack Servers With Project Olympus. https://www.nextplatform.com/2016/11/01/microsoft-azure-goes-back-rack-servers-project-olympus/. (Accessed July 2018).

[65] Diba Mirza, Deeksha Dangwal, and Timothy Sherwood. 2019. PyRTL in Early Undergraduate Research. In *Proceedings of the Workshop on Computer Architecture Education* (Phoenix, AZ, USA) *(WCAE'19).* ACM, New York, NY, USA, Article 6, 8 pages. http://doi.acm.org/10.1145/3338698.3338890

[66] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 163–174. https://doi.org/10.1145/1168917.1168878

[67] Nicholas Moore, Albert Conti, Miriam Leeser, Benjamin Cordes, and Laurie Smith King. 2007. An extensible framework for application portability between reconfigurable supercomputing architectures.

[68] NEC. [n.d.]. neoface | NEC Today. http://nectoday.com/tag/neoface/. (Accessed April 2019).

[69] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16).* ACM, New York, NY, USA, 111–117. https://doi.org/10.1145/2847263.2847337

[70] Wesley Peck, Erik K. Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David L. Andrews. 2006. Hthreads: A Computational Model for Reconfigurable Devices.. In *FPL.* IEEE, 1–4.

[71] K. Dang Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell. 2013. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on.* 219–226. https://doi.org/10.1109/ASAP.2013.6567578

[72] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodík, and Thomas E. Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.* 663–679. https://www.usenix.org/conference/osdi18/presentation/phothilimthana

[73] Christian Plessl and Marco Platzner. 2005. 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. IEEE, 213–218.

[74] M Pool et al. 2016. *distcc: A free distributed C/C++ compiler system.* https://github.com/distcc/distcc

[75] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 389–402. https://doi.org/10.1145/3079856.3080256

[76] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *41st Annual International Symposium on Computer Architecture (ISCA)*. http://research.microsoft.com/apps/pubs/default.aspx?id=212001

[77] Kyle Rupnow, Wenyin Fu, and Katherine Compton. 2009. 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*. 63–70. https://doi.org/10.1109/FCCM.2009.30

[78] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 271–286.

[79] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ning-Yi Xu, and Huazhong Yang. 2010. FPMR: MapReduce framework on FPGA.. In *FPGA* (2010-03-01), Peter Y. K. Cheung and John Wawrzynek (Eds.). ACM, 93–102. http://dblp.uni-trier.de/db/conf/fpga/fpga2010.html#ShanWYWXY10

[80] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. 2016. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*.

[81] Hayden Kwok-Hay So and Robert Brodersen. 2008. A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH. *ACM Trans. Embed. Comput. Syst.* 7, 2, Article 14 (Jan. 2008), 28 pages. https://doi.org/10.1145/1331331.1331338

[82] Hayden Kwok-Hay So and Robert W. Brodersen. 2007. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.html

[83] Hayden Kwok-Hay So and John Wawrzynek. 2016. OLAF'16: Second International Workshop on Overlay Architectures for FPGAs. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/2847263.2847345

[84] C. Steiger, H. Walder, and M. Platzner. 2004. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Trans. Comput.* 53, 11 (Nov 2004), 1393–1407. https://doi.org/10.1109/TC.2004.99

[85] G. Stitt and J. Coole. 2011. Intermediate Fabrics: Virtual Architectures for Near-Instant FPGA Compilation. *IEEE Embedded Systems Letters* 3, 3 (Sept 2011), 81–84. https://doi.org/10.1109/LES.2011.2167713

[86] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. 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* (Monterey, California, USA) *(FPGA '16)*. ACM, New York, NY, USA, 16–25. https://doi.org/10.1145/2847263.2847276

[87] Lambert M. Surhone, Mariam T. Tennoe, and Susan F. Henssonow. 2010. *Node.Js*. Betascript Publishing, Mauritius.

[88] Donald Thomas and Philip Moorby. 2008. *The Verilog® Hardware Description Language*. Springer Science & Business Media.

[89] A Tridgell, J Rosdahl, et al. 2016. *ccache: A Fast C/C++ Compiler Cache*. https://ccache.samba.org

[90] A. Tsutsui, T. Miyazaki, K. Yamada, and N. Ohta. 1995. Special Purpose FPGA for High-speed Digital Telecommunication Systems. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors (ICCD '95)*. IEEE Computer Society, Washington, DC, USA, 486–491. http://dl.acm.org/citation.cfm?id=645463.653355

[91] G. Wassi, Mohamed El Amine Benkhelifa, G. Lawday, F. Verdier, and S. Garcia. 2014. Multi-shape tasks scheduling for online multitasking on FPGAs. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*. 1–7. https://doi.org/10.1109/ReCoSoC.2014.6861366

[92] Matthew A. Watkins and David H. Albonesi. 2010. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 497–508. https://doi.org/10.1109/MICRO.2010.15

[93] Jagath Weerasinghe, François Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. 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*. 1078–1086.

[94] Stephan Werner, Oliver Oey, Diana Göhringer, Michael Hübner, and Jürgen Becker. 2012. Virtualized On-chip Distributed Computing for Heterogeneous Reconfigurable Multi-core Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Dresden, Germany) *(DATE '12)*. EDA Consortium, San Jose, CA, USA, 280–283. http://dl.acm.org/citation.cfm?id=2492708.2492778

[95] Tobias Wiersema, Ame Bockhorn, and Marco Platzner. 2014. Embedding FPGA overlays into configurable Systems-on-Chip: ReconOS meets ZUMA. In *ReConFig*. IEEE, 1–6.

[96] Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, and George Constantinides. 2015. MATCHUP: Memory Abstractions for Heap Manipulating Programs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. ACM, New York, NY, USA, 136–145. https://doi.org/10.1145/2684746.2689073

[97] Xilinx. [n.d.]. *SDAccel Development Environment*. (Accessed on 7/17/2018).

[98] Xilinx. 2018. *Vivado Design Suite*. https://www.xilinx.com/products/design-tools/vivado.html

[99] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *ASPLOS 2020: Architectural Support for Programming Languages and Operating Systems*. ACM.

[100] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060