Leveraging Latency-Insensitivity to Ease Multiple FPGA Design

Kermin Fleming[¶] Michael Adler[†] Michael Pellauer[†] Angshuman Parashar[†]
Arvind[¶] Joel Emer[†¶]

[†]Intel Corporation
VSSAD Group
{michael.adler, michael.i.pellauer,
angshuman.parashar, joel.emer}
@intel.com

[¶]Massachusetts Institute of Technology
Computer Science and A.I. Laboratory
{kfleming, arvind,
emer}
@csail.mit.edu

ABSTRACT

Traditionally, hardware designs partitioned across multiple FPGAs have had low performance due to the inefficiency of maintaining cycle-by-cycle timing among discrete FPGAs. In this paper, we present a mechanism by which complex designs may be efficiently and automatically partitioned among multiple FPGAs using explicitly programmed latency-insensitive links. We describe the automatic synthesis of an area efficient, high performance network for routing these inter-FPGA links. By mapping a diverse set of large research prototypes onto a multiple FPGA platform, we demonstrate that our tool obtains significant gains in design feasibility, compilation time, and even wall-clock performance.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Automatic Synthesis

General Terms

Design, Performance

Keywords

FPGA, compiler, design automation, high-level synthesis, switch architecture, DSP, programming languages

1. INTRODUCTION

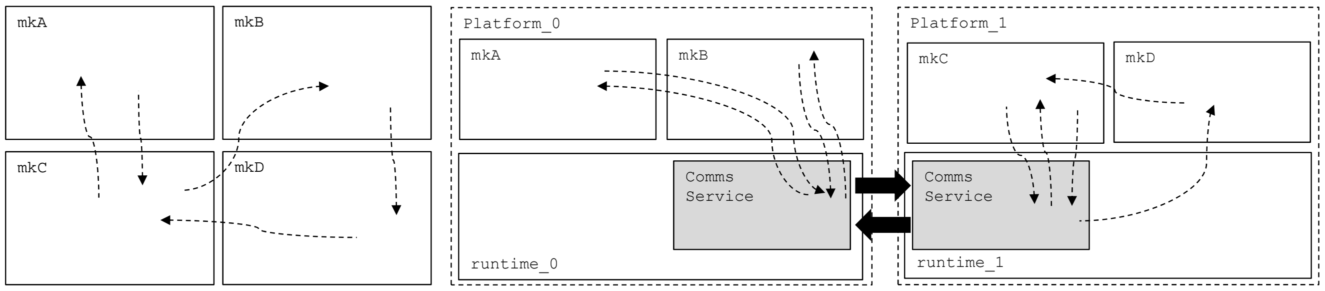
FPGAs are an extremely valuable substrate for prototyping and modeling hardware systems. However, some interesting designs may not fit in the limited area of a single FPGA. If a design cannot fit onto a given FPGA, the designer is faced with a handful of choices. The designer may use a larger single FPGA or refine the design to reduce area, neither of which may be possible. A third possibility is to partition the design among multiple FPGAs. This option is typically feasible from an implementation perspective, but has some serious drawbacks. Manual partitioning may obtain high performance, but represents a time consuming design effort. Tool-based partitioning, while automatic, may suffer performance degradation.

Consider a processor model manually partitioned across two FPGAs in which there are two channels sharing a multiplexed physical link between the FPGAs. One channel is information from the processor decode stage. It has a narrow bit-width, but occurs in nearly every processor cycle. The other channel is the memory interface between the last-level cache and backing main memory. This link is exceptionally wide, with perhaps as many as 1000 bits. However, because programs generally have good locality this link is infrequently used. Exploiting this knowledge produces a high performance partitioned design: data for each link is sent as it becomes available and only when it is available. This implementation recognizes two high-level properties of the underlying design; first, although physical wires are driven to specific values every cycle, not all of those values impact behavior, and second, that links may have different semantic properties, such as priority.

Automatic partitioning tools avoid the engineering overhead of manual partitioning at the cost of throughput. In the previous example the tool needs to automatically understand when the last-level cache is making requests in order to achieve high throughput. If the tool cannot derive this high level meaning, then it must conservatively transport all or nearly all values between FPGAs on each cycle to maintain functional correctness. Because extracting this high-level meaning is difficult, existing tools take the conservative approach in partitioning. Thus, pin bandwidth, serialization, and latency become major sources of performance degradation in partitioned designs, even if the values transported between FPGAs ultimately do not impact design behavior on the majority of cycles. The difficulty in extracting high-level knowledge from RTL lies in automatically differentiating the cycle-by-cycle timing behavior of RTL from the *functional* behavior of a design.

In this paper, we explore the application of latency insensitive design [3] to multiple FPGA implementation. We view hardware designs as sets of modules connected by latency insensitive FIFOs, as shown in Figure 1(a). Because inter-module communication is restricted to latency-insensitive FIFOs, we have broad freedom in implementing the network between modules. We leverage this freedom in our compiler to automatically generate design implementations that span multiple FPGAs. A key technical contribution of this paper is the automatic synthesis of a high-performance and deadlock-free network optimized for design-specific, latency-insensitive communications.

By partitioning designs among multiple FPGAs at latency insensitive FIFOs, we gain efficiency over traditional tools in two ways. First, only data explicitly enqueued into the FIFOs needs to be transported. Second, we have more options in transporting data over multiplexed physical links. By compiling several large research prototypes to a multiple FPGA platform, we will demonstrate that our tool obtains significant gains in design feasibility, compilation



(a) Application (b) Partitioned Implementation of 1(a)

Figure 1: A sample application and its mapping to a two FPGA platforms. Links to modules on the same FPGA are directly tied together by a FIFO, while inter-FPGA links are tied to a synthesized communications complex produced during compilation.

time, and, in some cases, wall-clock performance as compared to conventional single FPGA implementations.

2. BACKGROUND

2.1 Model of Computation

Recently, a number of highly modular research prototypes [15] [6] have been developed using latency-insensitive design [5]. In latency-insensitive design, the goal is to maintain the *functional* correctness of the design in response to variations in data availability. Latency-insensitive designs are typically composed of modules connected by FIFO links; modules are designed to compute when data is available, as opposed to when some clock ticks. Latency-insensitive design offers a number of practical benefits, including improved modularity and simplified design exploration.

The usual framing of the latency-insensitive design focuses on the properties of and composition of modules, but we observe that an implication of this style of design is that the FIFOs connecting the modules may take an arbitrarily long time to transport data without affecting the functional correctness of the original design. As a result, the send and receive ends of these special FIFOs can be spatially far apart, even spanning multiple FPGAs. By breaking designs at latency-insensitive boundaries, it is possible to automatically produce efficient and high performance implementations that span multiple FPGAs.

In this work, we constrain our designs to be composed of modules, abstract entities that interface to each other through asynchronous, latency-insensitive FIFO communications channels. The communications channels have two critical properties. First, they have arbitrary transport latency, and second, they have arbitrary, but finite size. Internally modules may have whatever behavior the designer sees fit, provided that the interface properties are honored. For example, once data has been obtained from an interface FIFO, it might pass through a traditional latency-sensitive hardware pipeline. Alternatively, the module could be implemented in software on a soft-core, or even on an attached CPU. Module internals do not matter. Rather, it is the latency-insensitive interface that is fundamental. Modules alone describe an abstract model of computation. To achieve a physical implementation, modules are mapped on to platforms, entities which can execute modules. Figure 1 shows a stylized representation of a design with four modules mapped on to two platforms. Cross-platform links transit special communications services, providing longer latency, point-to-point communications.

We will solve the following problems related to the mapping of modules to platforms. First, we introduce a means of describing latency insensitivity in an HDL. Second, we develop a portable technique for interfacing latency-insensitive modules with external, latency-sensitive wired devices. Third, we present a high-performance, low latency architecture for an automatically synthesizable deadlock-free inter-FPGA communication network. We tie the preceding technologies together in a compiler capable of automatically partitioning latency-insensitive designs across multiple FPGAs.

2.2 Related Work

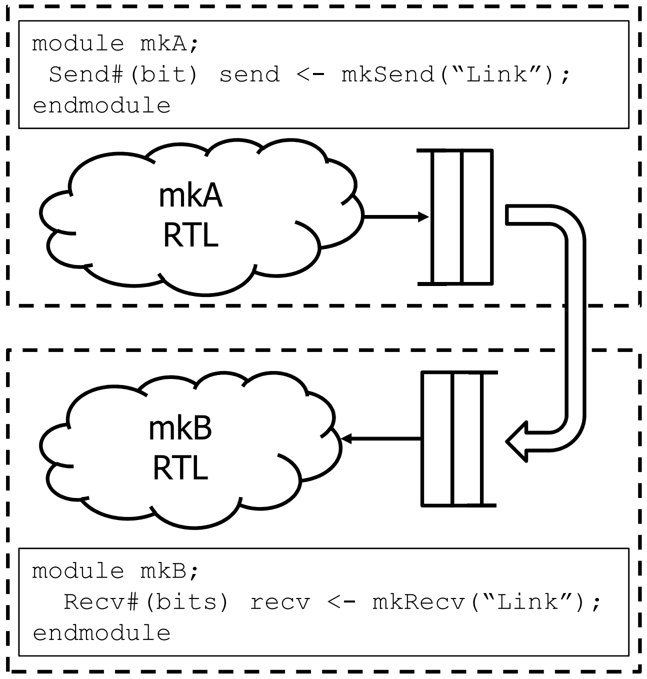
There exist a number of commercially available tools [13] [10] capable of mapping RTL designs across multiple FPGAs. These tools generate emulators intended primarily for ASIC verification in preparation for silicon implementation. As such, they are required to maintain the cycle accurate behavior of all signals in the design. Existing partitioning tools are differentiated by whether they provide dedicated [25] or multiplexed [2] [11] chip-to-chip wires.

Dedicated wire partitioning tools include inter-FPGA link delays in the circuit-level timing equations for determining setup and hold times. The result is that the emulation clock is greatly slowed, since delays on chip-to-chip wires are much longer than on-chip delays. However, board-level wires have physical meaning that may be useful in certain debugging regimes.

Multiplexed-wire partitioner operate by first running a single clock cycle of the base design on each FPGA, then propagating values across multiplexed inter-device links, and finally running another model cycle once all value from the previous cycle become available. As with dedicated wires, multiplexed wires incur performance overhead. To maintain cycle accuracy, the emulator must conservatively transport all inter-FPGA values every cycle, whether or not they impact the behavior of the succeeding cycle. As a result, these partitioning tools do not typically exhibit high performance, achieving cycle-accurate operating speeds of a few megahertz [23] [22].

Our tool is fundamentally different than either emulator approach: it is not required to maintain the cycle behavior of an unpartitioned design. Our language primitives allow designers to explicitly annotate locations in which it is safe to change cycle-by-cycle behavior of the design. As a result, partitions are free to run independently and operate on data as soon as it becomes available. This allows us to take advantage of the natural pipeline parallelism of hardware designs at a much finer grain than existing partitioners [21]. Furthermore, because we permit only FIFOs to cross between FPGAs, we transport only useful, explicitly enqueued values.

Our compiler operates on latency-insensitive designs, and the examples the we will present in Section- 7 were originally written in this style. However, it is not necessary to write latency-insensitive designs to make use of our compiler. Methodologies [24] [19] exist that transform existing latency-sensitive RTL designs into a functionally equivalent set of latency-insensitive modules *while preserving the timing behavior of the original design*. These methodologies seek to preserve the cycle-accurate behavior of some design signals, while permitting some parts of the original design to be re-written to



```
module mkA;
 Send#(bit) send <- mkSend("Link");
endmodule
```

```
module mkB;
  Recv#(bits) recv <- mkRecv("Link");
endmodule
```

Figure 2: A pair of modules connected by a latency-insensitive link. The arrow represents an automatically generated connection. Users may specify a minimum buffering as an argument to the link constructor. Here, "Link" is a tag used to match connections during the compilation flow.

more efficiently map onto an FPGA. Although our tool provides no inherent guarantees relating to cycle-accuracy, it can be composed with these transformation tools should the cycle-accuracy of some signals be required. Indeed, one of our example codes in Section 7 uses the A-Ports technique. In association with these tools, our compiler can be used to verify any synchronous design, including those designs written in a latency-insensitive style. However, there is no free lunch: as the number of cycle-accurate signals increases, our synthesized implementations will degrade in performance until they reach parity with traditional cycle-accurate partitioning tools.

3. LANGUAGE LEVEL SUPPORT

Synthesizable HDLs provide a very basic hardware abstraction: register, logic, and, to a lesser extent, memory. The behavior of these elements are tied to a specific clock cycle. Traditional HDLs reflect the physical reality of high-performance hardware, but do not offer the compiler much room to change cycle-to-cycle behavior. For example, if a programmer instantiates a two element FIFO in an HDL, this FIFO will have some well defined cycle-by-cycle behavior. Even changing the depth of this FIFO is difficult to automate, because changing the depth will almost assuredly perturb cycle behavior and potentially break the design. Even if the HDL design can tolerate such perturbations, it is difficult for the HDL compiler to prove that this is the case. Inter-FPGA communications, which are almost guaranteed to take many FPGA cycles, do not map well into the existing HDL model.

We avoid the generally undecidable problem of reasoning about cycle accurate behavior in the compiler by introducing a primitive syntax for latency-insensitive, point-to-point FIFO communication. By using these constructs, the programmer asserts that the compiler is free to modify the transport latency and buffer depth of the FIFO in question. This is a departure from previous multi-FPGA compilation efforts in that we permit the compiler to modify not only the cycle behavior of the communications link, but also the resulting behavior of the user design. Users must ensure that their designs can tolerate these changes in behavior, but in practice this is little different than interfacing to a normal, fixed-behavior FIFO. By providing a latency insensitive primitive, we push the burden of reasoning about high-level design properties to the designer, simplifying the task of the compiler to generating high-speed, deadlock-free interconnects.

An example of our link syntax [17] is shown in Figure 2. mkSend and mkRecv operate as expected, with the send endpoint injecting messages into a logical FIFO with arbitrary latency and receive draining those messages. This syntax is convenient because it provides a simple way for logically separate modules to share links while abstracting and encapsulating the physical interconnect. In our compiler, only these special FIFOs are treated as latency-insensitive. Other FIFOs, for example those provided in the basic language, retain their original fixed-cycle behavior.

In addition to point-to-point communications, we also provide a ring interconnect primitive. Like the point-to-point links, rings are named, but may have many ring stops. Ring stops are logically connected in sequence, with messages flowing around the ring. The ring primitive is useful in scaling and sharing runtime services across FPGAs, since rings can have an arbitrary number of stops.

4. PLATFORM SERVICES

Our model of computation consists of modules communicating over latency-insensitive FIFO links. However, practical FPGA systems must have some interaction with some platform-specific, wired devices, like memory. The difficulty with these physical interfaces is that we require the freedom to move modules to any FPGA, but physical interfaces are fundamentally unportable. Therefore, we introduce an abstraction layer [16], mapping physical devices into platform services using our latency-insensitive primitives. Services are tied to specific FPGA, but can be used by any module in a design. Common services in a platform runtime include memory and inter-FPGA communications, but application-specific platforms may include network, wireless, or video interfaces.

Platform services differ from user modules in two ways. First, platform services are permitted to have an arbitrary wire interface. Second, platform services are shared among user modules. In addition to a wired, external interface, platform services must also expose a set of latency-insensitive links, typically some form of request-response. User modules may then interface to a platform service as if it was a regular module.

Since memory is perhaps the most important service, we will use our memory hierarchy, shown in Figure 3, as an example of the design of a platform service. This hierarchy is an extension of the scratchpads [1] memory abstraction. In the scratchpads hierarchy, modules are presented the abstraction of a disjoint, non-coherent, private memory spaces with simple read-write interfaces. Backing this simple interface is a complex, high-performance cache hierarchy.

Platform services may be shared among several different clients. In this case, each client instantiates an interface provided by the service. The service interface internally connects the clients, typically using a ring, and handles multiplexing among clients. In the case of scratchpads, each client instantiating a memory link gets a private cache, connected to the rest of the memory hierarchy by a ring. At runtime, the clients are dynamically discovered as they begin sending memory traffic. The ring permits a single, parametric service implementation to automatically scale to an arbitrary number of clients and FPGAs.

Scratchpads also introduces a shared chip-level L2 cache to improve memory performance. In the context of multiple FPGAs, there are two interesting cases for a chip-level resource: the case in which a resource is located on a remote FPGA and the case in which several FPGAs provide the same resource.

Several FPGAs may provide the same service. If useful, a chip-level ring can also be introduced connecting the services and allowing them to communicate with each other dynamically. In the case of Scratchpads, a secondary ring is introduced on top of the chip-level caches allowing all of them to share a single interface to host virtual memory.

In the case that a platform does not provide a service required by a module mapped to the platform, the compiler will automatically connect the module to an instance of the service provided on a different FPGA. This flexibility in module mapping is valuable in cases where the runtime service is used infrequently. In the case of scratchpads, the local ring will automatically be mapped across FPGA boundaries, resulting in a correct, but lower performance implementation. In the case that there are several remote services available, modules have the option of specifying the FPGA whose service should be used.

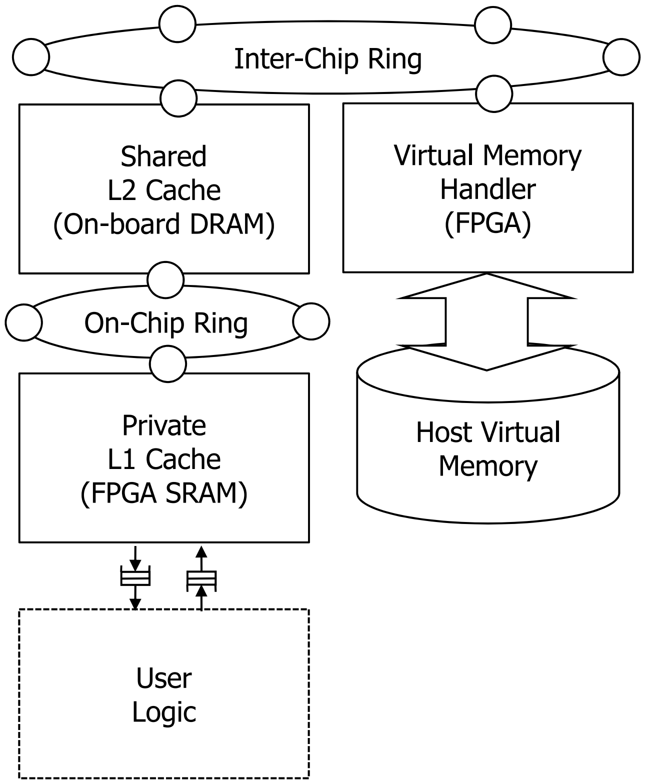

Inter-Chip Ring
Shared L2 Cache (On-board DRAM)
Virtual Memory Handler (FPGA)
On-Chip Ring
Private L1 Cache (FPGA SRAM)
Host Virtual Memory
User Logic

Figure 3: A view of a scalable multiple FPGA memory hierarchy. Fast, private local caches are backed by a shared last-level cache, which in turn is backed by global virtual memory. The structure of the hierarchy is automatically inferred at compilation time.

5. COMPILATION

Given a design consisting of modules connected by latency-insensitive links, the goal of our compiler is to automatically connect all the exposed links by synthesizing the most efficient network interconnect possible. Modules mapped to the same FPGA may have an interconnect as simple a FIFO, but distant modules may require an interconnection network transiting several FPGAs.

Our compilation currently assumes a static, user provided mapping of modules to FPGAs. Initially, source modules are decorated with tags representing the specific FPGA to which the module has been mapped. Synthesis proceeds per FPGA in two passes. During the first pass, source for the entire system is compiled in order to discover latency-insensitive endpoints. As endpoints are discovered within modules, they are tagged with the FPGA to which they belong. At the end of the pass, each FPGA has a set of dangling send and receive connections associated with it. Some of these represent local connections and some represent off-chip links. Sends and receives are then matched by name on a per FPGA basis. Matched links represent connections local to the specific FPGA and are connected by a simple FIFO, while unmatched names are propagated to a global matching stage.

The global matching stage synthesizes a router for each specified communication link between FPGAs. Connections are matched based on name and are routed to their destination, which may require inserting links across several intervening FPGAs. The routers generated in this stage are parameterized instances of the communications stack discussed in Section 6. Each connection is assigned a virtual channel on each link that it traverses. At the source and sink of each path, the inverse latency-insensitive primitive is inserted into the router, for example, a dangling send will have its corresponding receive inserted. During the second compilation pass, the inserted connections will be matched with existing connections, completing the inter-chip routing. For intermediate hops, a new, unique link is introduced between ingress and egress. Figure 4 depicts the result of the global compilation for a single FPGA, in which the user logic has three inter-chip links, and one link transits the FPGA.

The generated routers are then injected into the compilation for each FPGA, and compilation proceeds as in the first pass. However, at the end of this pass, during the local match step, all formerly dangling connections are matched to local endpoints at the synthesized routers. The Verilog generated by this final step can be simulated or passed to back-end tools to produce bit-files.

6. INTER-FPGA COMMUNICATION

There are two issues in synthesizing an inter-FPGA communications network for latency-insensitive links: performance and correctness. Partitioning designs at latency-insensitive FIFOs allows us to transport only explicitly enqueued data. However, to achieve high performance a network must be able exploit the pipeline parallelism inherent in the partitioned design. In practice this means that many messages must be in-flight at any given time.

For our model of computation, network correctness means the in-order delivery of messages, a relatively simple requirement. However, because many links can cross between FPGAs, there is a need to multiplex the physical links between the FPGAs. This multiplexing can introduce deadlocks, but we will show that our synthesized networks are *deadlock-free*.

Deadlocks arise in shared interconnect when dependent packets are forced to share the same routing paths, which can cause the packets to block each other. To get around this issue, virtual channels are introduced to break dependence cycles [4]. In traditional computer architectures, this is a tractable problem since the communications protocols are known statically and dependencies can be explicitly broken at design time. However, reasoning about the communications dependencies of an arbitrary hardware design is difficult. Therefore, we simply allocate a virtual channel to each link crossing between FPGAs. Virtual channel allocation alone is not sufficient to ensure deadlock freedom, because full virtual channels can still cause head-of-line blocking across the shared physical links. To resolve this issue we must also introduce flow control across each virtual channel. Together, universal virtual channel allocation and flow control are sufficient to guarantee that our compiler does not introduce deadlocks into previously deadlock-free latency-insensitive designs. This property is an easy corollary of the Dally-Seitz theo-

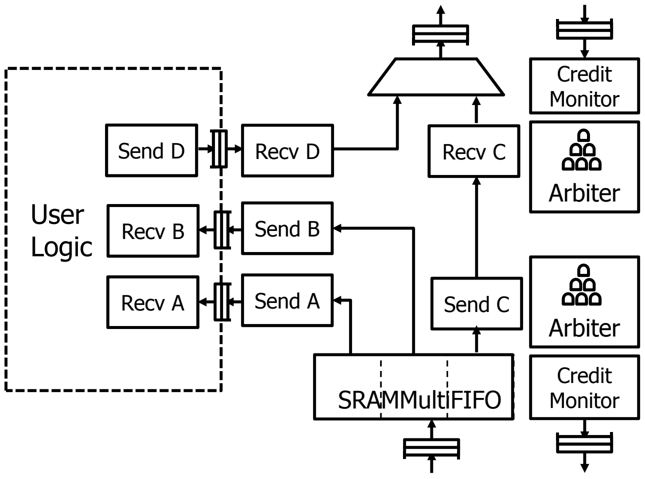


Figure 4: An example of a synthesized router connecting an FPGA with to two other FPGAs (not shown). Link "Recv C" is routed through the FPGA.

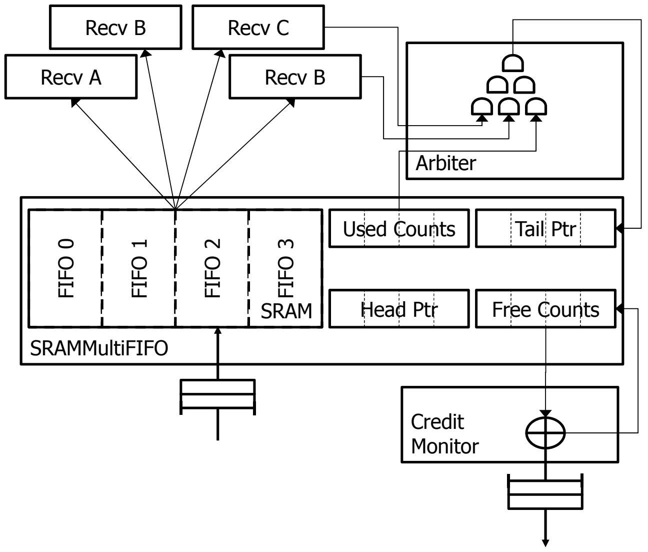


Figure 5: FIFOs are folded onto a single logical SRAM resource. Each FIFO in the SRAM represents a buffer for a single virtual channel

rem [4], wherein we insert a virtual channel for each communication link, trivially preventing dependent packets from blocking one another.

We require flow control per link to guarantee functional correctness in our partitioned designs. This is a seemingly costly proposition, particularly since flow-control packets incur a high latency round-trip between FPGAs. This latency appears to create a cost-performance tradeoff between buffering per virtual channel and the performance and area of the router, since too little buffering can cause the sender to stall before the receiver even begins to receive packets, while too much buffering reduces the area available to the user design. A naive register-based flow control implementation with buffering sufficient to cover a round-trip latency of 16 cycles requires half the area of a large FPGA. Clearly, this kind of implementation does not scale beyond a pair of FPGAs.

The problem with the register-based design is that it is too parallel and therefore needlessly wasteful of resources. In any cycle, any of the registers in any of the buffers can potentially supply a data value to transmit. However, we observe that the inter-chip bandwidth between FPGAs is limited to a single, potentially wide, data per cycle. This bandwidth limitation means that to sustain the maximum rate across the link, we need to enqueue and dequeue exactly one FIFO in any given cycle. Therefore, a structure with low parallelism, but high storage density is sufficient to sustain nearly the maximum throughput of the physical channel.

Most modern FPGAs are rich in SRAM, with a single chip containing megabytes of storage. Although large amounts of memory are available, the bandwidth to each slice of this memory is limited to a single four to eight byte word per cycle. Because inter-FPGA communication is similarly constrained, we can map many virtual channels with relatively large buffers onto the resource-efficient SRAM without significant performance loss. We call this optimized storage structure, depicted in Figure 5, the SRAMMultiFIFO (SMF) [7]. Because the SMF maps many FIFOs onto a single SRAM with a small number of ports, it must introduce an arbiter to choose which FIFO will use the SRAM port in a given cycle. SMF FIFOs have uniform and constant size which simplifies control logic at the cost of storage space. The SMF is fully pipelined, and each mapped FIFO can utilize the full bandwidth of the SRAM.

Area usage for SMF and a functionally similar register-based FIFO implementation are shown in Figure 6. The SMF scales in BRAM usage as FIFO depth increases, consuming only around 2% of slices on a Virtex-5 LX-330T. The low area usage of the SMF-based switch makes it amenable to FPGA platforms with a high-degree of inter-platform interconnection. On the other hand, the registered buffer schemes can quickly exhaust large amounts of area. The largest implementable registered FIFO switch has no better performance than a more resource efficient, but deeper SMF switch, despite the inherent parallelism of the registered implementation.

The density of the SMF fundamentally changes the way that communication networks between FPGAs are designed. Unlike processor network on chips, which multiplex virtual channels and offer extremely limited buffering in network to conserve area, SMF based switches can liberally allocate virtual channels to each connection traversing the inter-FPGA link without significant area penalty. As a result, concerns involving shared virtual channels [14] do not apply to our switches and routing scheme. Because SMF provides deep buffers, each flow-controlled inter-chip channel can sustain full bandwidth across high-latency physical links. Deep buffers also permit us to send control messages relatively infrequently, minimizing throughput loss.

In our switches, we use a simple block-update control control scheme. The virtual channel source keeps a conservative count of the number of free buffer spaces available at the virtual channel sink. Each time a packet it sent, this count is decremented. The virtual channel sink maintains a count of the free buffer space available, which is updated as user logic drains data out of the virtual channel. When this free space counter passes a threshold, it is set to zero and a bulk credit message is sent to the virtual channel source. These credit messages are given priority over the data message to improve throughput.

Although the SMF is the core of our router, the router architecture consists of three pipelined layers: packetization, virtual channel, and physical channel. At compilation time the compiler generates an FPGA-specific router using parametric components from a library.

The physical channel layer consists of specially annotated FIFOs provided by the platform runtime. The backing implementation of these FIFOs is irrelevant from the user's perspective and could range from LVDS to Ethernet. Given the specific name of an inter-FPGA communications link provided by the platform runtime, the

	LUTS	Registers	BRAM	Relative Performance
Registered FIFOs, depth 8	10001	22248	0	1
Registered FIFOs, depth 32	25494	68813	0	1.11
SRAM MultiFIFOs, depth 32	4996	4778	2	1.09
SRAM MultiFIFOs, depth 128	5225	4850	8	1.11

Figure 6: Synthesis and performance metrics for various switch architectures. Results were produced by mapping a simple HAsim dual core processor model to two FPGAs. In this design, 29 individual links and 1312 bits cross the inter-FPGA boundary.

compiler simply instantiates a connection to the link and ties it in to the synthesized communication hierarchy.

We introduce packetization into our communications hierarchy to simplify both the virtual-channel hardware layer and to handle the presence of wide links. Since all communications links and link widths are statically determined at compile time, our compiler can infer bit-optimal packet protocol for each link. These protocols are specific instantiations of a header-body packet schema in which the header contains information about the packet length, type, and virtual channel. The parameterized packetization and de-packetization hardware then infer an efficient implementation based on the data width to be transported. In the case that the data width is wider than the physical link, marshalling and de-marshalling logic is automatically inserted. However, if the data width is sufficiently small, the packet header and body will be bit-packed together. Since the data communicated between FPGAs tends to be narrow, this is a significant performance optimization.

7. EVALUATION

To evaluate the quality of our compiler, we partition three large research prototypes. These prototypes already used latency insensitive links to obtain better modularity and we were able to partition and run these designs *without source modification*. We tested our designs on two platforms: the ACP [12], consisting of two Virtex-5 LX330 chips mounted on Intel's front-side bus, and a multiple FPGA software simulator, which can model an arbitrary number and interconnection of FPGAs.

Partitioning a design using our compiler has four potential benefits. First, wall-clock runtime of the design can decrease, due to improved clock frequency and increased access to resources. Second, some designs can be scaled to handle larger problem sizes, again due to increased access to resources. Third, synthesis times are reduced due to the smaller size of design partitions. Fourth, partial recompilation is available in earnest because only those FPGAs that have changing logic need to be rebuilt. Different designs will experience different combinations of these salutary effects. On the other hand, because we are partitioning a design between chips, any communication between the chips will have increased latency. Our experiments will show that this negative effect is minimal for typical designs; the natural pipeline parallelism of hardware and improved operating frequency together compensate for increased latency.

Wireless Processing: Airblue is a highly parametric library for implementing OFDM baseband processors such as WiFi and WiMAX. A typical baseband pipeline implemented in Airblue, shown in Figure 7 has relatively little feedback, although the main data path has high bandwidth and low latency requirements. Typical wireless protocols implemented using Airblue have protocol-level latency requirements on the order of tens of microseconds. Based on these requirements, our compiler presents an ideal mechanism for scaling Airblue protocol implementations to multiple FPGAs, because our partitioned implementations can be made to favor high-bandwidth links even in the presence of inter-FPGA traffic on non-critical links. The latency introduced by inter-FPGA hops is small, approximately 100 nanoseconds, and well within the timing requirements of the high-level protocols.

To evaluate our compiler, we partition a micro-architectural simulator for SoftPHY [8], a recently proposed cross-layer protocol which extends commercially deployed forward error correction schemes to improve wireless throughput. Partitioning benefits the micro-architectural simulator in two ways. First, because only the microarchitecture of the error correction algorithm varies, by partitioning the simulator at the error correction algorithm, the bulk of the hardware simulator needs to be compiled only once. To test a different algorithm, a relatively small logical change, only one bit-file needs to be rebuilt. Second, because the clock frequencies of the FPGAs can be scaled, the wall-clock performance of the simulator improves.

Figure 12(b) shows the normalized performance of two experiments: one using a complex software channel model and the other using a simpler hardware channel model. In the first experiment, the software channel model is the performance bottleneck and limits the throughput of both the single and multiple FPGA implementations. In this case the multiple FPGA implementation achieves near performance parity with the single FPGA implementation, even though it has a much higher clock frequency. For one data point, QAM-64, the multiple FPGA implementation slightly outperforms single FPGA implementation. This is because QAM-64 produces more bits per software communication and begins to overwhelm the serial portions of the slower single FPGA implementation.

When a simpler channel model is implemented in hardware, the multiple FPGA implementation outperforms the single FPGA implementation. In this case, the normalized performance is tied to the clock frequency ratios of the two designs. For BPSK, which stresses the FFT, the ratio is highest, since the FFT is located on FPGA 0 in the partitioned implementation. For higher bit-rate modulation schemes, the bit-wise error correction, located on FPGA 1, is the bottleneck. Since the ratio of the clocks of the single FPGA implementation and FPGA 1 is smaller, the performance gap narrows.

Processor Modeling: HAsim is a framework for constructing high speed, cycle-accurate simulators of multi-core processors. Like many FPGA-based processor models, HAsim uses multiple FPGA cycles to simulate one model cycle [18]. HAsim uses a technique called A-Ports [19] to allow different modules in the processor to simulate at different and runtime variable FPGA-cycle-to-Model-cycle Ratios (FMR). This makes HAsim amenable to our multi-FPGA implementation technique, as the A-Ports protocol can be layered on top of our latency-insensitive links without affecting the ability of A-Ports to resolve the cycle-by-cycle behavior of the original design. HAsim is written in a highly parameterized fashion, both in terms of the structure and the number of the cores modeled. HAsim models can scale to hundreds or thousands of cores by changing a handful of parameters, an important feature for modeling future processors. The difficulty in modeling such large processors is that, even though describing the models using HAsim is straightforward, the models themselves do not fit in a single FPGA.

HAsim is divided into a functional partition and a timing partition, which separates the calculation of simulation results from the amount of time that those results take in the modeled processor [20]. This partitioning creates a high degree of feedback. For example,

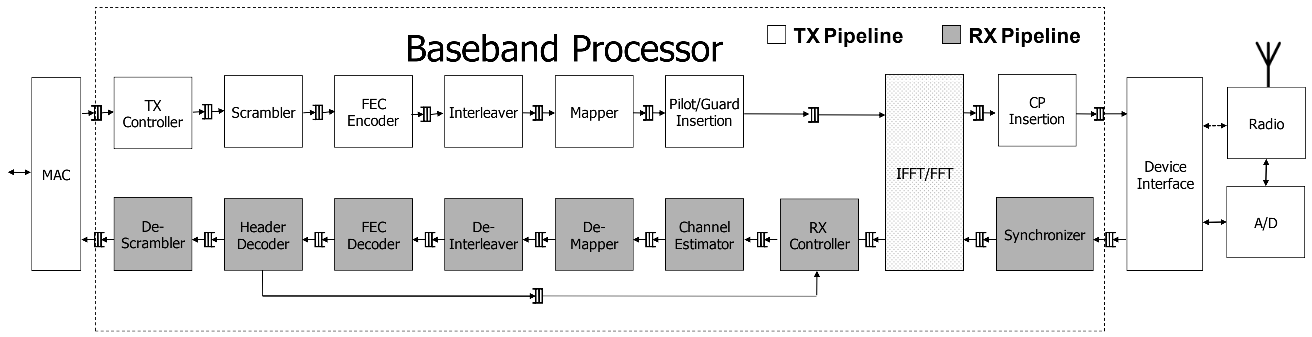


Figure 7: An Airblue 802.11g-compatible transceiver. In the SoftPHY experiment, only the forward error correction (FEC) decoder block is modified.

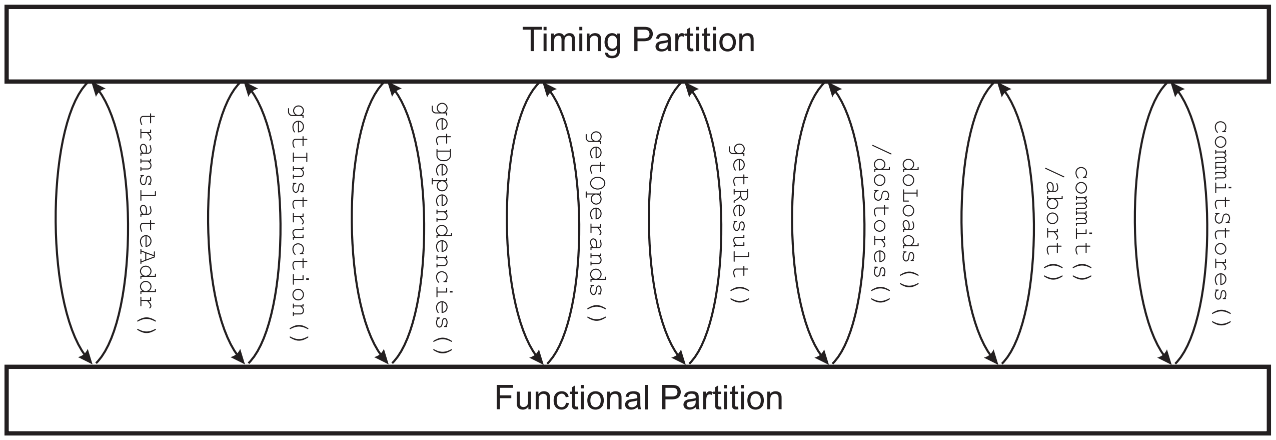


Figure 8: HAsim partitioned processor simulator. The timing partition relies on the functional partition for all computation related tasks, for example, instruction decoding.

the timing partition must query the functional partition to decode an instruction and wait for a response before proceeding. Similar feedback loops arise in the other processor stages and in the cache model. Despite this level of feedback, a natural mapping of HAsim to two FPGAs is placing the timing and functional partitions on separate FPGAs. This partition, shown in Figure 8, is attractive because all HAsim timing models share a common functional partition, enabling our compiler to compile the functional partition once and reuse it among many timing models.

The timing-functional partitioning works well because in practice HAsim is latency tolerant by design. In order to scale to multicore configurations without using large numbers of FPGAs, HAsim uses time-multiplexing to map several virtual processors onto a limited number of physical processors. This multiplexing means that individual logical cores can wait dozens of cycles for responses from the functional model without reducing overall model throughput. Moreover, this tolerance scales as the number of simulated cores increases.

Although HAsim gracefully degrades its performance in the presence of limited resources, introducing more resources both speeds simulation and enables HAsim to scale to simulations of larger numbers of more complex cores. In particular, large HAsim models need large amounts of fast memory. Partitioning HAsim designs among multiple FPGAs automatically introduces new chip-level resources, like DRAM, into the synthesized implementation, increasing cache capacity and memory bandwidth.

On a single FPGA, HAsim scales to 16 cores before the FPGA runs out of resources. By mapping HAsim to two FPGAs, we are able to build a partitioned model capable of supporting up to 128 cores and give them access to approximately twice the memory capacity and bandwidth of a single FPGA implementation. We achieve super linear scaling in problem size because many structures in HAsim are either time-multiplexed among all cores or scale logarithmically with the number of cores.

We evaluate the throughput of the models by running a mix of SPEC2000 integer and floating point applications in parallel on the modeled cores. Figure 12(c) shows the normalized performance of the multiple-FPGA simulator relative to the single FPGA simulator. For small numbers of cores, the gap between the single FPGA and multiple FPGA simulator is large, due to the request-response latency between the timing and functional partitions. However, as the number of simulated processors scales, models become more latency tolerant, and the performance gap closes.

The raw performance of various HAsim implementations is shown in Figure 9. Single FPGA performance decreases from 8 to 16 cores due to increased cache pressure on the simulator's internal memory hierarchy. The partitioned processor model achieves about 75% the aggregate throughput of the 16-core single FPGA implementation, due to the latency of communication between chips. As we scale the number of cores in the partitioned model, throughput increases until we hit 36 cores. The reason for this throughput improvement is that larger numbers of cores in a multiplexed model are more resilient to inter-link latency. Unlike the single FPGA implementation, cache pressure plays less of a role in the performance of the

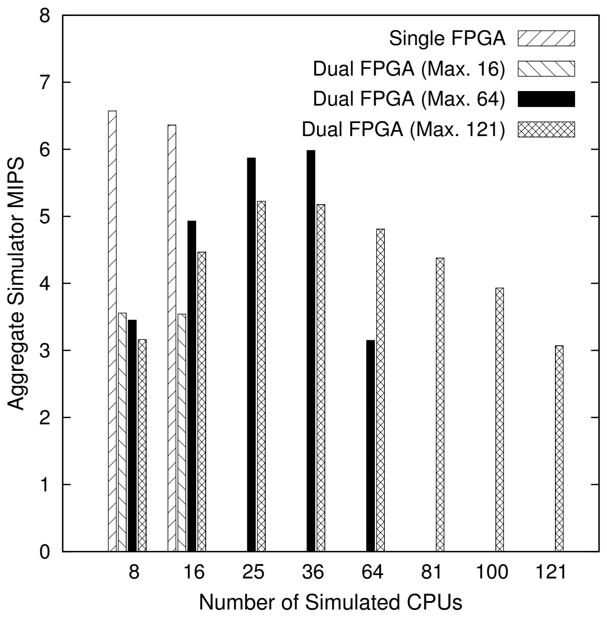


Figure 9: Performance results for various HAsim simulation configurations. Simulated cores run a combination of wupwise, applu, gcc, mesa, mcf, parser,perlbmk, and ammp from the SPEC2000 suite.

partitioned implementation because both FPGAs have chip-level memory caches, doubling the cache size and bandwidth available to the model. As the maximum number of simulated cores increases, the FPGA becomes more crowded, reducing operating frequency. As a result, partitioned models supporting more cores have lower performance even when simulating the same number of cores as a smaller model.

Video Decoder: H.264 [9], shown in Figure 11, is a state of the art video decoder, which has seen broad deployment both in custom hardware and in software. H.264 has several potential levels of implementation with widely varying feature sets and performance requirements. When implementing these various feature sets, it is useful to have a platform for rapidly evaluating the performance of different micro-architectures and memory organizations. The lower compile times offered by our compiler are useful in this kind of architectural exploration.

H.264 is naturally decomposed into a bit-serial front-end and a data parallel back-end. The front-end handles decompression and packet decoding, while the back-end applies a series of pixel-parallel transformations and filters to reconstruct the video. H.264 has limited feedback between blocks in the main pipeline. The pipeline synchronizes only at frame boundaries, which occur at the granularity of millions of cycles. Intrapredicition does require some feedback from interprediction, but this feedback is somewhat coarse-grained, occurring on blocks of sixty-four pixels.

Because H.264 generally lacks tight coupling among processor modules, many high performance partitionings are possible. We choose to partition the bit-serial fronted because the front-end computation does not parallelize efficiently. As such, its performance can only be increased by raising operating frequency. The front-end also contains a number of difficult feedback paths, which end up limiting frequency in a single FPGA implementation.

Figure 12(a) shows the performance of a partitioned implementation of H.264 relative to a single FPGA implementation. In the case of the low resolution, the multiple FPGA implementation outperforms the single FPGA implementation by 20%. This performance gain comes from increasing the clock frequency of the partitioned implementation relative to the single FPGA implementation. However, at higher resolution, interprediction memory traffic becomes more significant which has the effect of frequently stalling the processing pipeline. As a result some part of the latency of inter-chip communications is exposed and the multiple FPGA performance degrades slightly.

Compilation Time: To this point, we have focused on the wall clock performance and design scaling that our system provides. However, our compiler also provides another important performance benefit: reduced compilation time. FPGAs have notoriously long tool run times, primarily due to their need to solve several intractable problems to produce a functional design. In practice, these run times represent a serious impediment both to experimentation and to debugging. Our compiler helps alleviate the compilation problem in two ways. First, partitioned designs are fundamentally easier to implement; in the context of nonlinear run times, even a small decrease in design size can reduce compilation time significantly. Second, by partitioning we obtain a degree of modular compilation. If the design is modified, but the gate-ware of a partition has not changed, then that partition does not need to be recompiled. This savings is significant in two contexts: debugging and micro-architectural experimentation. In the case of debugging, the utility of the shortened recompilation cycle is obvious. However, the need to compile a partition only once per set of experiments is perhaps more beneficial. In this case, effort can be spent tweaking the tools to produce the best possible implementation of the shared infrastructure, in order to accelerate all experiments. In the case of HAsim, a single functional partition can be used in conjunction with all timing partitions.

Figure 13 shows selected compilation times for single and partitioned designs. It is important to note that the numbers reported for multiple FPGA designs reflect parallel compilation on the same machine, although to maximize speed, compilation should be distributed. By partitioning we achieve reduced compilation time, even though, in aggregate, we are building more complex, higher frequency designs. For Airblue and HAsim, the two examples in which modular recompilation of FPGA 1 is a useful, our recompilation facilities represent a substantial time savings.

8. CONCLUSION

In this paper, we present a language extension and compiler that leverages latency-insensitive design to produce high-performance implementations spanning multiple FPGAs. Our language and compiler permit us to build larger research prototypes, improve compilation time, and, in some cases, gain performance over single FPGA implementations.

Our compiler performs best in partitioning digital signal processing applications. These applications usually feature high bandwidth and computation requirements, but very little global control or feedback. As a result they are more resilient to the latency introduced in chip-to-chip communication and have the potential for super-linear performance increases when scaling to systems with multiple FPGAs. Applications with larger amounts of feedback, like processor prototypes, may experience performance degradations relative to a single FPGA due to latency. However, these applications still benefit from improved access to resources, design scaling, and reduced compile times.

The compiler that we have presented in this paper is promising, and we see four areas of exploration moving forward. The first is hardware-software co-design. Sequential languages are intrinsically latency-insensitive and so can be easily integrated into our model of computation. We believe that our proposed syntax provides a convenient mechanism for bridging the gap between both host PC and soft-cores instantiated on the FPGA itself. Second, we see compiler optimization as an area for exploration. Unlike traditional

	LUTS	Registers	BRAM	fMax(MHz)
Airblue, SOVA, Single	115780	67975	46	25
Airblue, SOVA, FPGA 0	77982	56499	34	65
Airblue, SOVA, FPGA 1	46852	21707	39	45
HAsim, 16 cores, Single	185002	153906	127	70
HAsim, 16 cores, FPGA 0	119231	102161	136	75
HAsim, 16 cores, FPGA 1	123892	99066	88	80
HAsim, 64 cores, FPGA 0	148107	108617	220	65
HAsim, 64 cores, FPGA 1	164920	111145	133	70
H.264, Single	79839	59212	63	55
H.264 FPGA 0	66893	52860	65	65
H.264 FPGA 1	13998	9493	19	85

Figure 10: Synthesis metrics for single and multiple FPGA implementations of our sample designs. Xilinx 12.1 was used to produce bit-files. To limit compile times, we stepped fMax at increments of 5MHz.

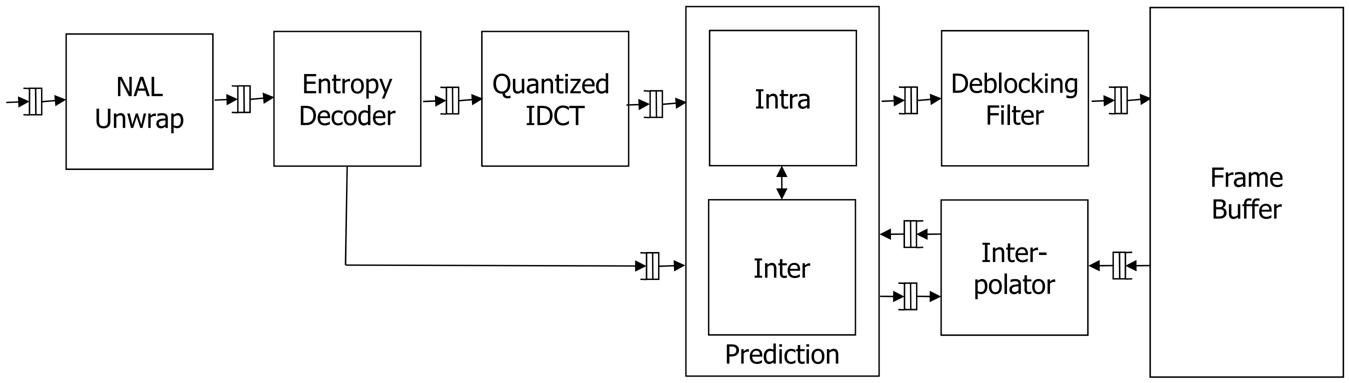


Figure 11: An H.264 decoder.

circuits, for which communication is statically determined by wire connection, our designs exhibit complex phase behavior. We believe that static, dynamic, and feedback-driven optimization techniques may be applied with great effect. Third, we see an opportunity to introduce quality of service (QoS) to improve design reliability. Programmer visible QoS is necessary to maintain program correctness for those workloads with high-level latency and throughput requirements, such as wireless protocols. Fourth, we see our model of computation as a means of alleviating the place and route problem, even on a single FPGA. Reuse of pre-routed components is difficult in current FPGA design because there is no guarantee that the tool can make timing on inter-component interface wires. However, our paradigm breaks these long paths with registered buffer stages as needed. For many designs, including those presented in this paper, small delays are negligible, especially during debugging and design exploration. By enabling large-scale component reuse, the synthesis back-end might reduce to a simple and fast linking step, dramatically reducing compile times.

Acknowledgements: During the course of this work, Kermin Fleming was supported by the Intel Graduate Fellowship.

9. REFERENCES

[1] Michael Adler, Kermin Fleming, Angshuman Parashar, Michael Pellauer, and Joel S. Emer. LEAP Scratchpads: Automatic Memory and Cache Management For Reconfigurable Logic. In *FPGA*, pages 25–28, 2011.

[2] Jonathan Babb, Russell Tessier, Matthew Dahl, Silvina Hanono, David M. Hoki, and Anant Agarwal. Logic Emulation With Virtual Wires. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(6):609–626, 1997.

[3] Luca P. Carloni, Kenneth McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, 20(9), September 2001.

[4] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, 36:547–553, May 1987.

[5] Nirav Dave, Man Cheuk Ng, Michael Pellauer, and Arvind. Modular Refinement and Unit Testing. In *MEMOCODE'10*.

[6] K. Fleming, Chun-Chieh Lin, N. Dave, Arvind, G. Raghavan, and J. Hicks. H.264 Decoder: A Case Study in Multiple Design Points. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 165 –174, Jun. 2008.

[7] Kermin Fleming, Myron King, Man Cheuk Ng, Asif Khan, and Muralidaran Vijayaraghavan. High-throughput Pipelined Mergesort. In *MEMOCODE*, pages 155–158, 2008.

[8] Kermin Elliott Fleming, Man Cheuk Ng, Samuel Gross, and Arvind. WiLIS: Architectural Modeling of Wireless Systems. In *ISPASS*, pages 197–206, 2011.

[9] ITU-T Video Coding Experts Group. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, May, 2003.

[10] `http://www.cadence.com/products/sd/palladium_series/pages/default.aspx`. "Cadence Palladium".

[11] http://www.eda.org/itc/scemi.pdf. Standard Co-Emulation Modelling Interface (SCE-MI): Reference Manual.

[12] http://www.nallatech.com. Nallatech ACP module.

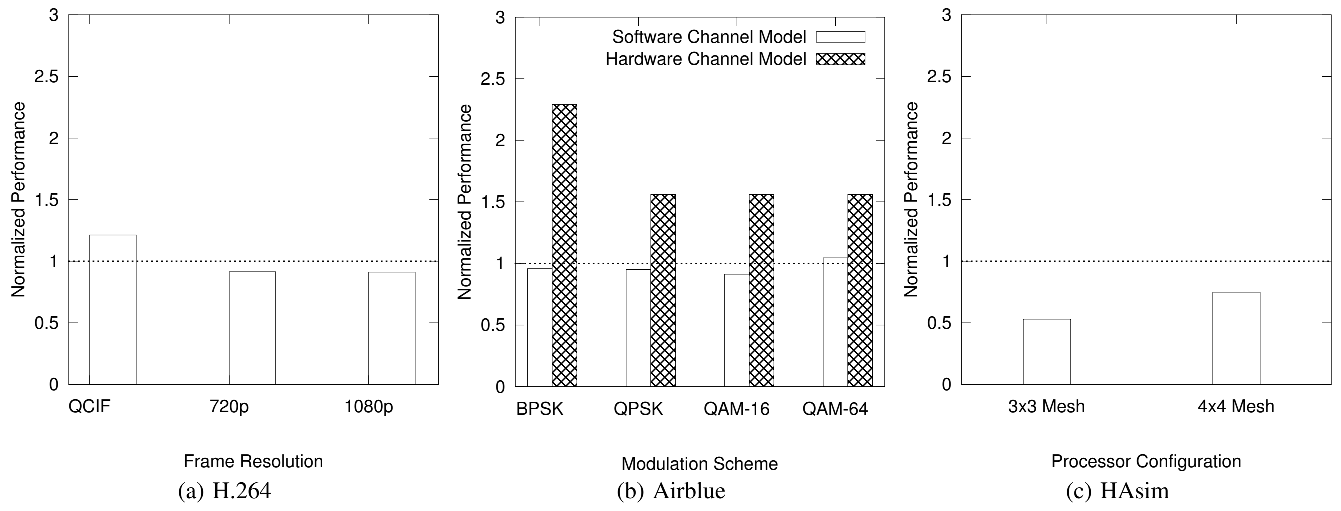


Figure 12: Performance results for various two FPGA partitioned workloads. Performance is normalized to a single FPGA implementation of the same hardware.

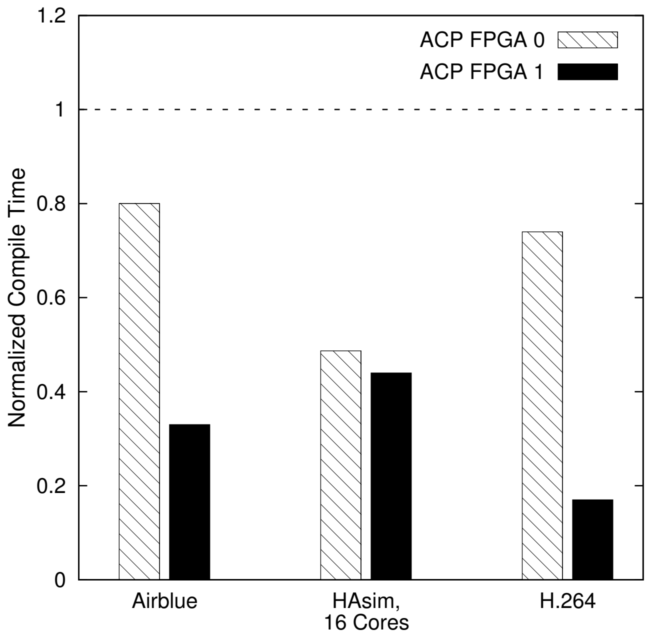


Figure 13: Compilation time relative to a single FPGA build. Xilinx 12.1 was used to produce bit-files. Compile times were collected on an unloaded Core i7 960 with 12 GB of RAM. Note that for multiple FPGA builds, the FPGA builds can proceed in parallel.

[13] "`http://www.synopsys.com/Systems/FPGABasedPrototyping/pages/certify.aspx`". "Synopsys Certify".

[14] Michel A. Kinsy, Myong Hyon Cho, Tina Wen, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. Application-aware Deadlock-free Oblivious Routing. In *ISCA*, pages 208–219, 2009.

[15] M. C. Ng, K. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan. Airblue: A System for Cross-Layer Wireless Protocol Development. In *ANCS'10*, San Diego, CA, 2010.

[16] Angshuman Parashar, Michael Adler, Kermin Fleming, Michael Pellauer, and Joel Emer. LEAP: A Virtual Platform Architecture for FPGAs. In *CARL '10: The 1st Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.

[17] M. Pellauer, M. Adler, D. Chiou, and J. Emer. Soft Connections: Addressing the Hardware-Design Modularity Problem. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 276–281. ACM, 2009.

[18] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing. In *The 17th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2011.

[19] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. A-Ports: An Efficient Abstraction for Cycle-Accurate Performance Models on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2008.

[20] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. Quick Performance Models Quickly: Closely-Coupled Timing-Directed Simulation on FPGAs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2008.

[21] Charles Selvidge, Anant Agarwal, Matthew Dahl, and Jonathan Babb. TIERS: Topology Independent Pipelined Routing and Scheduling for Virtual Wire Compilation. In *FPGA*, pages 25–31, 1995.

[22] Todd Snyder. Multiple FPGA Partitioning Tools and Their Performance. Private communication, 2011.

[23] Russel Tessier. Multi-FPGA Systems: Logic Emulation. *Reconfigurable Computing*, pages 637–669, 2008.

[24] Muralidran Vijayaraghavan and Arvind. Bounded Dataflow Networks and Latency-Insensitive Circuits. In *MEMOCODE'09*, Cambridge, MA, 2009.

[25] Nam Sung Woo and Jaeseok Kim. An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementation. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 202–207, New York, NY, USA, 1993. ACM.