DOI:10.1145/3613903

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3655633

Compiler-Driven FPGA Virtualization with SYNERGY

By Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza

Abstract

FPGAs are increasingly common in modern applications, and cloud providers now support on-demand FPGA acceleration in datacenters. Applications in datacenters 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 Intel SoCs and Xilinx FPGAs on Amazon F1. The workloads require no modification, run within 3–4x of unvirtualized performance, and incur a modest increase in FPGA fabric usage.

1. INTRODUCTION

Field-programmable gate arrays (FPGAs) combine the functional efficiency of hardware with the programmability of software. FPGAs can exceed CPU performance by orders of magnitude[20] and offer lower cost and time to market than ASICs. FPGAs have become a compelling acceleration alternative for machine learning,[4] databases,[14] graph processing,[17] and communication.[8] In datacenters, FPGAs serve diverse hardware needs with a single technology. Amazon provides F1 instances with large FPGAs attached and Microsoft deploys FPGAs in their new datacenters.

Virtualization is fundamental to datacenters. 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 *multitenancy* (multiplexing multiple hardware programs on a single FPGA). Better virtualization support is required for FPGAs to become a mainstream accelerator technology.

The original version of this paper was published in the *Intern. Conf. on Architectural Support for Programming Languages and Operating Systems* (April 2021).

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 such as 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* rewrite 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 subprograms, 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 them 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 transparently

combine the sub-program representations from multiple applications into a single hardware program by 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 FPGA-OS proposals introduce interfaces for state capture for context switch.[9,16] 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 mechanisms, SYNERGY can dramatically reduce the overhead for context switch and migration.

Our SYNERGY prototype extends the Cascade[21] JIT compiler and composes it with the AmorphOS[9] FPGA OS. We demonstrate the ability to suspend and resume programs running on a cluster of Intel SoCs and Xilinx FPGAs running on Amazon's F1 cloud instances, to transition applications between them, 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, extending the Verilog language, or modifying the programs. We achieve performance within 3–4x of unvirtualized code with a reasonable fabric cost.

2. BACKGROUND

Verilog is one of two standard HDLs used to program FPGAs. VHDL 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*. Figure 1 shows an example Verilog module. 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 (for example, `&`), arithmetic (for example, `+`), and nested sub-modules (`sm`). The value of a wire is derived from its inputs (lines 5, 21), whereas a register is updated at discrete intervals (lines 6, 11, 13).

Verilog supports sequential and concurrent semantics. Continuous assignments (lines 5, 21) are scheduled when the value of their right-hand-side changes while procedural blocks (lines 9–19) are scheduled when their guard is satisfied (for example, `clock` changes from 0 to 1). Only a `begin/end` block guarantees sequential execution; statements in a `fork/join` block may be evaluated in any order. There are two types of assignments to registers: blocking (`=`) and non-blocking (`<=`). Blocking assignments execute immediately, whereas non-blocking assignments wait until all continuous assignments and control blocks are finished.

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

```
 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[31:0] 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 = 4[47:16] & 31'hf0f0f0f0;
22: endmodule
```

When used idiomatically, these semantics map directly onto hardware primitives: 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 and can be interleaved with any assignment in lines 10–14. Likewise, the first execution of lines 12 and 14 can be interleaved with the assignment on line 5. While the assignment on line 11 is visible immediately, the one on line 13 is only performed after every block and assignment has been scheduled.

2.1. Cascade

Cascade is the first JIT compiler for Verilog. Cascade parses and adds Verilog to a program one line at a time, with side effects appearing immediately. While JIT compilation is orthogonal to SYNERGY, Cascade's runtime techniques are a fundamental building block. Cascade transforms programs to 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 manages programs at module granularity. Its IR expresses a distributed system of Verilog *sub-programs*, each corresponding to a 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 that are replaced over time by high-performance FPGA-resident engines. The IR's constrained ABI enables engines to be relocated through messages sent over the runtime's data/control plane. `Get/set` messages read and write an engine's inputs, outputs, and program variables. `Evaluate/update` messages instruct an engine to run until no more continuous assigns or procedural blocks can be

scheduled, and to latch the result of non-blocking assignments, respectively.

Unsynthesizable traps are placed in a queue and evaluated between clock ticks, when the engine state has 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 enables hardware programs to scale dynamically in response to FPGA load and availability. AmorphOS introduces an FPGA process abstraction called *Morphlets*, which access OS-managed resources through a shell-like component called a hull. The hull acts as an isolation boundary and a compatibility layer, enabling AmorphOS to increase utilization by co-locating several Morphlets in a single reconfigurable *zone* without compromising security. AmorphOS leaves the problems of efficient context switch, over-subscription, and support for multiple FPGAs mostly unsolved by relying on a compilation cache and a programmer-exposed quiescence interface.

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, but difficult for programs that can execute unbounded sequences of instructions, such as a RISC core. 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`, which are necessary for supporting workload migration. We frame this discussion with a file IO case study.

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 file. The program opens the file (line 2) and, on every clock tick, attempts to read a 32-bit value (line 7). When the program reaches the end-of-file, it prints the sum and returns to the host (lines 8–10). Otherwise, it adds the value to the running sum and continues (line 12). 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

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

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

IO tasks introduce data-dependencies within a single clock-tick. The end-of-file check on line 8 depends on the result of the read operation on line 7, as does the assignment on line 12. Because these operations interact with the file system, we 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 co-routine style semantics. While a programmer could adopt this idiom, it would harm both readability and maintainability.

3.2. Scheduling Transformations

SYNERGY uses several transformations 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 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 (details to follow) as all of the conjuncts would otherwise be executed when the core is triggered. These transformations are sound, even for programs with multiple clock domains.

3.3. Control Transformations

Additional transformations 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 communicated through `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 pro-

gram (for example, __pos_x is true when a set message changes x from false to true). We also declare new variables (__state and __task) to track the 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 synthesizable statements, terminated by either 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 3 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 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 can take control, place its results (if any) in the appropriate hardware location, and yield 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. Collectively, these steps represent the compute portion of the evaluate and update requests required by the ABI.

Figure 3. The motivating example after modification to yield control to the runtime at the sub-clock-tick granularity.

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

3.5. Workload Migration

With these transformations, support for the $save and $restart system tasks is straightforward. Both can be materialized as traps into 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 prompts a sequence of set requests. Either statement can be triggered via normal program execution or an eval statement. Once a program's state is read out, it can be suspended, migrated to another machine, and resumed.

4. HYPERVISOR DESIGN

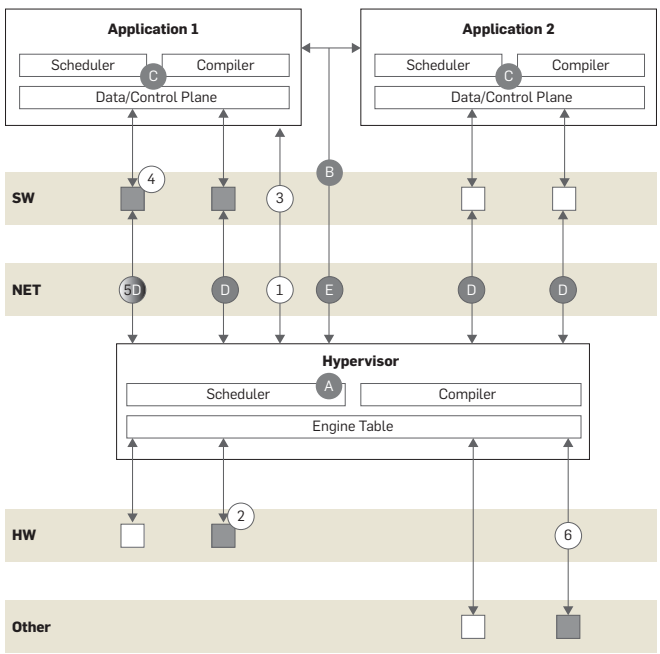
In this section, we describe SYNERGY's support for the two primary forms of hardware multiplexing: *spatial* (where two programs run simultaneously on the same fabric) and *temporal* (where two programs share resources using timeslice scheduling). SYNERGY provides an indirection layer that allows multiple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

4.1. Program Coalescing

Figure 4 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 code for a sub-program over the connection, where it is passed to the

Figure 4. The Synergy virtualization layer. The hypervisor combines sub-programs from multiple applications onto a single target (1–6). A handshake protocol establishes state-safe interrupts in the scheduler (A–E).



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 simply forwards requests across the network to the hypervisor (5) and blocks 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[21] 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, enabling near-native performance even for programs separated from the hypervisor by a network connection. In contrast, applications that invoke frequent ABI calls (for example, 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, destroying 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 4 shows the handshake protocol used to establish these invariants. Compilation requests are scheduled asynchronously (A), and run until they would do something destructive. The hypervisor then sends a request to every connected runtime instance (B) to schedule an interrupt between their logical clock-ticks when they are in a consistent state (C). The interrupt causes the instances to send `get` requests to Synergy (D) to save their program state. When they have finished, the instances send a reply indicating it is safe to reprogram the device (E) and block until they receive an acknowledgment. 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.

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 family of Intel devices that feature reprogrammable fabric and an ARM CPU. The core feature these targets share is support for Intel's Avalon interface for memory-mapped IO. This allows us to lower the transformations described in §3 onto a Verilog module that converts accesses on the Avalon interface into ABI requests.

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. 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. 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 10x more LUTs and operates 5x 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 (for example, during reconfiguration), allowing them to quiesce and back up their state accordingly. 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 `$yield` task, shown in Figure 5. Developers can assert `$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`.

Figure 5. The `$yield` task enables Synergy's quiescence interface. Volatile variables must be managed by the user.

```
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:     SubModule sm(clock);
8: endmodule
```

6. EVALUATION

We evaluated SYNERGY using a combination of Intel DE10 SoCs and Amazon F1 cloud instances. The DE10s consist of a Cyclone V device with an 800MHz dual core ARM processor, reprogrammable fabric of 110K LUTs, 50MHz clock, and 1GB 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 support multiple Xilinx UltraScale+ VU9Ps running at 250MHz and four 16GB 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 debugging. Benchmarks were compiled prior to running experiments to prime SYNERGY's bitstream caches. This is appropriate as SYNERGY's goal is to provide virtualization support for applications which have spent sufficient time in development to have converged on a stable design.

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

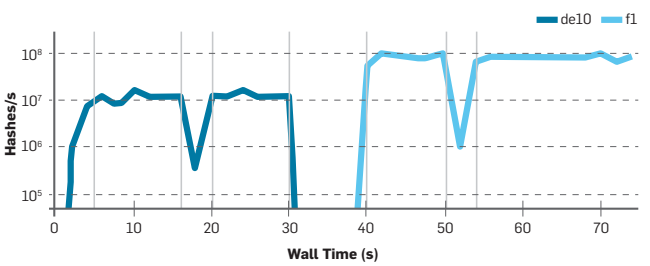
Name	Description
adpcm	Pulse-code modulation encoder/decoder
bitcoin	Bitcoin mining accelerator
df	Double-precision arithmetic circuits
mips32	Bubble-sort on a 32-bit MIPS processor
nw*	DNA sequence alignment
regex*	Streaming regular expression matcher

6.1. Workload Migration

Figure 6 plots `bitcoin`'s performance as it is moved between software and hardware on two different target architectures. This workflow is typical of suspend and resume style virtualization.

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

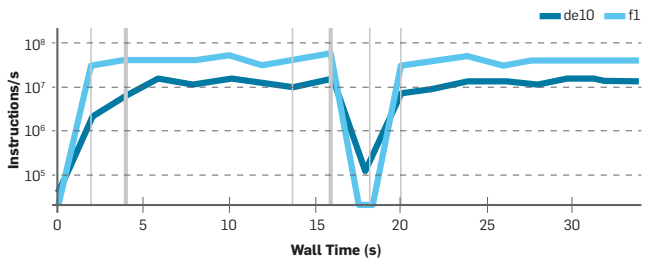
Figure 6. Suspend and Resume. `Bitcoin` is executed on a DE10 target, suspended, saved, and resumed on `F1`.



throughput (83M), but suffers from higher performance degradation during the `restart` as it takes longer to reconfigure.

Figure 7 plots the performance of a 32-bit MIPS processor consisting of registers, a datapath, and on-chip memory. The CPU repeatedly randomizes and sorts an in-memory array, with execution transitioning between two FPGAs. This workload is typical of long-running batch computations which are coalesced to improve datacenter 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. 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 (the state of a MIPS processor consists of its registers, data memory, and instruction memory).

Figure 7. Hardware Migration. `Mips32` begins execution on one target and is migrated mid-execution to another.

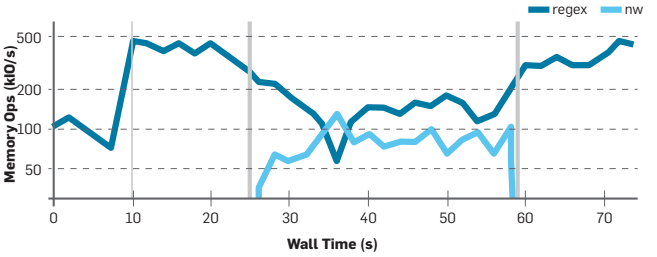


6.2. Multitenancy

Figure 8 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 use of round-rob-

Figure 8. Temporal Multiplexing. `Regex` and `nw` are timeslice scheduled to resolve contention on off-device `IO`.



in 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).

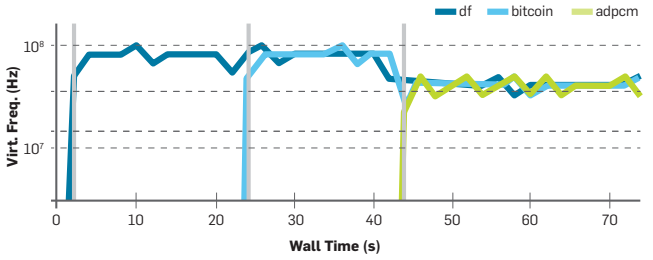
At (t = 60), the sequence aligner completes execution, and the throughput for the matcher returns to its peak value shortly thereafter. Compared to previous examples, the time required to transition between performance regimes is slightly more pronounced. This is due to Synergy's use of adaptive refinement[21] 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 9 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. 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[21] of 83MHz. 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 250MHz. To accommodate all three applications, the global clock is set to 125MHz, reducing their virtual clock frequencies to 41MHz. 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 7) 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

Figure 9. Spatial Multiplexing. `Bitcoin`, `df`, and `adpcm` are co-scheduled on one device without contention.



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 ma jor sources of overheads in programs constructed by Synergy. The first are discrete, nonfundamental 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 3x performance overhead. This not 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. 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.

We find that Synergy's FF and LUT usage is generally 2–4x and 1–6x that of native, respectively (figures shown in the full paper). Overall, Synergy's overheads are similar to Cascade's, with quiescence annotations providing a savings of up to ~2*x*.

Figure 10. Design frequency achieved in MHz.

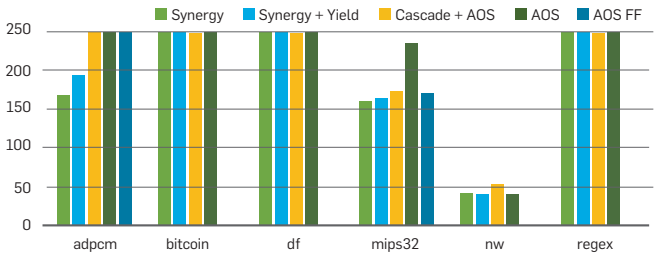


Figure 10 shows that Synergy does not reduce the design's operating frequency in most cases. `adpcm` is an exception, likely due to its use of system tasks from inside complex control logic. Synergy's frequency overhead for `mips32` is almost entirely due to forcing the use of FFs to implement RAMs, which is not a fundamental limitation. Compared to AmorphOS using FFs (AOS FF), Synergy was less than 6% slower despite supporting full state capture. When combined with the previous 3x overhead, we find that Synergy's overall execution overhead is within 3–4x that of native.

7. RELATED WORK

Primitives for FPGAs include sharing FPGA fabric,[1,2,6,11,12,23] spatial multiplexing,[3,22] memory virtualization,[15,24] preemption,[16] and interleaved hardwaresoftware task execution.[22] Core techniques include virtualizing FPGA fabric, including regions[19] and abstraction layers/overlays[10] Extending OS abstractions to FPGAs is an area of active research. ReconOS[15] extends eCos[5] with *hardware threads* similar to Hthreads.[18] Previous multi-application FPGA sharing proposals[3] restrict programming models or fail to provide isolation. OS primitives have been combined to form OSes for FPGAs[7,15] as well as FPGA hypervisors.[19] Chen et al. explore virtualization challenges when FPGAs are a shared resource;[2] AmorphOS[9] provides an OS-level management layer to concurrently share FPGAs among mutually distrustful processes. ViTAL[25] 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[16] hypervisor supports spatial and temporal sharing of FPGAs attached to the host memory bus, but does virtualize reconfiguration capabilities. Coyote[13] 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.

8. CONCLUSION

FPGAs are emerging in datacenters 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*.

9. ACKNOWLEDGMENTS

This research was supported by NSF grants CNS-1846169 and CNS-2006943, and U.S. Department of Energy, National Nuclear Security Administration Award Number DENA0003969. C

References

1. Byma, S. et al. FPGAs in the cloud: Booting virtualized hardware accelerators with openstack. In *Proceedings of the 2014 IEEE 22nd Intern. Symp. on Field-Programmable Custom Computing Machines, FCCM '14*. IEEE Computer Society, Washington, DC, USA, 109–116.
2. Chen, F. et al. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conf. on Computing Frontiers, CF '14*. ACM, New York, NY, USA, (2014), 3:1–3:10.
3. Chen, L., Marconi, T., and Mitra, T. Online scheduling for multi-core shared reconfigurable fabric. In *Proceedings of the Conf. on Design, Automation and Test in Europe, DATE '12*. EDA Consortium, San Jose, CA, USA, (2012), 582–585.
4. Chung, E. et al. *Serving DNNs in Real*

Time at Datacenter Scale with Project Brainwave. IEEE, (March 2018).
5. Domahidi, A., Chu, E., and Boyd, S. ECOS: An SOCP solver for embedded systems. In *Control Conf. (ECC) European*. IEEE, (2013), 3071–3076.
6. Fahmy, S.A., Vipin, K., and Shreejith, S. Virtualized FPGA accelerators for efficient cloud computing. In *Proceedings of the 2015 IEEE 7th Intern. Conf. on Cloud Computing Technology and Science (CloudCom), CLOUDCOM '15*. IEEE Computer Society, Washington, D.C., USA, 430–435
7. 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 Intern. Symp., 240–240*.
8. Kapitza, R. et al. CheapBFT: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conf. on Computer Systems, EuroSys '12*. ACM, New York, NY, USA, (2012), 295–308.
9. Khawaja, A. et al. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th [USENIX] Symp. on Operating Systems Design and Implementation ([OSDI]*, 2018, 107–127.
10. 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.
11. Knodel, O., Lehmann, P., and Spallek, R.G. RC3E: Reconfigurable accelerators in data centres and their provision by adapted service models. In *2016 IEEE 9th Intern. Conf. on Cloud Computing (CLOUD)*, 19–26.
12. Knodel, O. and Spallek, R.G. Rc3e: Provision and management of reconfigurable hardware accelerators in a cloud environment. *CoRR, abs/1508.06843*, (2015).
13. Korolija, D., Roscoe, T., and Alonso, G. Do OS abstractions make sense on FPGAs? In *14th USENIX Symp. on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, (Nov. 2020), 991–1010.
14. Li, S. et al. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual Intern. Symp. on Computer Architecture, ISCA '15*. ACM, New York, NY, USA, (2015), 476–488.
15. Lübbers, E. and Platzner, M. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst. 9*, 1, Oct. 2009, 8:1–8:33.
16. Ma, J. et al. A hypervisor for shared-memory FPGA platforms. In *Proceedings of the 25th Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2020.
17. Oguntebi, T. and Olukotun, K. GraphOps: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA Intern. Symp. on Field-Programmable Gate Arrays, FPGA '16*. ACM, New York, NY, USA, 111–117.
18. Peck, W. et al. Hthreads: A computational model for reconfigurable devices. In *FPL*. IEEE, (2006), 1–4.
19. Pham, K.D. et al. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th Intern. Conf. on, pages 219–226*.
20. Putnam, A. et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual Intern. Symp. on Computer Architecture (ISCA)*, (June 2014).
21. Schkufza, E., Wei, M., and Rossbach, C.J. Just-in-time compilation for verilog: A new technique for improving the FPGA programming experience. In *Proceedings of the 24th Intern. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA*, (Apr. 2019), 271–286.
22. Wassi, G. et al. Multi-shape tasks scheduling for online multitasking on FPGAs. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th Intern. Symp. on, pages 1–7*, (May 2014).
23. Weerasinghe, J., Abel, F., Hagleitner, C., and Herkersdorf, A. Enabling FPGAs in hyperscale data centers. In *2015 IEEE 12th Intern. Conf. on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intern. Conf. on Autonomic and Trusted Computing and 2015 IEEE 15th Intern. Conf. on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), Beijing, China, (August* 2015), 1078–1086.
24. Winterstein, F. et al. Matchup: Memory abstractions for heap manipulating programs. In *Proceedings of the 2015 ACM/SIGDA Intern. Symp. on Field-Programmable Gate Arrays, FPGA '15*. ACM, New York, NY, USA, (2015), 136–145.
25. Zha, Y. and Li, J. Virtualizing FPGAs in the cloud. In *ASPLOS 2020: Architectural Support for Programming Languages and Operating Systems*. ACM, (2020).

Joshua Landgraf (jland@cs.utexas.edu), University of Texas at Austin, TX, USA.

Tiffany Yang (tiffanyyang@utexas.edu), University of Texas at Austin, TX, USA.

Will Lin (w5lin@ucsd.edu), University of California, San Diego, CA, USA. This work was done while he was at University of Texas at Austin.

Christopher J. Rossbach (rossbach@cs.utexas.edu), University of Texas at Austin and VMware Research Group, Austin, TX, USA.

Eric Schkufza (eric.schkufza@gmail.com), Graft, Inc., San Francisco, CA, USA. This work was done while he was at VMWare Research Group.