Soft Connections: Addressing the Hardware-Design Modularity Problem

Michael Pellauer[†]   Michael Adler[‡]   Derek Chiou[*]   Joel Emer[†‡]

[†] Massachusetts Institute of Technology
Computer Science and A.I. Lab
Computation Structures Group
{pellauer, emer}@csail.mit.edu

[‡] Intel Corporation
VSSAD Group
{Michael.Adler,
Joel.Emer}@intel.com

[*] University of Texas at Austin
Electrical and Computer
Engineering Department
derek@ece.utexas.edu

ABSTRACT

Hardware-design languages typically impose a rigid communication hierarchy that follows module instantiation. This leads to an undesirable side-effect where changes to a child's interface result in changes to the parents. Soft connections address this problem by allowing the user to specify connection endpoints that are automatically connected at compilation time, rather than by the user.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids - *hardware description languages*

General Terms

Design, Languages

Keywords

High-Level Communication Description

1. INTRODUCTION

Modularity is a critical feature of high-level hardware description languages (HDLs). Ideally designers should be able to swap alternative modules in a "plug-and-play" manner. Such swapping enables code reuse and design-space exploration, and thus enhances designer productivity.

It is becoming increasingly popular to insert an FPGA into a general-purpose computer using a fast link such as PCIe [8] or Intel Front-Side Bus [7]. In such a setup the FPGA, configured by a standard HDL toolchain, acts as an accelerator to the CPU, running standard software. This usage model is gaining traction in the microprocessor performance modeling community, being used by projects such as Protoflex [6], UT-FAST [4], [5] and our HAsim simulator [2], [9] as part of the umbrella RAMP project [11]. In such an environment FPGA reconfigurations are frequent, so modular refinement and reuse become especially important.

In structural HDLs it can be difficult to swap one module for an alternative in isolation. This is because communication between modules can only follow the instantiation hierarchy. A module can only pass wires to its parent and children. Cross-hierarchical communication goes through the least-common ancestor and every other intervening node. If a new module requires communication with anything other than its direct parent, then we must change the parent module, the parent's parent, and so on.[1]
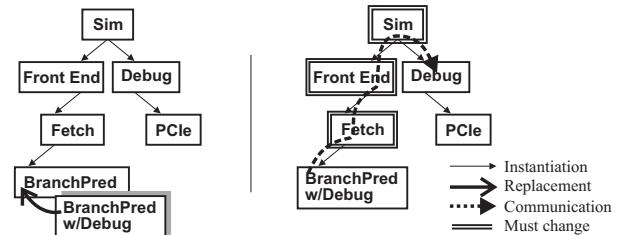
**Figure 1. Introducing cross-hierarchical communication.**
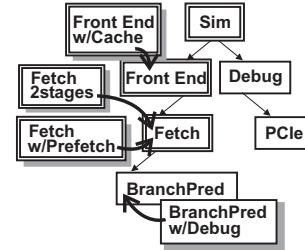
Figure 2. Alternative modules can worsen the problem.

Consider the situation shown in Figure 1. The designer knows that the Branch Predictor on the FPGA has a bug. He wants to swap it for a variant that outputs additional debugging information, that is sent to the host processor using PCIe. In order to do so he must add those wires to the Fetch, Front End, and top-level Simulator modules, then down to the Debug block.

The situation quickly deteriorates as we add more module alternatives to the system. In Figure 2 we have three alternative Fetch units and two Front Ends that the designer is exploring. Each setup uses the branch predictor, and each manifests the bug in different ways. The designer must now produce alternative implementations of these, one of which does not pass debugging wires up, and one which does. In the worst case the number of modules needed grows multiplicatively with the number of alternatives.

In this paper we attack this problem by "softening" the rigid communication hierarchy—thus we name our technique Soft Connections. This scheme restores modularity by allowing the designer to specify a *logical topology* of communication which is separated from its physical implementation. The endpoints are not connected by the user, but rather done automatically using static elaboration. Using Soft Connections restores modularity, allowing individual modules to be swapped in isolation, independent of the instantiation hierarchy.

---

[1] We do not consider Verilog Out-of-Module References (OOMR) to be a satisfactory solution as they break modular abstraction. Languages such as SystemVerilog raise the level of abstraction so that the user works with typed interfaces instead of wires, but the basic problem remains.

This paper deals with whole-design compilation. Discussion of separately-compiled blocks is omitted for space concerns, but is presented in [1]. Although this paper uses the simulation of microprocessors using FPGAs as an ongoing example, the technique is general and could be used for ASIC design.

2. BACKGROUND: STATIC ELABORATION

Implementing Soft Connections in an existing structural HDL such as Verilog would require either modifying the language or using external scripts to transform the source code. Instead, we implement our Soft Connections scheme in Bluespec SystemVerilog [3], an existing hardware description language.

Bluespec provides a powerful *static elaboration* phase which allows users to transform their design arbitrarily without giving up the static safety a hardware-aware language provides. During elaboration statically known values are aggressively propagated in order to resolve polymorphism and "unroll" static loops and function calls. For example, the designer may describe an *n*-bit ripple-carry adder as follows:

```
function bit[n:0] addRC(bit[n:0] x, bit[n:0] y);
  bit[n:0] res = 0;
  bit c = 0;
  for (int k = n; k >= 0; k) begin
      res[k] = x[k] ^ y[k] ^ c;
      c = (x[k] & y[k]) | (x[k] & c) | (y[k] & c);
  end
  return res;
endfunction
```

The designer may then call this addRC function multiple times using different types. The HDL compiler will execute the function and its loop, using statically known values of *n* and *k*. If *x* and *y* are known statically than the function itself may result in no hardware, but rather a new static constant. However if *x* and *y* are dynamic inputs to the hardware block then the result is a netlist of AND- and XOR-gates. If for some reason *n* was dynamic, the result would be an error as the loop could not be turned into bounded hardware.

HDLs such as Verilog feature elaboration primarily through the use of *generate* blocks, which allow the user to create static control-flow structures such as loops and if-statements. Bluespec expands this into a Turing-complete software interpreter. This allows the user to work with high-level datatypes such as linked-lists or unbounded integers. These types do not have a hardware representation, but the designer can use them to *influence* the hardware that the compiler generates. For example, here is a Bluespec module that takes as input a list of integers. For each one it instantiates a 32-bit FIFO of that depth (note that <- is the module instantiation operator in Bluespec):

```
module mkFIFOList#(List#(Integer) depths);
  let result_list = nil;
  while (depths != nil)  begin
      Integer d = head(depths);
      FIFO#(bit[31:0]) q <- mkSizedFIFO(d);
      result_list = append(result_list, q);
      depths = pop(depths);
  end
  return result_list;
endmodule
```

This use of static elaboration could be thought of as "embedding a small software program in our hardware description source that the compiler runs to generate hardware." Soft Connections represent a novel use of static elaboration, and help to demonstrate how a more powerful notion of elaboration can benefit hardware designers.


```
method Action train(BPredInfo inf);
  if (inf.branchTaken != table.lookup(inf.pc))
    link_to_debug.send(debugMsgMispredict(inf.pc));
    table.update(inf.pc, inf.branchTaken);
endmethod
```



```
rule debugToPCIe;
  let msg = link_from_sender.receive();
  pcie.transmit(pcieRequest(msg));
  link_from_sender.deq();
endrule
```

Figure 3. This branch predictor sends debug information when it is trained with a misprediction. Separately, the Debug module transmits the debug information to software using PCIe.
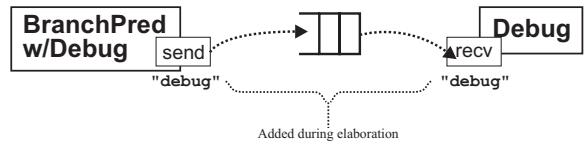

Figure 4. The physical buffering connecting the two endpoints is added during elaboration, as explained in Section 5.

3. SOFT CONNECTIONS
3.1. Point-To-Point Connections

Soft Connections are a library of communication primitives that the designer uses to describe a *logical topology* of communication. The basic Soft Connection is a point-to-point First-In-First-Out channel. This channel is used as if it were a familiar guarded FIFO (Figure 3).

Where Soft Connections differ is the instantiation. Instead of instantiating a single channel module and passing it to both of the users, the communicating modules instantiate the endpoints separately, naming the channel with a unique identifier. For example:

```
let linkToDebug <- mkConnectionSend("debug");
```

Elsewhere, the receiving module instantiates the dual endpoint:

```
let linkFromSender <- mkConnectionRecv("debug");
```

The channel itself is instantiated during elaboration (Figure 4).

As Soft Connections often represent communication between distant modules, we have chosen to implement them using a guarded buffer. Flow-control is handled via Bluespec's standard guarded interface scheme [10], so that the producer's action may not be taken if the buffer is full, nor the consumer's if it is empty.

If our algorithm finds no matching endpoint with the same name, the result is a compilation error. If an error is not desired either endpoint may be specified as optional:

```
let linkFromSender <- mkConnRecvOptional("debug");
```

An optional receiver with no corresponding sender will never receive data. Data can be enqueued to an optional sender with no corresponding receiver but that data will simply disappear. Either are like a wire unterminated on one side - they will have no effect on synthesis results.
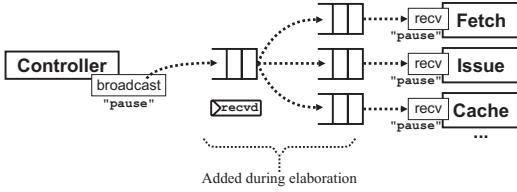
Figure 5. A one-to-many connection. When every receiver has gotten the data the main queue is dequeued. Note that the end-points of the receivers are standard Point-to-Point receives.
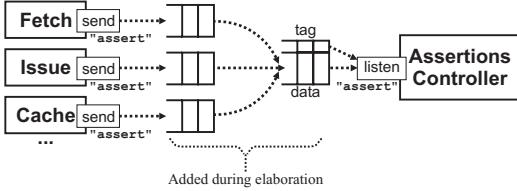

Figure 6. A many-to-one connection. Incoming data is tagged to identify the sender. Note that the senders are standard point-to-point sends.
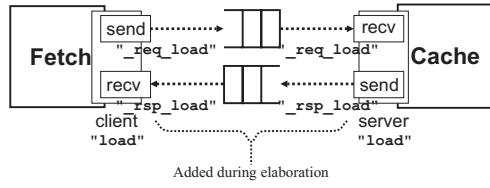

Figure 7. Clients and Servers are abstractions for a bundle of standard send and receive connections.

3.2. One-to-Many and Many-to-One

A one-to-many send is a broadcast that transmits the same data to all listeners (Figure 5). A many-to-one receive is a channel multiplexed by an arbitrator, that also tags the data with a bit field indicating which sender the data comes from (Figure 6). These tags are assigned by our algorithm.

One-to-many connections are useful for relaying control messages from software to many hardware modules—for instance to start, pause, or reset operation. Many-to-one receives are useful for aggregating data such as assertions or debugging information for transmission to software.

3.3 Clients and Servers

The uni-directional channels presented above represent the primitive Soft Connections on which our elaboration algorithm operates. We then use these as building blocks to create useful abstractions for bi-directional communication. The first abstraction is that of a request/response paradigm (Figure 7). The client makes requests and gets responses. The server receives requests and makes responses.

This arrangement is often used to connect functional units to their users. This idea can be combined with one-to-many and many-to-one connections to make multi-user clients and servers. A server with a many-to-one connection can receive requests from multiple clients, and uses many point-to-point connections which deliver responses (Figure 8). The dual of this is a client that is connected to many servers. It broadcasts requests to all of them, then receives the responses in serial. This is a one-to-many send for the requests, and a many-to-one receive for the responses (Figure 9).
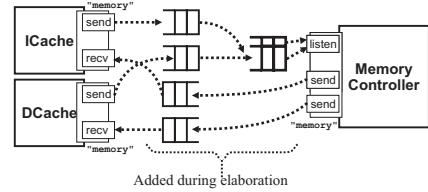
Figure 8. A multi-user server. Note that the clients are standard one-to-one clients.
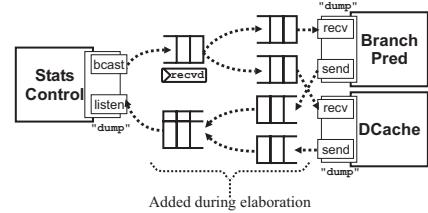
Figure 9. A client connected to multiple servers. Note that the servers are standard one-to-one servers.

3.4. Example: Simulation Controller

The simulation controller presented in Figure 10 represents an example of how Soft Connections can improve designer productivity. The controller is a module that sits on the FPGA and communicates with software on the CPU, mediating interaction with the PCIe link. The controller instantiates six sub-controllers:

- **Commands (Client) :** This receives commands from software such as "start" or "pause" and broadcasts them to listening modules. These modules respond when simulation is finished. Thus this module is a client of many distributed servers.

- **Params (Client):** This receives dynamic parameters set on the command line when the user initiates the software. These parameters are sent to the appropriate listeners. Thus, for example, the cache can be disabled without re-synthesizing the design.

- **Events (Client):** These represent a detailed trace of results from the simulator. Software enables or disable event-dumping dynamically, and these requests are passed on to the modules.

- **Stats (Client):** Periodically the host software can request a dump of statistics. This request is relayed to all listeners, who respond with their current values, which are relayed to the host.

- **Assertions (Listener):** When an assertion fails in a hardware module, it sends a message to this controller, which relays it to software that prints out a message and ends the simulation gracefully.

- **Debug (Listener):** This module listens for debugging messages and relays them to the host software where they are logged.

Using Soft Connections for the communication from these controllers to the simulator modules results in several benefits. First, the designer can fluidly swap modules without rewiring their connection to the controllers. This encourages users to create many variations of their module, without worrying that (for example) a direct-mapped write-through cache contains a smaller set of statistics than an associative write-back cache. Finally, it raises the level of abstraction for the user, who just records stats and assertion failures, without worrying about how this information is communicated to software.

4. PHYSICAL INTERCONNECT SHARING

Soft connections make life easier for the designer by making module communication implicit. The disadvantage of this is that the designer can lose intuition about the implementation cost of their communication network. For example, we have found that the assertions facility is useful for the FPGA in practice. Thus it
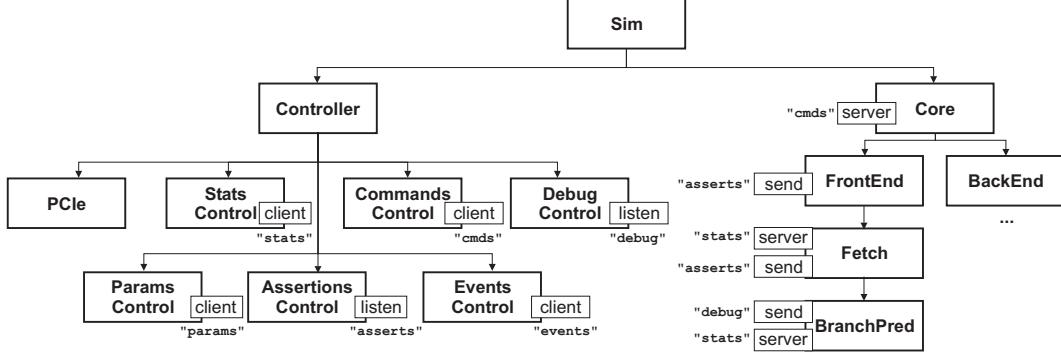
Figure 10. Example: A simulation controller mediates the connection between host software and hardware modules.
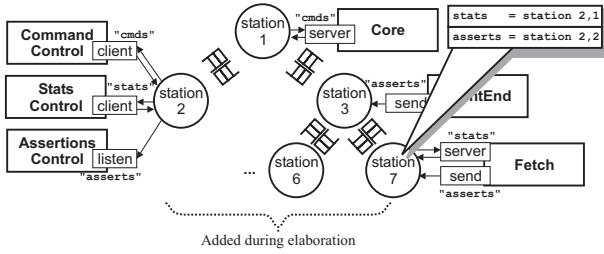
Figure 11. Multiple Soft Connections implemented on a shared physical interconnect. Each station routes logical channels to the appropriate destination using a generated routing table.

becomes frequently used. A typical configuration of our simulator has 42 dynamic assertions, most of them sanity checks relating to correct instruction execution. Implementing these as 42 FIFOs arbitrating directly with the controller is expensive, and places a large burden on the place-and-route tools due to the fan-in. Assertion failures are (hopefully) a rare occurrence, so it makes sense to aggregate them using a multiplexed physical interconnect such as a tree. Such a scheme increases the latency a message takes to reach the endpoint, but can result in more efficient hardware.

Other rarely-used connections such as statistics can also be mapped onto the same interconnect. Thus the user can separate a Soft Connection's *physical representation* (exclusive channel or shared interconnect) from its *logical representation* (point-to-point, one-to-many, etc).

The user creates a shared connection by first instantiating a network station. This station is then passed in to the constructor of the Soft Connection:

```
let fetchStation <- mkStationTree("fetch");
let linkAssert <- mkConnSendShared(fetch_station,
                                   "asserts");
```

Figure 11 shows an example mapping many Soft Connections onto the same shared interconnect. Whether a Soft Connection is implemented as an exclusive or shared interconnect is transparent to the modules which use the endpoints—they use the connection's operations (send, receive, broadcast, etc) as normal. The only difference from the user's point of view is that the latency of communication between sender and receiver has increased, as the messages are in fact being passed over an interconnect which is shared with other endpoints. Our elaboration algorithm connects the stations together into a physical network, and creates a routing table to dynamically guide messages to the appropriate destination. The addressing and routing of messages is handled by the stations themselves.

Algorithm 1. Connecting Soft Connection endpoints directly.

```
 1:   (sends, recvs) = … // Get collected info
 2:   for each s in sends do
 3:       rs = matchByName(s, recvs)
 4:       if rs ={} and not optional(s) then
 5:           error("Unmatched Send " + s)
 6:       else if rs = {r} then
 7:           connect(s, r)  // Instantiate buffering
 8:       else
 9:           connectBroadcast(s, rs) // as in Figure 5
10:       recvs = recvs – rs
11:   for each r in recvs do
12:       error("Unmatched Recv" + r)
```

Currently our algorithm connects the stations into a branching tree topology that follows the module instantiation hierarchy. (Layers in the hierarchy with no stations are optimized away.) This topology was chosen because it maximizes spatial locality by keeping the stations near their endpoints, and because it results in a single static route between two given endpoints, which minimizes station routing logic. In the future, support is planned for other physical network topologies such as rings, two-way rings, or grids.

5. CONNECTION ALGORITHM

When a module instantiates a Soft Connection endpoint it is implicitly transforming the interface it presents to the outside world. For a module with interface i its new interface i' is a tuple of i plus linked lists that describe what Soft Connection endpoints the module has instantiated:

$$i' = (i, \{sends\}, \{recvs\})$$

The module's parent (and the parent's parent) see only the original interface i. This, along with collecting all the lists from all of the modules, is accomplished using a standard Bluespec library called `ModuleCollect`.

Algorithm 1 describes our process for connecting Soft Connection endpoints directly. For space reasons we omit many-to-one connections, which work similarly. Connections that are unmatched (and not optional) result in a compilation error via Bluespec's built-in `error` function, which halts elaboration.

The algorithm for instantiating Soft Connections sharing a physical interconnect is most naturally described as a recursive module—it may call itself during elaboration, resulting in a tree-topology of stations connected to each other:

Algorithm 2. Constructing a station's routing table

```
1:    let (childs, sends, recvs) = ... // Parameters
2:    // Routing decisions for traffic from local sends.
3:    for each s in sends do
4:        if matchByName(s, childs) = {c} then
5:            // A child (or its descendant) has the recv.
6:            sendRoute[s] := toChild c
7:        else // The endpoint is not in this subtree.
8:            sendRoute[s] := toParent
9:    // Routing decisions for traffic from children.
10:   for each c in childs do
11:       // Find all sends this child is routing up to us.
12:       for each s in sendsRoutedToParent(c) do
13:           if matchInStation(s, childs) = {c₂} then
14:               // This station is the least-common ancestor.
15:               childRoute[c][s] := toChild c₂
16:           else if matchByName(s, recvs) = {r} then
17:               // The endpoint is local to this station.
18:               childRoute[c][s] := toRecv r
19:           else // The endpoint is not in this subtree.
20:               childRoute[c][s] := toParent
```

```
module mkStationTree#(STATION_INFO info)(STATION);
  List#(STATION) child_stations = nil;
  for (int x = 0; x < length(info.children); x++)
  begin
    let cur_child = info.children[x];
    // Recurse down the tree.
    let c <- mkStationTree(cur_child);
    child_stations = append(child_stations, c);
  end
  let table <- mkRoutingTable(child_stations,
                      info.recvs, info.sends);
  let s <- connectStation(table, child_stations,
                      info.recvs, info.sends);

  return s;
endmodule
```

The routing table is constructed mechanically using Algorithm 2. We have omitted the details of routing one-to-many sends for space concerns. They have the potential to be sent to multiple receivers and children. Additionally, they are always routed up to the parent (which drops the message if none of its other children are receivers). Endpoints that are unmatched at the root station result in an error, as in the unshared case.

6. ASSESSMENT
6.1. Impact on Productivity

In this section we examine a real-world example in order to give some insight into how Soft Connections can improve the process of engineering an FPGA-based accelerator. For the example we have chosen an FPGA-based model of a 5-stage microprocessor pipeline that runs the Alpha instruction set using the HAsim simulator [2].

As shown in Figure 11, the FPGA is configured into a simulator of this target machine. This simulator bears little resemblance to the 5-stage pipeline itself, but accurately computes the performance of the target. This is because the timing properties of the physical implementation—such as the FPGA BlockRAM or the speed of memory through the PCIe—are different from the speeds in the machine we wish to study. Thus we add logic to translate FPGA cycles into model cycles in the target. The simulator is divided into three major partitions: model timing, model functionality, and the simulation controller. The full technique for creating such a simulator is presented in [2].

We synthesized our simulator for a Virtex5 110t part on a PCIe board manufactured by HiTechGlobal [8] using Xilinx ISE 10.1:

Slice LUTs	47214/69120 (68%)
BlockRAM	121/148 (81%)
Critical Path	15.313 ns
Frequency	65 MHZ

It may seem surprising that modeling a simple architecture would use so many FPGA resources. This is because the simulator uses a large number of FPGA resources as on-chip cache. As we have divorced FPGA time from model time, our simulator can devote an arbitrary amount of on-chip memory to cache, even if the target has a smaller cache. This extra cache speeds up simulation, but has no effect on the behavior of the target machine. (In some sense, any unused slice is a wasted resource for an accelerator FPGA.)

Figure 12 gives an overview of how our design uses Soft Connections. We have attempted to quantify the productivity these provide by defining a metric called span. For each connection c between two modules:

> $span(c)$ = the number of module instantiation boundaries between the send and receive endpoints.

Span measures the potential work the Soft Connection is saving the designer. Namely, the number of modules that the designer would have to change if she was not using Soft Connections and swapped in a module with a different interface. We acknowledge the limitations of measuring the amount of work that our technique *potentially* can save, but believe that this metric gives valuable insight into the degree that communication between distant endpoints can exist in a hardware design.

Figure 13 shows a histogram of the span of every connection in our simulator—i.e., our simulator contains 74 connections with a span of 7. Spans of 0 represent optional connections which are not being used. We found that the average Soft Connection in our simulator crosses 5.27 module instantiations, and that 50% of them cross 7 or more. This demonstrates that cross-hierarchical communication can be prevalent in real-world situations.

6.2. The Effect of Shared Interconnects

Much of the cross-hierarchical communication—and all of the many-to-one/one-to-many connections—involve communicating data to or from the Simulation Controller (Section 3.4). The cost of multiplexing between these signals can be high, and can result in a burden on the place and route tools. In order to explore this we implemented an alternative version of our simulator where all connections to the controller shared the same interconnect tree.

Overall 100/217 connections were mapped onto this tree, representing the statistics, assertions, commands, parameters, and events facilities. The tree had 14 stations arranged into a depth of 4, with the controller as the root node. All told, this tree spanned 20 module instantiations. We found this version consumes an additional 3076 slice LUTs (4% of total available) because of its extra buffering and routing tables. RAM utilization and clock speed are not affected, as the critical path is elsewhere.

Multiplexing these connections onto the same tree can increase the latency of communication. To measure the impact of this on
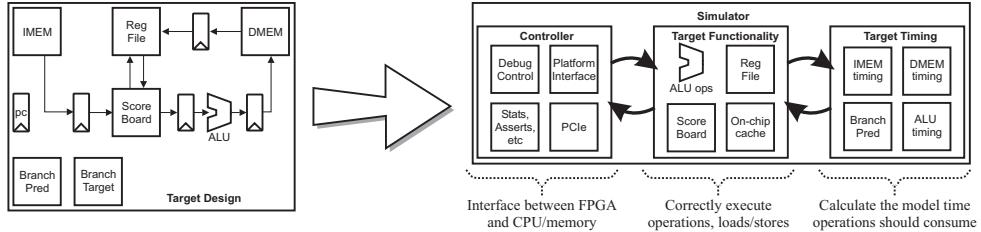
Figure 11. Using the HAsim simulator to model an inorder microprocessor pipeline. The FPGA is not configured into the target pipeline itself, but into three partitions which interoperate to model the pipeline at the fastest rate possible. Only the timing partition must be changed to represent the specifics of the target pipeline. The controller and functional partition can be reused across targets [2].

Category	Number
Intra-Timing	33
Intra-Functional	19
Intra-Controller	20
Timing-Functional	24
Timing-Controller	42
Functional-Controller	76
Unused Optional	3
Total	217

Figure 12. Number and use of Soft Connections in the HAsim inorder pipeline model.
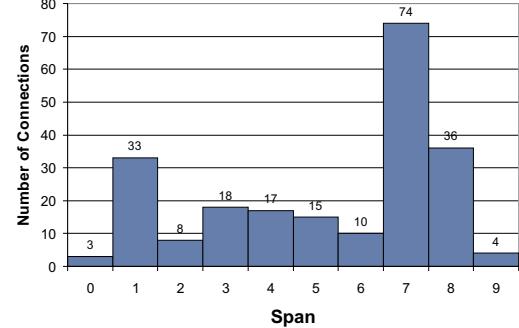
Figure 13. Histogram of Soft Connection span.

Benchmark	Model Cycles	FPGA Cycles: Baseline	FPGA Cycles: Shared	Change
test 164 gzip	7,612,202,736	120,866,746,639	120,848,407,550	-.0002%
test 176 gcc	4,412,926,919	97,284,304,169	96,331,305,044	-.001%
test 181 mcf	515,321,465	13,393,128,486	13,375,809,359	-.001%

Figure 14. Running SPEC benchmarks on the shared interconnect version.

dynamic performance we ran three SPEC benchmarks on each model. The results, shown in Figure 14 demonstrate that over a run which spans billions of model cycles there was no measurable impact on performance—the differences in total FPGA cycles fall within expected run-to-run variation.

7. DISCUSSION

Soft Connections are currently implemented as direct point-to-point connections, or as a shared tree topology. In the future we plan to explore adding support for new physical communication topologies such as rings or grids. We expect grid networks to present a particular challenge as the stations need awareness of the dynamic traffic conditions in order to route messages efficiently. We believe that insights from networks-on-chip—which are traditionally used to connect distinct hardware cores together—may also apply to distributing the connections within the cores themselves.

As FPGA accelerators become more common the barrier to entry becomes a large concern. Traditional tools can force the designer to spend too much effort thinking about on-chip communication and not enough time thinking about the actual logic. Soft Connections are a way to automatically generate a physical implementation of communication from its logical specification. This provides a richer module interface and makes the communication hierarchy less rigid. We believe that these kind of ease-of-use efforts will be critical for FPGAs to gain acceptance as computation accelerators in general-purpose computers.

REFERENCES

[1] M. Pellauer, M. Adler, J. Emer, Modular Soft Connections. Computation Structures Group Technical Report #505, MIT.

[2] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind and J Emer. Quick Performance Models Quickly: Timing-Directed Simulation on FPGAs. In Proceedings of ISPASS, 2008.

[3] Bluespec Inc. http://www.bluespec.com/, 2008.

[4] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated simulation technologies FAST: Fast, full-system, cycle-accurate simulators. In MICRO, 2007.

[5] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu. The FAST methodology for high-speed SoC/Computer simulation. In Proceedings of ICCAD, 2007.

[6] E. Chung, E. Nurvitadhi, J. Hoe K. Mai, and B. Falsafi. Accelerating Architectural-level, Full-System Multiprocessor Simulations using FPGAs. In FPGA '08: 11th International Symposium on Field Programmable Gate Arrays, 2008.

[7] Nallatech, Inc. http://www.nallatech.com/, 2009.

[8] HiTech Global, LLC. http://www.hitechglobal.com/, 2009.

[9] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. APorts: an efficient abstraction for cycle-accurate performance models on FPGAs. In FPGA '08: 11th International Symposium on Field Programmable Gate Arrays, 2008.

[10] D. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In Proceedings of DAC, San Diego, CA, 2004.

[11] J. Wawrzynek, D. Patterson, M. Oskin, S. L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: a research accelerator for multiple processors. IEEE Micro, Mar/Apr 2007.