A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services

Andrew Putnam Adrian M. Caulfield Eric S. Chung Derek Chiou[1]
Kypros Constantinides[2] John Demme[3] Hadi Esmaeilzadeh[4] Jeremy Fowers
Gopi Prashanth Gopal Jan Gray Michael Haselman Scott Hauck[5] Stephen Heil
Amir Hormati[6] Joo-Young Kim Sitaram Lanka James Larus[7] Eric Peterson
Simon Pope Aaron Smith Jason Thong Phillip Yi Xiao Doug Burger

Microsoft

Abstract

Datacenter workloads demand high computational capabilities, flexibility, power efficiency, and low cost. It is challenging to improve all of these factors simultaneously. To advance datacenter capabilities beyond what commodity server designs can provide, we have designed and built a composable, reconfigurable fabric to accelerate portions of large-scale software services. Each instantiation of the fabric consists of a 6x8 2-D torus of high-end Stratix V FPGAs embedded into a half-rack of 48 machines. One FPGA is placed into each server, accessible through PCIe, and wired directly to other FPGAs with pairs of 10 Gb SAS cables.

In this paper, we describe a medium-scale deployment of this fabric on a bed of 1,632 servers, and measure its efficacy in accelerating the Bing web search engine. We describe the requirements and architecture of the system, detail the critical engineering challenges and solutions needed to make the system robust in the presence of failures, and measure the performance, power, and resilience of the system when ranking candidate documents. Under high load, the large-scale reconfigurable fabric improves the ranking throughput of each server by a factor of 95% for a fixed latency distribution—or, while maintaining equivalent throughput, reduces the tail latency by 29%.

1. Introduction

The rate at which server performance improves has slowed considerably. This slowdown, due largely to power limitations, has severe implications for datacenter operators, who have traditionally relied on consistent performance and efficiency improvements in servers to make improved services economically viable. While specialization of servers for specific scale workloads can provide efficiency gains, it is problematic for two reasons. First, homogeneity in the datacenter is highly desirable to reduce management issues and to provide a consistent platform that applications can rely on. Second, datacenter services evolve extremely rapidly, making non-programmable hardware features impractical. Thus, datacenter providers are faced with a conundrum: they need continued improvements in performance and efficiency, but cannot obtain those improvements from general-purpose systems.

Reconfigurable chips, such as Field Programmable Gate Arrays (FPGAs), offer the potential for flexible acceleration of many workloads. However, as of this writing, FPGAs have not been widely deployed as compute accelerators in either datacenter infrastructure or in client devices. One challenge traditionally associated with FPGAs is the need to fit the accelerated function into the available reconfigurable area. One could virtualize the FPGA by reconfiguring it at run-time to support more functions than could fit into a single device. However, current reconfiguration times for standard FPGAs are too slow to make this approach practical. Multiple FPGAs provide scalable area, but cost more, consume more power, and are wasteful when unneeded. On the other hand, using a single small FPGA per server restricts the workloads that may be accelerated, and may make the associated gains too small to justify the cost.

This paper describes a reconfigurable fabric (that we call Catapult for brevity) designed to balance these competing concerns. The Catapult fabric is embedded into each half-rack of 48 servers in the form of a small board with a medium-sized FPGA and local DRAM attached to each server. FPGAs are directly wired to each other in a 6x8 two-dimensional torus, allowing services to allocate groups of FPGAs to provide the necessary area to implement the desired functionality.

We evaluate the Catapult fabric by offloading a significant fraction of Microsoft Bing's ranking stack onto groups of eight FPGAs to support each instance of this service. When a server wishes to score (rank) a document, it performs the software portion of the scoring, converts the document into a format suitable for FPGA evaluation, and then injects the document to its local FPGA. The document is routed on the inter-FPGA network to the FPGA at the head of the ranking pipeline. After running the document through the eight-FPGA pipeline, the computed score is routed back to the requesting server. Although we designed the fabric for general-purpose service

[1]Microsoft and University of Texas at Austin
[2]Amazon Web Services
[3]Columbia University
[4]Georgia Institute of Technology
[5]Microsoft and University of Washington
[6]Google, Inc.
[7]École Polytechnique Fédérale de Lausanne (EPFL)
All authors contributed to this work while employed by Microsoft.

978-1-4799-4394-4/14/$31.00

acceleration, we used web search to drive its requirements, due to both the economic importance of search and its size and complexity. We set a performance target that would be a significant boost over software—2x throughput in the number of documents ranked per second per server, including portions of ranking which are not offloaded to the FPGA.

One of the challenges of maintaining such a fabric in the datacenter is resilience. The fabric must stay substantially available in the presence of errors, failing hardware, reboots, and updates to the ranking algorithm. FPGAs can potentially corrupt their neighbors or crash the hosting servers during bitstream reconfiguration. We incorporated a failure handling protocol that can reconfigure groups of FPGAs or remap services robustly, recover from failures by remapping FPGAs, and report a vector of errors to the management software to diagnose problems.

We tested the reconfigurable fabric, search workload, and failure handling service on a bed of 1,632 servers equipped with FPGAs. The experiments show that large gains in search throughput and latency are achievable using the large-scale reconfigurable fabric. Compared to a pure software implementation, the Catapult fabric achieves a 95% improvement in throughput at each ranking server with an equivalent latency distribution—or at the same throughput, reduces tail latency by 29%. The system is able to run stably for long periods, with a failure handling service quickly reconfiguring the fabric upon errors or machine failures. The rest of this paper describes the Catapult architecture and our measurements in more detail.

2. Catapult Hardware

The acceleration of datacenter services imposes several stringent requirements on the design of a large-scale reconfigurable fabric. First, since datacenter services are typically large and complex, a large amount of reconfigurable logic is necessary. Second, the FPGAs must fit within the datacenter architecture and cost constraints. While reliability is important, the scale of the datacenter permits sufficient redundancy that a small rate of faults and failures is tolerable.

To achieve the required capacity for a large-scale reconfigurable fabric, one option is to incorporate multiple FPGAs onto a daughtercard and house such a card along with a subset of the servers. We initially built a prototype in this fashion, with six Xilinx Virtex 6 SX315T FPGAs connected in a mesh network through the FPGA's general-purpose I/Os. Although straightforward to implement, this solution has four problems. First, it is inelastic: if more FPGAs are needed than there are on the daughtercard, the desired service cannot be mapped. Second, if fewer FPGAs are needed, there is stranded capacity. Third, the power and physical space for the board cannot be accommodated in conventional ultra-dense servers, requiring either heterogeneous servers in each rack, or a complete redesign of the servers, racks, network, and power distribution. Finally, the large board is a single point of failure, whose failure would result in taking down the entire subset of servers.

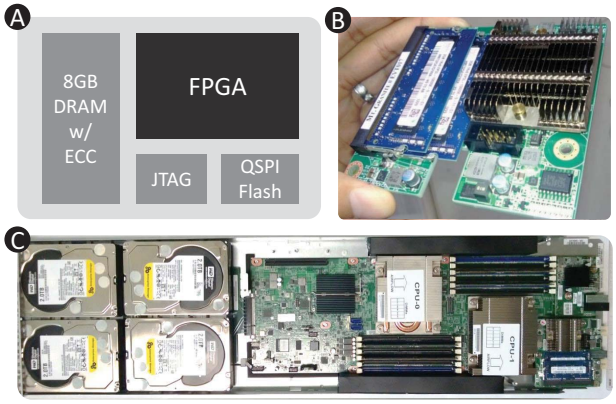

Figure 1: (a) A block diagram of the FPGA board. (b) A picture of the manufactured board. (c) A diagram of the 1 U, half-width server that hosts the FPGA board. The air flows from the left to the right, leaving the FPGA in the exhaust of both CPUs.

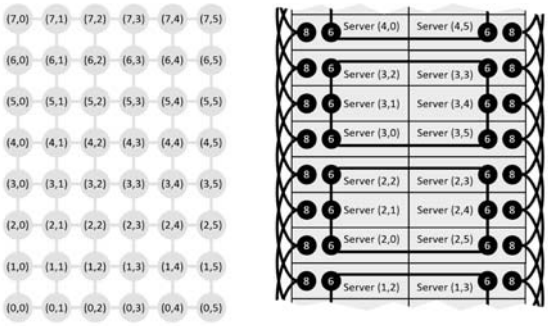

Figure 2: The logical mapping of the torus network, and the physical wiring on a pod of 2 x 24 servers.

The alternative approach we took places a small daughtercard in each server with a single high-end FPGA, and connects the cards directly together with a secondary network. Provided that the latency on the inter-FPGA network is sufficiently low, and that the bandwidth is sufficiently high, services requiring more than one FPGA can be mapped across FPGAs residing in multiple servers. This elasticity permits efficient utilization of the reconfigurable logic, and keeps the added acceleration hardware within the power, thermal, and space limits of dense datacenter servers. To balance the expected per-server performance gains versus the necessary increase in total cost of ownership (TCO), including both increased capital costs and operating expenses, we set aggressive power and cost goals. Given the sensitivity of cost numbers on elements such as production servers, we cannot give exact dollar figures; however, adding the Catapult card and network to the servers did not exceed our limit of an increase in TCO of 30%, including a limit of 10% for total server power.

2.1. Board Design

To minimize disruption to the motherboard, we chose to interface the board to the host CPU over PCIe. While a tighter coupling of the FPGA to the CPU would provide benefits in

terms of latency, direct access to system memory, and potentially coherence, the selection of PCIe minimized disruption to this generation of the server design. Since the FPGA resides in I/O space, the board needed working memory to accommodate certain services. We chose to add local DRAM, as SRAM QDR arrays were too expensive to achieve sufficient capacity. 8 GB of DRAM was sufficient to map the services we had planned, and fit within our power and cost envelopes.

Figure 1 shows a logical diagram of the FPGA board along with a picture of the manufactured board and the server it installs into [20]. We chose a high-end Altera Stratix V D5 FPGA [3], which has considerable reconfigurable logic, on-chip memory blocks, and DSP units. The 8 GB of DRAM consists of two dual-rank DDR3-1600 SO-DIMMs, which can operate at DDR3-1333 speeds with the full 8 GB capacity, or trade capacity for additional bandwidth by running as 4 GB single-rank DIMMs at DDR3-1600 speeds. The PCIe and inter-FPGA network traces are routed to a mezzanine connector on the bottom of the daughtercard, which plugs directly into a socket on the motherboard. Other components on the board include a programmable oscillator and 32 MB of Quad SPI flash to hold FPGA configurations. Because of the limited physical size of the board and the number of signals that must be routed, we used a 16-layer board design. Our target applications would benefit from increased memory bandwidth, but there was insufficient physical space to add additional DRAM channels. We chose to use DIMMs with ECC to add resilience as DRAM failures are commonplace at datacenter scales.

Figure 1(c) shows the position of the board in one of the datacenter servers. We used the mezzanine connector at the back of the server so that heat from the FPGA did not disrupt the existing system components. Since the FPGA is subject to the air being heated by the host CPUs, which can reach 68 °C, we used an industrial-grade FPGA part rated for higher-temperature operation up to 100 °C. It was also necessary to add EMI shielding to the board to protect other server components from interference from the large number of high-speed signals on the board. One requirement for serviceability was that no jumper cables should be attached to the board (e.g., power or signaling). By limiting the power draw of the daughtercard to under 25 W, the PCIe bus alone provided all necessary power. By keeping the power draw to under 20 W during normal operation, we met our thermal requirements and our 10% limit for added power.

2.2. Network Design

The requirements for the inter-FPGA network were low latency and high bandwidth to meet the performance targets, low component costs, plus only marginal operational expense when servicing machines. The rack configuration we target is organized into two half-racks called pods. Each pod has its own power distribution unit and top-of-rack switch. The pods are organized in a 24 U arrangement of 48 half-width 1 U servers (two servers fit into each 1 U tray).

Based on our rack configuration, we selected a two-dimensional, 6x8 torus for the network topology. This arrangement balanced routability and cabling complexity. Figure 2 shows how the torus is mapped onto a pod of machines. The server motherboard routes eight high-speed traces from the mezzanine connector to the back of the server chassis, where the connections plug into a passive backplane. The traces are exposed on the backplane as two SFF-8088 SAS ports. We built custom cable assemblies (shells of eight and six cables) that plugged into each SAS port and routed two high-speed signals between each pair of connected FPGAs. At 10 Gb/s signaling rates, each inter-FPGA network link supports 20 Gb/s of peak bidirectional bandwidth at sub-microsecond latency, with no additional networking costs such as NICs or switches.

Since the server sleds are plugged into a passive backplane, and the torus cabling also attaches to the backplane, a server can be serviced by pulling it out of the backplane without unplugging any cables. Thus, the cable assemblies can be installed at rack integration time, tested for topological correctness, and delivered to the datacenter with correct wiring and low probability of errors when servers are repaired.

2.3. Datacenter Deployment

To test this architecture on a number of datacenter services at scale, we manufactured and deployed the fabric in a production datacenter. The deployment consisted of 34 populated pods of machines in 17 racks, for a total of 1,632 machines. Each server uses an Intel Xeon 2-socket EP motherboard, 12-core Sandy Bridge CPUs, 64 GB of DRAM, and two SSDs in addition to four HDDs. The machines have a 10 Gb network card connected to a 48-port top-of-rack switch, which in turn connects to a set of level-two switches.

The daughtercards and cable assemblies were both tested at manufacture and again at system integration. At deployment, we discovered that 7 cards (0.4%) had a hardware failure, and that one of the 3,264 links (0.03%) in the cable assemblies was defective. Since then, over several months of operation, we have seen no additional hardware failures.

3. Infrastructure and Platform Architecture

Supporting an at-scale deployment of reconfigurable hardware requires a robust software stack capable of detecting failures while providing a simple and accessible interface to software applications. If developers have to worry about low-level FPGA details, including drivers and system functions (e.g., PCIe), the platform will be difficult to use and rendered incompatible with future hardware generations. There are three categories of infrastructure that must be carefully designed to enable productive use of the FPGA: (1) APIs for interfacing software with the FPGA, (2) interfaces between FPGA application logic and board-level functions, and (3) support for resilience and debugging.

3.1. Software Interface

Applications targeting the Catapult fabric share a common driver and user-level interface. The communication interface between the CPU and FPGA must satisfy two key design goals: (1) the interface must incur low latency, taking fewer than 10 μs for transfers of 16 KB or less, and (2) the interface must be safe for multithreading. To achieve these goals, we developed a custom PCIe interface with DMA support.

In our PCIe implementation, low latency is achieved by avoiding system calls. We allocate one input and one output buffer in non-paged, user-level memory and supply the FPGA with a base pointer to the buffers' physical memory addresses. Thread safety is achieved by dividing the buffer into 64 *slots*, where each slot is 1/64th of the buffer, and by statically assigning each thread exclusive access to one or more slots. In the case study in Section 4, we use 64 slots of 64 KB each.

Each slot has a set of status bits indicating whether the slot is full. To send data to the FPGA, a thread fills its slot with data, then sets the appropriate full bit for that slot. The FPGA monitors the full bits and fairly selects a candidate slot for DMA'ing into one of two staging buffers on the FPGA, clearing the full bit once the data has been transferred. Fairness is achieved by taking periodic snapshots of the full bits, and DMA'ing all full slots before taking another snapshot of the full bits. When the FPGA produces results for readback, it checks to make sure that the output slot is empty and then DMAs the results into the output buffer. Once the DMA is complete, the FPGA sets the full bit for the output buffer and generates an interrupt to wake and notify the consumer thread.

To configure the fabric with a desired function, user level services may initiate FPGA reconfigurations through calls to a low-level software library. When a service is deployed, each server is designated to run a specific application on its local FPGA. The server then invokes the reconfiguration function, passing in the desired bitstream as a parameter.

3.2. Shell Architecture

In typical FPGA programming environments, the user is often responsible for developing not only the application itself but also building and integrating system functions required for data marshaling, host-to-FPGA communication, and inter-chip FPGA communication (if available). System integration places a significant burden on the user and can often exceed the effort needed to develop the application itself. This development effort is often not portable to other boards, making it difficult for applications to work on future platforms.

Motivated by the need for user productivity and design re-usability when targeting the Catapult fabric, we logically divide all programmable logic into two partitions: the *shell* and the *role*. The *shell* is a reusable portion of programmable logic common across applications—while the *role* is the application logic itself, restricted to a large fixed region of the chip.

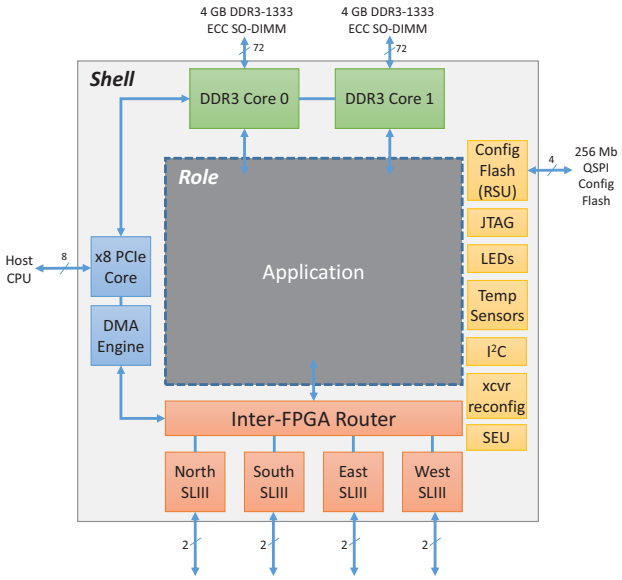

Figure 3: Components of the Shell Architecture.

Role designers access convenient and well-defined interfaces and capabilities in the shell (e.g., PCIe, DRAM, routing, etc.) without concern for managing system correctness. The shell consumes 23% of each FPGA, although extra capacity can be obtained by discarding unused functions. In the future, partial reconfiguration would allow for dynamic switching between roles while the shell remains active—even routing inter-FPGA traffic while a reconfiguration is taking place.

Figure 3 shows a block-level diagram of the shell architecture, consisting of the following components:

- Two DRAM controllers, which can be operated independently or as a unified interface. On the Stratix V, our dual-rank DIMMs operate at 667 MHz. Single-rank DIMMs (or only using one of the two ranks of a dual-rank DIMM) can operate at 800 MHz.
- Four high-speed serial links running SerialLite III (SL3), a lightweight protocol for communicating with neighboring FPGAs. It supports FIFO semantics, Xon/Xoff flow control, and ECC.
- Router logic to manage traffic arriving from PCIe, the role, or the SL3 cores.
- Reconfiguration logic, based on a modified Remote Status Update (RSU) unit, to read/write the configuration Flash.
- The PCIe core, with the extensions to support DMA.
- Single-event upset (SEU) logic, which periodically scrubs the FPGA configuration state to reduce system or application errors caused by soft errors.

The router is a standard crossbar that connects the four inter-FPGA network ports, the PCIe controller, and the application role. The routing decisions are made by a static software-configured routing table that supports different routing policies. The transport protocol is virtual cut-through with no retransmission or source buffering.

Since uncorrected bit errors can cause high-level disruptions (requiring intervention from global management software), we employ double-bit error detection and single-bit error correction on our DRAM controllers and SL3 links. The use of ECC on our SL3 links incurs a 20% reduction in peak bandwidth. ECC on the SL3 links is performed on individual flits, with correction for single-bit errors and detection of double-bit errors. Flits with three or more bit errors may proceed undetected through the pipeline, but are likely to be detected at the end of packet transmission with a CRC check. Double-bit errors and CRC failures result in the packet being dropped and not returned to the host. In the event of a dropped packet, the host will time out and divert the request to a higher-level failure handling protocol.

The SEU scrubber runs continuously to scrub configuration errors. If the error rates can be brought sufficiently low, with conservative signaling speeds and correction, the rare errors can be handled by the higher levels of software, without resorting to expensive approaches such as source-based retransmission or store-and-forward protocols. The speed of the FPGAs and the ingestion rate of requests is high enough that store-and-forward would be too expensive for the applications that we have implemented.

3.3. Software Infrastructure

The system software, both at the datacenter level and in each individual server, required several changes to accommodate the unique aspects of the reconfigurable fabric. These changes fall into three categories: ensuring correct operation, failure detection and recovery, and debugging.

Two new services are introduced to implement this support. The first, called the Mapping Manager, is responsible for configuring FPGAs with the correct application images when starting up a given datacenter service. The second, called the Health Monitor, is invoked when there is a suspected failure in one or more systems. These services run on servers within the pod and communicate through the Ethernet network.

3.4. Correct Operation

The primary challenge we found to ensuring correct operation was the potential for instability in the system introduced by FPGAs reconfiguring while the system was otherwise up and stable. These problems manifested along three dimensions. First, a reconfiguring FPGA can appear as a failed PCIe device to the host, raising a non-maskable interrupt that may destabilize the system. Second, a failing or reconfiguring FPGA may corrupt the state of its neighbors across the SL3 links by randomly sending traffic that may appear valid. Third, reconfiguration cannot be counted on to occur synchronously across servers, so FPGAs must remain robust to traffic from neighbors with incorrect or incompatible configurations (e.g. "old" data from FPGAs that have not yet been reconfigured).

The solution to a reconfiguring PCIe device is that the driver that sits behind the FPGA reconfiguration call must first disable non-maskable interrupts for the specific PCIe device (the FPGA) during reconfiguration.

The solution to the corruption of a neighboring FPGA during reconfiguration is more complex. When remote FPGAs are reconfigured, they may send garbage data. To prevent this data from corrupting neighboring FPGAs, the FPGA being reconfigured sends a "TX Halt" message, indicating that the neighbors should ignore all further traffic until the link is reestablished. In addition, messages are delayed a few clock cycles so that, in case of an unexpected link failure, it can be detected and the message can be suppressed.

Similarly, when an FPGA comes out of reconfiguration, it cannot trust that its neighbors are not sending garbage data. To handle this, each FPGA comes up with "RX Halt" enabled, automatically throwing away any message coming in on the SL3 links. The Mapping Manager tells each server to release RX Halt once all FPGAs in a pipeline have been configured.

3.5. Failure Detection and Recovery

When a datacenter application hangs for any reason, a machine at a higher level in the service hierarchy (such as a machine that aggregates results) will notice that a set of servers are unresponsive. At that point, the Health Monitor is invoked. The Health Monitor queries each machine to find its status. If a server is unresponsive, it is put through a sequence of soft reboot, hard reboot, and then flagged for manual service and possible replacement, until the machine starts working correctly. If the server is operating correctly, it responds to the Health Monitor with information about the health of its local FPGA and associated links. The Health Monitor returns a vector with error flags for inter-FPGA connections, DRAM status (bit errors and calibration failures), errors in the FPGA application, PLL lock issues, PCIe errors, and the occurrence of a temperature shutdown. This call also returns the machine IDs of the north, south, east, and west neighbors of an FPGA, to test whether the neighboring FPGAs in the torus are accessible and that they are the machines that the system expects (in case the cables are miswired or unplugged).

Based on this information, the Health Monitor may update a failed machine list (including the failure type). Updating the machine list will invoke the Mapping Manager, which will determine, based on the failure location and type, where to relocate various application roles on the fabric. It is possible that relocation is unnecessary, such as when the failure occurred on a spare node, or when simply reconfiguring the FPGA inplace is sufficient to resolve the hang. The Mapping Manager then goes through its reconfiguration process for every FPGA involved in that service—clearing out any corrupted state and mapping out any hardware failure or a recurring failure with an unknown cause. In the current fabric running accelerated search, failures have been exceedingly rare; we observed no hangs due to data corruption; the failures that we have seen have been due to transient phenomena, primarily machine reboots due to maintenance or other unresponsive services.

3.6. Debugging Support

In a large-scale datacenter deployment, hardware bugs or faults inevitably occur at scale that escape testing and functional validation. Diagnosing these scenarios often requires visibility into the state of the hardware leading up to the point of failure. The use of traditional interactive FPGA debugging tools at scale (e.g., Altera SignalTap, Xilinx ChipScope) is limited by (1) finite buffering capacity, (2) the need to automatically recover the failed service, and (3) the impracticality of putting USB JTAG units into each machine.

To overcome these issues, we embed a lightweight "always-on" Flight Data Recorder that captures only salient information about the FPGA during run-time into on-chip memory that can be streamed out (via PCIe) at any time during the health status check. The information kept in the FDR allows us to verify at scale that FPGAs' power-on sequences were correct (e.g., SL3 links locked properly, PLLs and resets correctly sequenced, etc.) and that there were no intermittent errors.

In addition, the FDR maintains a circular buffer that records the most recent head and tail flits of all packets entering and exiting the FPGA through the router. This information includes: (1) a trace ID that corresponds to a specific compressed document that can be replayed in a test environment, (2) the size of the transaction, (3) the direction of travel (e.g., north-to-south link), and (4) other miscellaneous states about the system (e.g., non-zero queue lengths).

Although the FDR can only capture a limited window (512 recent events), it was surprisingly effective during late-stage deployment and enabled us to diagnose and resolve problems that only manifested at scale such as: (1) rare deadlock events on an 8-stage FPGA pipeline, (2) untested inputs that resulted in hangs in the stage logic, (3) intermittent server reboots, and (4) unreliable SL3 links. In the future, we plan to extend the FDR to perform compression of log information and to opportunistically buffer into DRAM for extended histories.

4. Application Case Study

To drive the requirements of our hardware platform, we ported a significant fraction of Bing's ranking engine onto the Catapult fabric. We programmed the FPGA portion of the ranking engine by hand in Verilog, and partitioned it across seven FPGAs—plus one spare for redundancy. Thus, the engine maps to rings of eight FPGAs on one dimension of the torus.

Our implementation produces results that are identical to software (even reproducing known bugs), with the exception of uncontrollable incompatibilities, such as floating-point rounding artifacts caused by out-of-order operations. Although there were opportunities for further FPGA-specific optimizations, we decided against implementing them in favor of maintaining consistency with software.

Bing search has a number of stages, many outside the scope of our accelerated ranking service. As search queries arrive at the datacenter, they are checked to see if they hit in a front-end cache service. If a request misses in the cache, it is routed to a top-level aggregator (TLA) that coordinates the processing of the query and aggregates the final result. The TLA sends the same query (through mid-level aggregators) to a large number of machines performing a *selection* service that finds documents (web pages) that match the query, and narrows them down to a relatively small number per machine. Each of those high-priority documents and its query are sent to a separate machine running the *ranking* service, (the portion that we accelerate with FPGAs) that assigns each document a score relative to the query. The scores and document IDs are returned to the TLA that sorts them, generates the captions, and returns the results.

The ranking service is performed as follows. When a query+document arrives at a ranking service server, the server retrieves the document and its metadata, which together is called a *metastream*, from the local solid-state drive. The document is processed into several sections, creating several metastreams. A "hit vector", which describes the locations of query words in each metastream, is computed. A tuple is created for each word in the metastream that matches a query term. Each tuple describes the relative offset from the previous tuple (or start of stream), the matching query term, and a number of other properties.

Many "features", such as the number of times each query word occurs in the document, are then computed. Synthetic features, called free-form expressions (FFEs) are computed by arithmetically combining computed features. All the features are sent to a machine-learned model that generates a score. That score determines the document's position in the overall ranked list of documents returned to the user.

We implemented most of the feature computations, all of the free-form expressions, and all of the machine-learned model on FPGAs. What remains in software is the SSD lookup, the hit vector computation, and a small number of software-computed features.

4.1. Software Interface

While the ranking service processes {document, query} tuples, transmitting only a compressed form of the document saves considerable bandwidth. Each encoded {document, query} request sent to the fabric contains three sections: (i) a header with basic request parameters, (ii) the set of software-computed features, and (iii) the hit vector of query match locations for each document's metastreams.

The header contains a number of necessary additional fields, including the location and length of the hit vector, the software-computed features, document length, and number of query terms. The software-computed features section contains one or more pairs of {feature id, feature value} tuples for features which are either not yet implemented on the FPGA, or do not make sense to implement in hardware (such as document features which are independent of the query and are stored within the document).

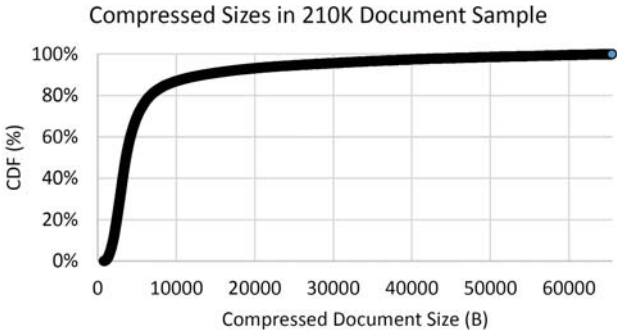

Figure 4: Cumulative distribution of compressed document sizes. Nearly all compressed documents are 64 KB or less.

To save bandwidth, software computed features and hit vector tuples are encoded in three different sizes using two, four, or six bytes depending on the query term. These streams and tuples are processed by the feature extraction stage to produce the dynamic features. These, combined with the precomputed software features, are forwarded to subsequent pipeline stages.

Due to our slot-based DMA interface and given that the latency of Feature Extraction is proportional to tuple count, we truncate compressed documents to 64 KB. This represents the only unusual deviation of the accelerated ranker from the pure software implementation, but the effect on search relevance is extremely small. Figure 4 shows a CDF of all document sizes in a 210 Kdoc sample collected from real-world traces. As shown, nearly all of the compressed documents are under 64 KB (only 300 require truncation). On average, documents are 6.5 KB, with the 99th percentile at 53 KB.

For each request, the pipeline produces a single score (a 4 Byte float) representing how relevant the document is to the query. The score travels back up the pipeline through the dedicated network to the FPGA that injected the request. A PCIe DMA transfer moves the score, query ID, and performance counters back to the host.

4.2. Macropipeline

The processing pipeline is divided into macropipeline stages, with the goal of each macropipeline stage not exceeding 8 μs, and a target frequency of 200 MHz per stage, This means that each stage has 1,600 FPGA clock cycles or less to complete processing. Figure 5 shows how we allocate functions to FPGAs in the eight-node group: one FPGA for feature extraction, two for free-form expressions, one for a compression stage that increases the efficiency of the scoring engines, and three to hold the machine-learned scoring models. The eighth FPGA is a spare which allows the Service Manager to rotate the ring upon a machine failure and keep the ranking pipeline alive.

4.3. Queue Manager and Model Reload

So far the pipeline descriptions assumed a single set of features, free form expressions and machine-learned scorer. In practice, however, there are many different sets of features, free forms,

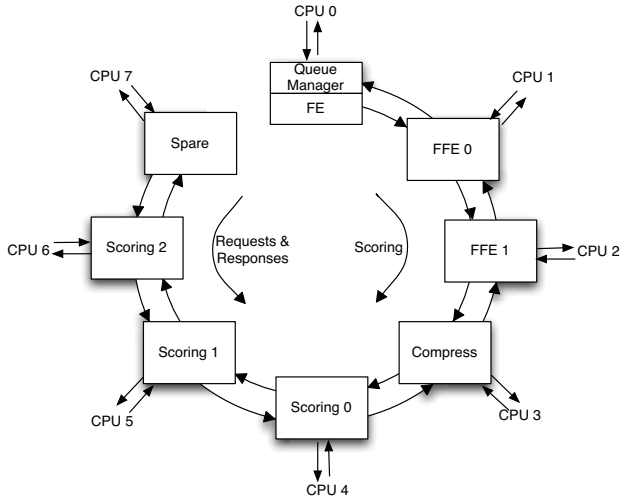

Figure 5: Mapping of ranking roles to FPGAs on the reconfigurable fabric.

and scorers. We call these different sets *models*. Different models are selected based on each query, and can vary for language (e.g. Spanish, English, Chinese), query type, or for trying out experimental models.

When a ranking request comes in, it specifies which model should be used to score the query. The query and document are forwarded to the head of the processing pipeline and placed in a queue in DRAM which contains all queries using that model. The Queue Manager (QM) takes documents from each queue and sends them down the processing pipeline. When the queue is empty or when a timeout is reached, QM will switch to the next queue. When a new queue (i.e. queries that use a different model) is selected, QM sends a Model Reload command down the pipeline, which will cause each stage to load the instructions and data needed to evaluate the query with the specified model.

Model Reload is a relatively expensive operation. In the worst case, it requires all of the embedded M20K RAMs to be reloaded with new contents from DRAM. On each board's D5 FPGA, there are 2,014 M20K RAM blocks, each with 20 Kb capacity. Using the high-capacity DRAM configuration at DDR3-1333 speeds, Model Reload can take up to 250 μs. This is an order of magnitude slower than processing a single document, so the queue manager's role in minimizing model reloads among queries is crucial to achieving high performance. However, while model reload is slow relative to document processing, it is fast relative to FPGA configuration or partial reconfiguration, which ranges from milliseconds to seconds for the D5 FPGA. Actual reload times vary both by stage and by model. In practice model reload takes much less than 250 μs because not all embedded memories in the design need to be reloaded, and not all models utilize all of the processing blocks on the FPGAs.

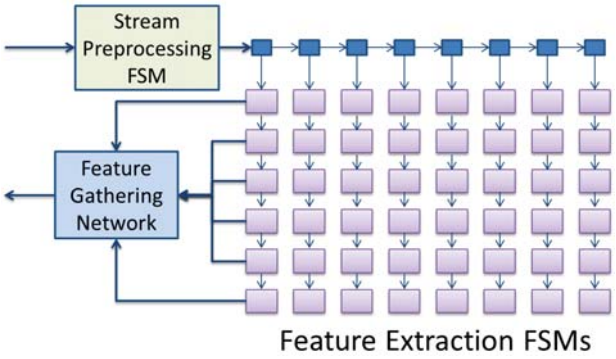

Figure 6: The first stage of the ranking pipeline. A compressed document is streamed into the Stream Processing FSM, split into control and data tokens, and issued in parallel to the 43 unique feature state machines. Generated feature and value pairs are collected by the Feature Gathering Network and forward on to the next pipeline stage.

4.4. Feature Extraction

The first stage of the scoring acceleration pipeline, Feature Extraction (FE), calculates numeric scores for a variety of "features" based on the query and document combination. There are potentially thousands of unique features calculated for each document, as each feature calculation produces a result for every stream in the request—furthermore, some features produce a result per query term as well. Our FPGA accelerator offers a significant advantage over software because each of the feature extraction engines can run in parallel, working on the same input stream. This is effectively a form of Multiple Instruction Single Data (MISD) computation.

We currently implement 43 unique feature extraction state machines, with up to 4,484 features calculated and used by downstream FFE stages in the pipeline. Each state machine reads the stream of tuples one at a time and performs a local calculation. For some features that have similar computations, a single state machine is responsible for calculating values for multiple features. As an example, the *NumberOfOccurences* feature simply counts up how many times each term in the query appears in each stream in the document. At the end of a stream, the state machine outputs all non-zero feature values—for *NumberOfOccurences*, this could be up to the number of terms in the query.

To support a large collection of state machines working in parallel on the same input data at a high clock rate, we organize the blocks into a tree-like hierarchy and replicate the input stream several times. Figure 6 shows the logical organization of the FE hierarchy. Input data (the hit-vector) is fed into a Stream Processing state machine which produces a series of control and data messages that the various feature state machines process. Each state machine processes the stream a rate of 1-2 clock cycles per token. When a state machine finishes its computation, it emits one or more feature index and values that are fed into the Feature Gathering Network

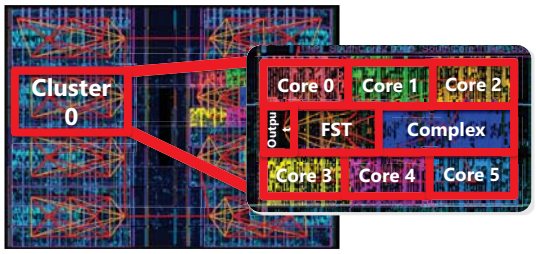

Figure 7: FFE Placed-and-Routed on FPGA.

that coalesces the results from the 43 state machines into a single output stream for the downstream FFE stages. Inputs to FE are double-buffered to increase throughput.

4.5. Free Form Expressions

Free Form Expressions (FFEs) are mathematical combinations of the features extracted during the Feature Extraction stage. FFEs give developers a way to create hybrid features that are not conveniently specified as feature extraction state machines. There are typically thousands of FFEs, ranging from very simple (such as adding two features) to large and complex (thousands of operations including conditional execution and complex floating point operators such as *ln*, *pow*, and *divide*). FFEs vary greatly across different models, so it is impractical to synthesize customized datapaths for each expression.

One potential solution is to tile many off-the-shelf soft processor cores (e.g., Nios II), but these single-threaded cores are not efficient at processing thousands of threads with long-latency floating point operations in the desired amount of time per macropipeline stage (8 μs). Instead, we developed a custom multicore processor with massive multithreading and long-latency operations in mind. The result is the FFE processor shown in Figure 7. As we will describe in more detail, the FFE microarchitecture is highly area-efficient, allowing us to instantiate 60 cores on a single D5 FPGA.

There are three key characteristics of the custom FFE processor that makes it capable of executing all of the expressions within the required deadline. First, each core supports 4 simultaneous threads that arbitrate for functional units on a cycle-by-cycle basis. While one thread is stalled on a long operation such as *fpdivide* or *ln*, other threads continue to make progress. All functional units are fully-pipelined, so any unit can accept a new operation on each cycle.

Second, rather than fair thread scheduling, threads are statically prioritized using a priority encoder. The assembler maps the expressions with the longest expected latency to Thread Slot 0 on all cores, then fills in Slot 1 on all cores, and so forth. Once all cores have one thread in each thread slot, the remaining threads are appended to the end of previously-mapped threads, starting again at Thread Slot 0.

Third, the longest latency expressions are split across multiple FPGAs. An upstream FFE unit can perform part of the computation and produce an intermediate result called a *metafeature*. These metafeatures are sent to the downstream

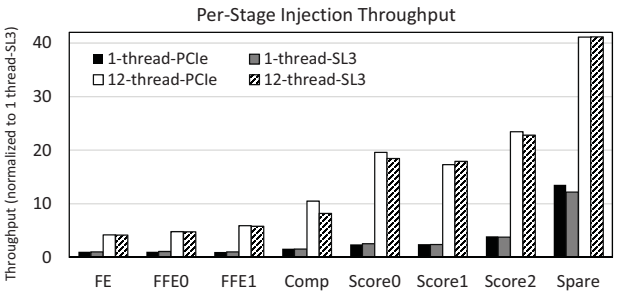

Figure 8: The maximum compressed document injection rate (single- and multi-threaded) for each individual FPGA stage when operating in PCIe-only and SL3 loopback. Results are normalized to single-threaded PCIe throughput.

FFEs like any other feature, effectively replacing that part of the expressions with a simple feature read.

Because complex floating point instructions consume a large amount of FPGA area, multiple cores (typically 6) are clustered together to share a single *complex* block. Arbitration for the block is fair with round-robin priority. The complex block consists of units for *ln*, *fpdiv*, *exp*, and *float-to-int*. *Pow*, *integer divide*, and *mod* are all translated into multiple instructions by the compiler to eliminate the need for expensive, dedicated units. In addition, the complex block contains the feature storage tile (FST). The FST is double-buffered, allowing one document to be loaded while another is processed.

4.6. Document Scoring

The last stage of the pipeline is a machine learned model evaluator which takes the features and free form expressions as inputs and produces single floating-point score. This score is sent back to the Search software, and all of the resulting scores for the query are sorted and returned to the user in sorted order as the sorted search results.

5. Evaluation

We evaluate the Catapult fabric by deploying and measuring the Bing ranking engine described in Section 4 on a bed of 1,632 servers with FPGAs. Our investigation focuses on node-, ring-, and system-level experiments to understand the impact of hardware acceleration on latency and throughput. We also report FPGA area utilization and power efficiency.

Node-Level Experiments We measure each stage of the pipeline on a single FPGA and inject scoring requests collected from real-world traces. Figure 8 reports the average throughput of each pipeline stage (normalized to the slowest stage) in two loopback modes: (1) requests and responses sent over PCIe and (2) requests and responses routed through a loopback SAS cable (to measure the impact of SL3 link latency and throughput on performance). Overall, the results show a significant variation in throughput across all stages. Although the stages devoted to scoring achieve very high processing rates, the pipeline is limited by the throughput of FE.

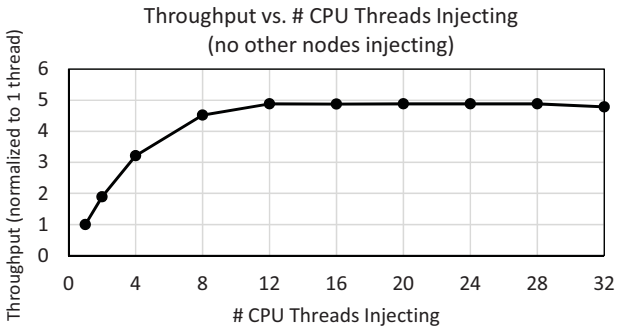

Figure 9: Overall pipeline throughput increases from 1 to 12 threads (normalized to a single thread). Beyond 12 threads, the throughput is limited by the slowest stage in the pipeline.

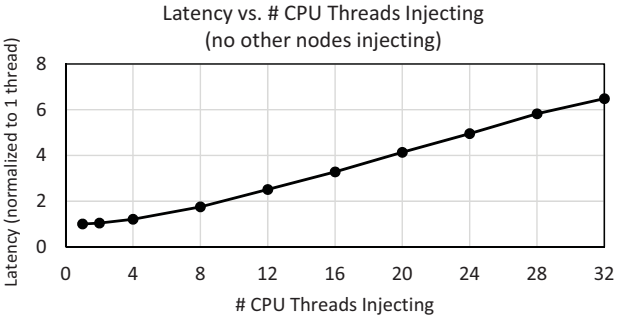

Figure 10: Latency in the pipeline increases with the number of threads due to queuing.

Ring-Level Experiments (single-node injector) In our ring-level experiments, we perform injection tests on a full pipeline with eight FPGAs. Figure 9 shows the normalized pipeline throughput when a single node (in this case FE) injects documents with a varying number of CPU threads. As shown in Figure 9, we achieve full pipeline saturation at around 12 CPU threads, a level consistent with our node-level throughput experiments. For the same set of conditions, Figure 10 plots the normalized latency for the user-level software (i.e., between the time the ranking application injects a document and when the response is received) as thread count increases.

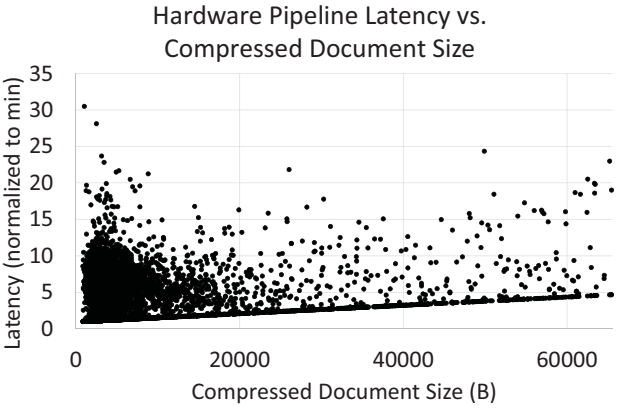

Figure 11: This experiment plots the end-to-end hardware pipeline latency (normalized to the smallest measured value) against the input compressed document size.

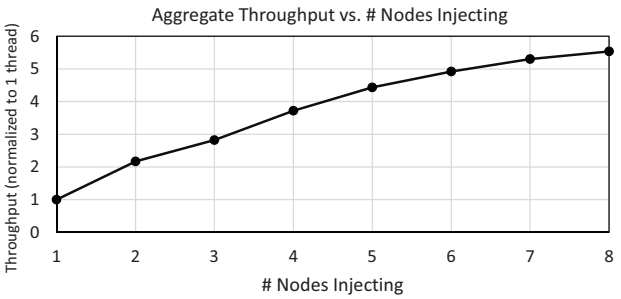

Figure 12: Aggregate throughput increases almost linearly with the number of injecting nodes. In this experiment, only 1 thread injects from each node.

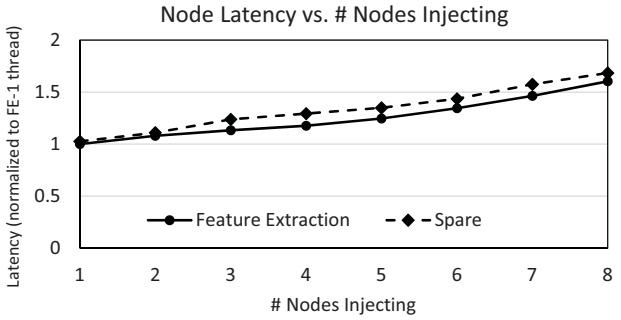

Figure 13: As the number of nodes injecting (1 thread each) increases from 1 to 8, request latency increases slightly due to increased contention for network bandwidth between nodes.

Figure 11 shows the unloaded latency of the scoring pipeline versus the size of a compressed document. The results show a minimum latency incurred that is proportional to the document size (i.e., the buffering and streaming of control and data tokens) along with a variable computation time needed to process the input documents.

Ring-Level Experiments (multi-node injectors) We next evaluate the effect on latency and throughput when multiple servers are allowed to inject documents into a shared ranking pipeline. Figure 12 shows the aggregate pipeline throughput as we increase the total number of injecting nodes. When all eight servers are injecting, the peak pipeline saturation is reached (equal to the rate at which FE can process scoring requests). Under the same conditions, Figure 13 shows the latencies observed by two different nodes injecting requests from the head (FE) and tail (Spare) of the pipeline. Because the Spare FPGA must forward its requests along a channel shared with responses, it perceives a slightly higher but negligible latency increase over FE at maximum throughput.

Production Software Measurements In this section, we compare the average and tail latency distributions of Bing's production-level ranker running with and without FPGAs on a bed of 1,632 servers (of which 672 run the ranking service). For a range of representative injection rates per server used in production, Figure 14 illustrates how the FPGA-accelerated ranker substantially reduces the end-to-end scoring latency relative to software. For example, given a target injection rate

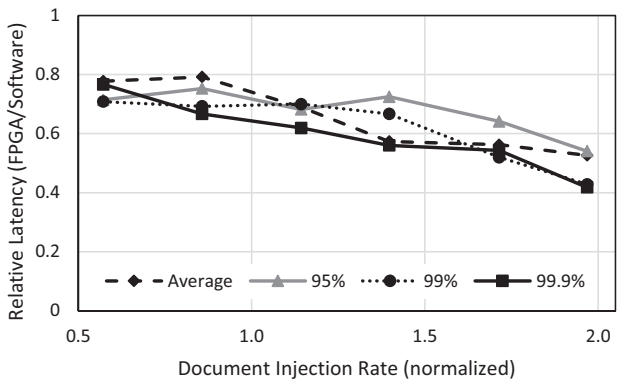

Figure 14: The FPGA ranker achieves lower average and tail latencies relative to software as the injection rate increases.

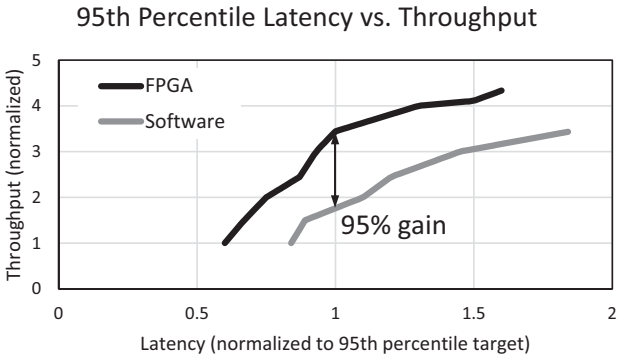

Figure 15: The points on the x-axis at 1.0 show the maximum sustained throughputs on both the FPGA and software while satisfying Bing's target for latency at the 95th percentile.

of 1.0 per server, the FPGA reduces the worst-case latency by 29% in the 95th percentile distribution. The improvement in FPGA scoring latency increases further at higher injection rates, because the variability of software latency increases at higher loads (due to contention in the CPU's memory hierarchy) while the FPGA's performance remains stable.

Figure 15 shows the measured improvement in scoring throughput while bounding the latency at the 95th percentile distribution. For the points labeled on the x-axis at 1.0 (which represent the maximum latency tolerated by Bing at the 95th percentile), the FPGA achieves a 95% gain in scoring throughput relative to software.

Given that FPGAs can be used to improve both latency and throughput, Bing could reap the benefits in two ways: (1) for equivalent ranking capacity, fewer servers can be purchased (in the target above, by nearly a factor of two), or (2) new capabilities and features can be added to the software and/or hardware stack without exceeding the maximum allowed latency.

FPGA Area, Power, and Frequency Table 1 shows the FPGA area consumption and clock frequencies for all of the stages devoted to ranking. Despite the modest area consumption and operating at clock frequencies much lower than conventional processors, the use of FPGAs significantly improves throughput and latency. In the long term, there is substan-

	FE	FFE0	FFE1	Comp	Scr0	Scr1	Scr2	Spare
Logic (%)	74	86	86	20	47	47	48	10
RAM (%)	49	50	50	64	88	88	90	15
DSP (%)	12	29	29	0	0	0	1	0
Clock (MHz)	150	125	125	180	166	166	166	175

Table 1: FPGA area usage and clock frequencies for each of the ranking stages.

tial headroom to improve both the FPGA clock rate and area efficiency of our current pipeline.

To measure the maximum power overhead of introducing FPGAs to our servers, we ran a "power virus" bitstream on one of our FPGAs (i.e., maxing out the area and activity factor) and measured a modest power consumption of 22.7 W.

6. Related Work

Many other groups have worked on incorporating FPGAs into CPU systems to accelerate workloads in large-scale systems.

One challenge to developing a hybrid computing system is the integration of server-class CPUs with FPGAs. One approach is to plug the FPGA directly onto the native system bus, for example, in systems using AMD's HyperTransport [23, 10], or Intel's Front Side Bus [18] and QuickPath Interconnect (QPI) [15]. While integrating the FPGA directly onto the processor bus would reduce DMA latency, this latency is not the bottleneck in our application and would not significantly improve overall performance. In addition, attaching the FPGA to QPI would require replacing one CPU with an FPGA, severely impacting the overall utility of the server for applications which cannot use the FPGA.

IBM's Coherence Attach Processor Interface (CAPI) [26] and Convey's Hybrid-core Memory Interconnect (HCMI) [8] both enable advanced memory sharing with coherence between the FPGA and CPU. Since our ranking application only requires simple memory sharing, these mechanisms are not yet necessary but may be valuable for future applications.

Instead of incorporating FPGAs into the server, several groups have created network-attached FPGA appliances that operate over Ethernet or Infiniband. The Convey HC-2 [8], Maxeler MPC series [21], BeeCube BEE4 [5] and SRC MAPstation [25] are all examples of commercial FPGA acceleration appliances. While the appliance model appears to be an easy way to integrate FPGAs into the datacenter, it breaks homogeneity and reduces overall datacenter flexibility. In addition, many-to-one network communication can result in dropped packets, making the bounds on latencies much harder to guarantee. Finally, the appliance creates a single point of failure that can disable many servers, thus reducing overall reliability. For these reasons, we distribute FPGAs across servers.

Several large systems have also been built with distributed FPGAs, including the Cray XD-1 [9], Novo-G [12], and QP [22]. These systems integrate the FPGA with the CPU, but the FPGA-to-FPGA communication must be routed through the CPU. Maxwell [4] is the most similar to our design, as it directly connects FPGAs in a 2-D torus using InfiniBand cables, although the FPGAs do not implement routing logic. These systems are targeted to HPC rather than datacenter workloads, but they show the viability of FPGA acceleration in large systems. However, datacenters require greater flexibility within tighter cost, power, and failure tolerance constraints than specialized HPC machines, so many of the design decisions made for these systems do not apply directly to the Catapult fabric.

FPGAs have been used to implement and accelerate important datacenter applications such as Memcached [17, 6] compression/decompression [14, 19], K-means clustering [11, 13], and web search. Pinaka [29] and Vanderbauwhede, et. al [27] used FPGAs to accelerate search, but focused primarily on the Selection stage of web search, which selects which documents should be ranked. Our application focuses on the Ranking stage, which takes candidate documents chosen in the Selection stage as the input.

The FFE stage is a soft processor core, one of many available for FPGAs, including MicroBlaze [28] and Nios II [2]. Unlike other soft cores, FFE is designed to run a large number of threads, interleaved on a cycle-by-cycle basis.

The Shell/Role design is aimed at abstracting away the board-level details from the application developer. Several other projects have explored similar directions, including VirtualRC [16], CoRAM [7], BORPH [24], and LEAP [1].

7. Conclusions

FPGAs show promise for accelerating many computational tasks, but they have not yet become mainstream in commodity systems. Unlike GPUs, their traditional applications (rapid ASIC prototyping and line-rate switching) are unneeded in high-volume client devices and servers. However, FPGAs are now powerful computing devices in their own right, suitable for use as fine-grained accelerators. This paper described a large-scale reconfigurable fabric intended for accelerating datacenter services. Our goal in building the Catapult fabric was to understand what problems must be solved to operate FPGAs at scale, and whether significant performance improvements are achievable for large-scale production workloads.

When we first began this investigation, we considered both FPGAs and GPUs as possible alternatives. Both classes of devices can support copious parallelism, as both have hundreds to thousands of arithmetic units available on each chip. We decided not to incorporate GPUs because the current power requirements of high-end GPUs are too high for conventional datacenter servers, but also because it was unclear that some latency-sensitive ranking stages (such as feature extraction) would map well to GPUs.

Our study has shown that FPGAs can indeed be used to accelerate large-scale services robustly in the datacenter. We have demonstrated that a significant portion of a complex datacenter service can be efficiently mapped to FPGAs, by

using a low-latency interconnect to support computations that must span multiple FPGAs. Special care must be taken when reconfiguring FPGAs, or rebooting machines, so that they do not crash the host server or corrupt their neighbors. We described and tested a high-level protocol for ensuring safety when reconfiguring one or more chips. With this protocol and the appropriate fault handling mechanisms, we showed that a medium-scale deployment of FPGAs can increase ranking throughput in a production search infrastructure by 95% at comparable latency to a software-only solution. The added FPGA compute boards only increased power consumption by 10% and did not exceed our 30% limit in the total cost of ownership of an individual server, yielding a significant overall improvement in system efficiency.

We conclude that distributed reconfigurable fabrics are a viable path forward as increases in server performance level off, and will be crucial at the end of Moore's Law for continued cost and capability improvements. Reconfigurability is a critical means by which hardware acceleration can keep pace with the rapid rate of change in datacenter services.

A major challenge in the long term is programmability. FPGA development still requires extensive hand-coding in RTL and manual tuning. Yet we believe that incorporating domain-specific languages such as Scala or OpenCL, FPGA-targeted C-to-gates tools such as AutoESL or Impulse C, and libraries of reusable components and design patterns, will be sufficient to permit high-value services to be productively targeted to FPGAs for now. Longer term, more integrated development tools will be necessary to increase the programmability of these fabrics beyond teams of specialists working with large-scale service developers. Within ten to fifteen years, well past the end of Moore's Law, compilation to a combination of hardware and software will be commonplace. Reconfigurable systems, such as the Catapult fabric presented here, will be necessary to support these hybrid computation models.

Acknowledgments

Many people across many organizations contributed to the construction of this system, and while they are too numerous to list here individually, we thank our collaborators in Microsoft Global Foundation Services, Bing, the Autopilot team, and our colleagues at Altera and Quanta for their excellent partnership and hard work. We thank Reetuparna Das, Ofer Dekel, Alvy Lebeck, Neil Pittman, Karin Strauss, and David Wood for their valuable feedback and contributions. We are also grateful to Qi Lu, Harry Shum, Craig Mundie, Eric Rudder, Dan Reed, Surajit Chaudhuri, Peter Lee, Gaurav Sareen, Darryn Dieken, Darren Shakib, Chad Walters, Kushagra Vaid, and Mark Shaw for their support.

References

[1] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11, 2011.

[2] *Nios II Processor Reference Handbook*, 13th ed., Altera, 2014.

[3] *Stratix V Device Handbook*, 14th ed., Altera, 2014.

[4] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. Mccormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell - a 64 FPGA Supercomputer," *Engineering Letters*, vol. 16, pp. 426–433, 2008.

[5] *BEE4 Hardware Platform*, 1st ed., BEECube, 2011.

[6] M. Blott and K. Vissers, "Dataflow Architectures for 10Gbps Line-Rate Key-Value Stores," in *HotChips 2013*, August 2013.

[7] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An In-fabric Memory Architecture for FPGA-based Computing," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11, 2011.

[8] *The Convey HC-2 Computer*, Conv-12-030.2 ed., Convey, 2012.

[9] *Cray XD1 Datasheet*, 1st ed., Cray, 2005.

[10] *DRC Accelium Coprocessors Datasheet*, Ds ac 7-08 ed., DRC, 2014.

[11] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic Transformations in the Implementation of K-Means Clustering on Reconfigurable Hardware," in *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '01, 2001.

[12] A. George, H. Lam, and G. Stitt, "Novo-G: At the Forefront of Scalable Reconfigurable Supercomputing," *Computing in Science Engineering*, vol. 13, no. 1, pp. 82–86, 2011.

[13] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly Parameterized K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," in *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs*, ser. RECONFIG '11, 2011.

[14] *IBM PureData System for Analytics N2001*, WAD12353-USEN-01 ed., IBM, 2013.

[15] Intel, "An Introduction to the Intel Quickpath Interconnect," 2009.

[16] R. Kirchgessner, G. Stitt, A. George, and H. Lam, "VirtualRC: A Virtual FPGA Platform for Applications and Tools Portability," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12, 2012.

[17] M. Lavasani, H. Angepat, and D. Chiou, "An FPGA-based In-line Accelerator for Memcached," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2013.

[18] L. Ling, N. Oliver, C. Bhushan, W. Qigang, A. Chen, S. Wenbo, Y. Zhihong, A. Sheiman, I. McCallum, J. Grecco, H. Mitchel, L. Dong, and P. Gupta, "High-performance, Energy-efficient Platforms Using In-socket FPGA Accelerators," in *International Symposium on Field Programmable Gate Arrays*, ser. FPGA '09, 2009.

[19] A. Martin, D. Jamsek, and K. Agarawal, "FPGA-Based Application Acceleration: Case Study with GZIP Compression/Decompression Streaming Engine," in *ICCAD Special Session 7C*, November 2013.

[20] *How Microsoft Designs its Cloud-Scale Servers*, Microsoft, 2014.

[21] O. Pell and O. Mencer, "Surviving the End of Frequency Scaling with Reconfigurable Dataflow Computing," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, Dec. 2011.

[22] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, and W. Hwu, "QP: A Heterogeneous Multi-Accelerator Cluster," 2009.

[23] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning, "An Open-source HyperTransport Core," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 3, Sep. 2008.

[24] H. K.-H. So and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, Jan. 2008.

[25] *MAPstation Systems*, 70000 AH ed., SRC, 2014.

[26] J. Stuecheli, "Next Generation POWER microprocessor," in *HotChips 2013*, August 2013.

[27] W. Vanderbauwhede, L. Azzopardi, and M. Moadeli, "FPGA-accelerated Information Retrieval: High-efficiency document filtering," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Aug 2009, pp. 417–422.

[28] *MicroBlaze Processor Reference Guide*, 14th ed., Xilinx, 2012.

[29] J. Yan, Z.-X. Zhao, N.-Y. Xu, X. Jin, L.-T. Zhang, and F.-H. Hsu, "Efficient Query Processing for Web Search Engine with FPGAs," in *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '12, 2012.