A Reconfigurable Computing System Based on a Cache-Coherent Fabric

Neal Oliver, Rahul R Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco,
Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, Henry Mitchel, Suchit Subhaschandra,
Arthur Sheiman, Tim Whisonant, and Prabhat Gupta
Intel Corporation
{neal.oliver, bhushan.chitlur, p.k.gupta}@intel.com

***Abstract*—Typical reconfigurable computing systems are based on an I/O interconnect such as PCIe. This yields good bandwidth performance, but incurs significant overhead for small packet sizes, and makes the implementation of non-streaming-data applications unduly difficult. We describe an architecture based on Intel® QuickPath Interconnect® that addresses these concerns.**

***Index Terms*—Reconfigurable computing, cache coherent, in-socket FPGA, shared virtual memory**

I. INTRODUCTION

A reconfigurable computing (RC) device, such as a Field Programmable Gate Array (FPGA), is a computing device whose computing elements and data paths are programmed dynamically (i.e. after manufacture) to correspond to the datapaths of an algorithm's dataflow graph. RC devices can achieve higher computational and energy efficiency, as measured by higher computational throughput, lower computational latency, or higher computational performance-per-watt.

Typical RC systems couple a CPU with FPGAs via a PCIe[1] interconnect to create a loosely coupled heterogeneous architecture. PCIe is designed primarily for high bandwidth; it incurs high overhead and latency for transferring small packets. To obtain a performance benefit from a PCIe-based FPGA system, an application must be balanced to minimize the PCIe data transfer overhead. Because of this, most applications running on such platforms are streaming data-oriented.

CPU architectures use cache-coherent, low-latency, high speed fabrics to communicate between CPUs. A cache-coherent fabric, such as Intel® QuickPath Interconnect Technology® (Intel QPI®), allows all computing devices in the system, including RC devices, to share system memory, intermediated by the fabric's cache coherence protocol. A range of new use-cases is thus enabled, including shared virtual memory (SVM) and low-latency message transfers. Intel QPI is designed for efficient cache line transfers, and so additionally works well for RC applications that require small payload data transfers.

Programming an RC device requires specialized skills in Hardware Description Language (HDL) and hardware debugging, which are not often possessed by systems or application programmers. The Intel® QuickAssist® QPI-based FPGA Accelerator Platform (QAP) architecture allows RC device programming and application programming tasks to be decoupled from one another in a programmer-friendly manner.

In the QAP platform, one of the CPU sockets is populated with a custom FPGA module capable of implementing a full speed Intel QPI link. Application-specific hardware elements, called Accelerator Functional Units (AFUs), are implemented on the module, and used by an application. Section II provides a brief summary of the Intel QPI architecture and introduces terms that are used later in the paper. Section III describes the FPGA module and the hardware architecture that enables the attachment of AFU(s) to a QPI-based system. Section IV gives an overview of the Accelerator Abstraction Layer (AAL). Section VI discusses the use-cases that drove the system architecture. We end with a discussion of future work in Section VII and a conclusion in Section VIII.

II. BACKGROUND

A. Intel QuickPath Interconnect

Intel QuickPath Interconnect (Intel QPI) [1] is a high-speed, cache coherent, packetized, point-to-point interconnect used in the latest generation of Intel microprocessors. This high speed link may be used to create a Non-Uniform Memory Access (NUMA) style system fabric. It has been acknowledged to be a very high-performing solution for interconnecting Intel microprocessors [2] [3]. Several Reliability-Availability-Serviceability (RAS) features, e.g. implicit Cyclic Redundancy Check (CRC) with link-level retry and error recovery, fail over modes, are supported

Rahul R Sharma is a Doctoral Candidate at UNC Charlotte at the time of publishing. He was a Graduate Intern when submitting this paper.

Elkin Garcia is a Graduate Intern and a Doctoral Candidate at University of Delaware at the time of publishing

[1]Intel® and Xeon® are registered trademarks of Intel Corporation. Other trademarks are the property of their respective owners.

TABLE I. Glossary of Acronyms

Term	Explanation
AAL	Accelerator Abstraction Layer
AAS	Accelerator Abstraction Subsystem
AFU	Accelerator Functional Unit
AIA	Application Interface Adapter
CA	Caching Agent: A QPI specification defined agent
CCI	Core-Cache Interface
CRC	Cyclic Redundancy Check
CSR	Control and Status Register
DDR	Double Data Rate
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
FSB	Front-side Bus
GT/Sec	Giga-Transfers per Second
HA	Home Agent: A QPI specification defined agent
HDL	Hardware Description Language
HLL	High-Level Language
MESIF	Modified, Exclusive, Shared, Invalid, Forward (cache coherency protocol)
NUMA	Non-Uniform Memory Access
PCIe	Peripheral Component Interconnect Express
QAP	QPI-based FPGA Accelerator Platform
QLP	QPI Link & Protocol Layer (implemented on FPGA)
QPH	QPI Physical Layer (implemented on FPGA)
QPI	QuickPath Interconnect
RAS	Reliability-Availability-Serviceability
RC	Reconfigurable Computing
Rx	Receive
SPL	System Protocol Layer
SVM	Shared Virtual Memory
Tx	Transmit
WS	Workspace

at high speeds ranging from 4.8 to 6.4 Giga-Transfers per Second (GT/sec).

In a distributed shared memory system, coherent memory may be distributed across different CPUs/devices on the platform. Intel QPI implements a cache coherency protocol that keeps the distributed memory and the caching structures coherent during system operation. Caching activities are carried out by two distinct agents, the Caching Agent (CA) and the Home Agent (HA). A CA initiates transactions to coherent memory and provides copies of coherent memory contents to other CAs. At a high level, an Intel QPI CA must be able to generate Read and Write messages to coherent system memory. The CA is also responsible for responding to snoops generated by other Intel QPI agents in the system. Each piece of coherent memory requires an HA whose role is to service coherent requests targeted to it from other CAs in the system. The HA is also responsible for managing conflicts that may arise among the different CAs.

III. QAP SYSTEM ARCHITECTURE

A. Platform Topology

Figure 1 depicts the Intel QAP Platform topology with 4 CPU sockets. The FPGA module can be placed in any of the CPU sockets to allow users to build a customized platform with varying combinations of CPUs and FPGA modules. The

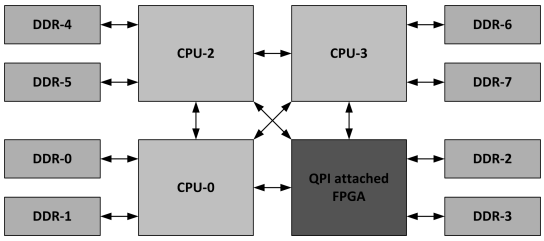

Fig. 1: Intel QAP Platform Topology

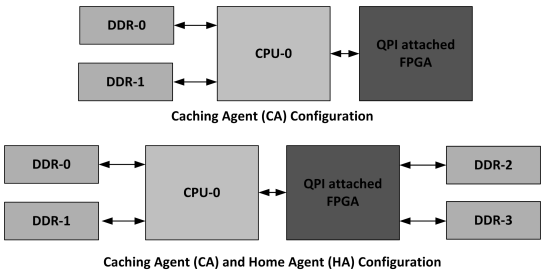

Fig. 2: QAP in CA/HA Configurations

FPGA module itself can have more than one FPGA device.

The QAP protocol stack can be configured in two different modes, depending on the requirements of the system being implemented on the FPGA. When configured as a CA, the FPGA can initiate cacheable transactions and participate in the coherence protocol with the memory connected to the CPU. When configured as an HA, FPGA-connected local memory is made visible to all CPUs across the platforms. Figure 2 depicts these two configurations.

Figure 3 depicts the QAP protocol stack implemented on the FPGA module. The Intel QPI Physical Layer uses the existing high speed serial I/Os on the FPGA device, configured to implement the Intel QPI Physical Layer signaling. The Intel QPI Link Layer provides reliable data transfer between two end-points using flow control and error recovery capabilities. The Intel QPI Link & Protocol Layer participates in the cache coherence protocol with other Intel QPI agents in the system. The Core-Cache Interface (CCI) provides direct access to the coherent system memory. The System Protocol Layer (SPL) implements the initialization, communication and data movement protocols that exist between the application and the AFU.

QPI Physical Layer

The QPI Physical Layer (QPH) consists of the actual wires carrying the signals, as well as the circuitry and logic required to support data transfer across the link. A link pair consists of two unidirectional links that operate simultaneously. The specification defines operation in full, half, and quarter widths, where full width is 20 bits. Each link provides one forwarded clock in each direction. The link is operated at double data rate, running at speeds up to

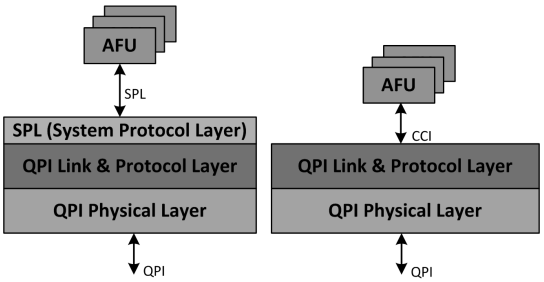

Fig. 3: QAP protocol Stack on FPGA - Implementation

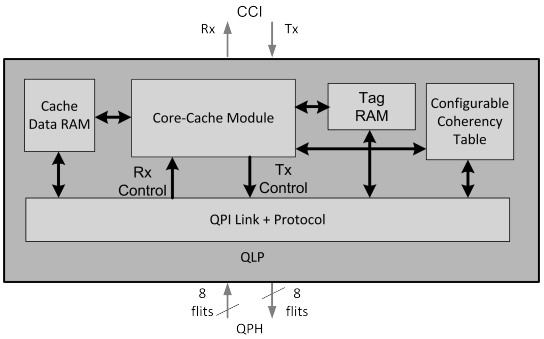

Fig. 4: QPI Link & Protocol Layer (QLP) Architecture

4.8 and 6.4 GT/Sec, depending on the CPU implementations [1]. QPH assembles the data it receives and transfers it to the FPGA Intel QPI Link layer in units of 8 flits (the basic layer unit of data transfer), where each flit is 80 bits long.

QPI Link & Protocol Layers

Formally, the Intel QPI Link and Protocol layers are distinct elements, but our implementation of these elements is relatively tightly integrated for performance; we henceforth refer to QPI Link & Protocol layer assembly as the QLP layer. Although FPGA IOs are designed to work in the GHz range, the programmable fabric and block RAM runs only in hundreds of MHz. To keep up with the offered Intel QPI bandwidth, the QLP is architected as a massively parallel engine that can sink and source 8 flits per clock. Intricate optimizations have been made in the link layer to the retry engine, virtual channel support logic and credit exchange, features that must be regulated at flit granularity. The implementation details are beyond the scope of this paper.

QLP can be configured as a CA, HA, and Configuration Agent. It participates in the MESIF [1] coherency protocol with other Intel QPI agents in the system to implement a 256KB 4-way set-associative cache in the FPGA. The behavior of the coherence protocol is configurable via a cache coherency table, implemented as a case block in RTL. Features such as cache line pre-fetcher and auto-flush, which periodically evicts the dirty (Modified state) lines from the cache, are also provided. As a Configuration Agent, the QLP implements Intel x86 configuration space, and is capable of

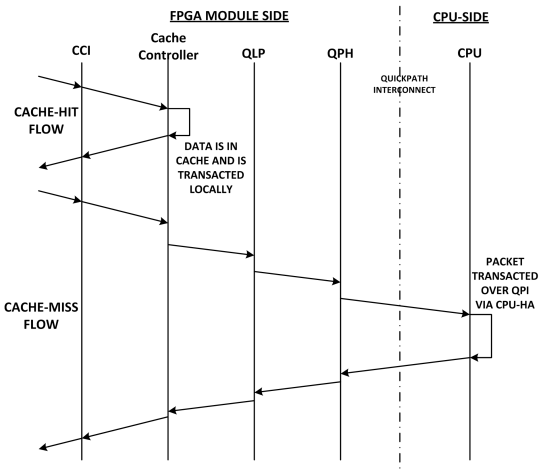

Fig. 5: Cache Hit/Miss Protocol Flow (FPGA implementing CA only)

processing Control & Status Register (CSR) read and write cycles targeted to the FPGA.

QLP implements the Intel QPI HA message types required to expose the FPGA connected DRAM as coherent memory to the rest of the system. HA resolves conflicts in case of simultaneous access from multiple CAs and sends snoops to other CAs. To reduce the number of snoops generated by the HA, QLP implements a snoop filter [1]. This configuration enables a range of applications that can benefit from the low latency and high bandwidth of locally attached memory. Since this memory is part of the coherent system memory, it is also directly accessible from the CPU. Such a configuration was previously impossible with PCIe-attached devices.

The CCI is designed to match the peak Intel QPI bandwidth. It accepts a Memory Read Request, a Write Request with 64B data, and an Interrupt Request in the Tx (FPGA-to-CPU) direction. In the Rx direction, CCI returns a Read Response with 64B data, a Write Response, CSR Write cycle with 4B data payload and an Unordered message. The Unordered message is specially crafted to provide very low latency signaling from CPU to the FPGA device.

A CCI request takes a 32-bit cache line address. The address is first checked against the cache. If it hits the cache, then it is completed locally, generating no Intel QPI transaction (see illustration in Figure 5). If the request misses the cache, then the address is looked up in a decode table to determine whether the request must be routed to the FPGA HA or must be directed to the CPU HA. A cache hit is orders of magnitude faster than a cache miss request. Support for posted write requests yields lower write latency for cache misses. QLP also implements a store fence, which is a memory barrier across write requests.

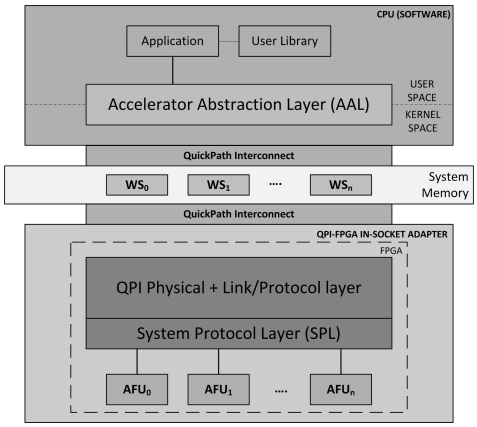

Fig. 6: QAP Hardware-Software Architecture

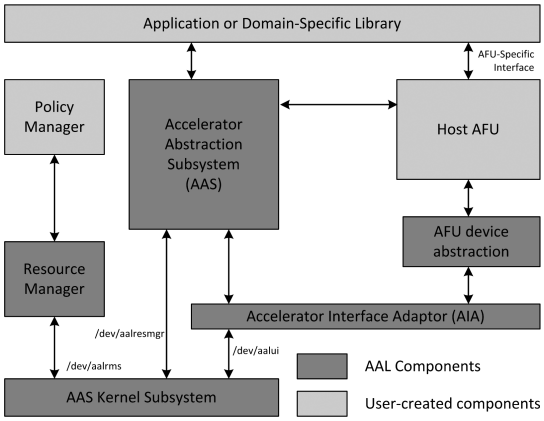

Fig. 7: Accelerator Abstraction Layer (AAL) Components and Interfaces

B. QAP Virtual Memory

The use of physically-addressed memory in accelerators has two drawbacks. Physically-addressed memory accesses are unprotected from errors and malice. It may be difficult or impossible to obtain large contiguous blocks of physical memory, which some applications require. QAP implements a virtual memory subsystem to avoid the necessity of physically-addressed memory; the subsystem is implemented partially in hardware by the SPL and partially in software by the Accelerator Abstraction Later (AAL).

AAL creates an FPGA page table structure in the system memory and passes the page table base pointer to the SPL. This page table is associated with the application, and does not change during the lifetime of the application. At initialization, SPL loads the page table entries pointed to by the page table base pointer. The SPL interface accepts memory read or write requests with either physical or virtual addresses, and provides in-order responses.

The above describes the core functionality of SPL. The next generation of SPL will support dynamic page table updates and demand paging (such as page faults). This is a layered architecture that can be extended to provide slave mode DMA functions, user mode descriptor queues, scatter-gather operations, and other features.

IV. QAP SOFTWARE ARCHITECTURE

The Accelerator Abstraction Layer (AAL) is the software subsystem for QAP. It is responsible for two primary tasks: setting up and managing FPGA virtual memory, and presenting a programming interface for applications. Figure 6 and Figure 7 depict the overall QAP hardware-software architecture and the components composing AAL.

In Figure 7, the application uses the AAL subsystem to perform its function. AAL creates and pins workspaces (WS_i in the figure) from QAP virtual memory. The application may read and write the workspaces as normal memory, but uses the AAL APIs to interact with the AFUs.

AAL is implemented partly in kernel mode, in order to perform operations with a minimum of context switches. The AAL-provided components are the Acceleration Abstraction Subsystem (AAS), Resource Manager, AAS Kernel Subsystem, Application Interface Adaptor (AIA) and AIA Proxies. They provide services and interfaces to the user-developed components that comprise the application. The performance-critical components are implemented in kernel mode.

A. Virtual Memory Management

The AAS is a persistent process that, upon startup, allocates and pins physically-addressed memory obtained from the operating system, and builds FPGA page tables, which are subsequently used by SPL (see Subsection III-A). AAS and SPL interact to allow read and write accesses in either physical or virtual address spaces.

The Resource Manager discovers QAP modules installed in the host and creates an internal database of the AFUs deployed in the modules.

B. Programming Interface

An AAL application consists of a top-level program and one or more AFUs. The top-level program allocates and configures AFUs. The operations implemented by the AFU are presented to the top-level program via Application Interface Adaptor (AIA) components.

An AFU is architected as two components: a Host AFU, resident on the host CPU, and a Hardware AFU, implemented in RTL on the QAP module and wrapped by the Host AFU.

The split of the AFU architecture between host and hardware components allows the Host AFU to present itself

as an object in the object-oriented AAL architecture, while allowing the Hardware AFU to be accommodated in the QAP protocol stack. It also helps to accommodate the differing skill-sets of hardware designers and systems programmers and algorithm developers. As an AFU is designed, the Host AFU may initially provide both the interfaces and the implementation of the operations. The Host AFU may therefore be used by the top-level program to implement a working application. As the Hardware AFU is designed, the Host AFU is modified to use the operations implemented in the Hardware AFU.

The AFU implementation in software consists of the Host AFU itself, the AFU Package, and a user-provided Policy Manager. The AFU Package encapsulates the hardware configuration knowledge required to instantiate a Host AFU and configure it properly in the AAL system. The Policy Manager encapsulates the rules governing access rights of AAL applications to its underlying hardware module.

V. RELATED WORK

Important tradeoffs between latency and throughput in the evolution of the common PCI interconnect architecture have been analyzed in [4]. They point out several drawbacks of the current trends in the PCI Express interconnect. QPI arises as a feasible alternative to PCIe.

Early studies of the cache coherent interfaces for accelerators has been introduced in [5] for improving performance by facilitating burst transfers of whole cache blocks and reducing control overheads. In addition, it has been shown that the cache coherence communication cost in Multi-Processor System-on-Chip (MPSoC) platforms is reasonable given their advantages over non-coherent systems [6].

Convey Computer has developed an accelerator system, described in [7], in which the FPGA module is attached to the Intel Front-Side Bus (FSB). User-defined instruction sets known as *personalities* implemented on the attached FPGAs, allowing the acceleration of scientific applications. The Smith-Waterman algorithm for aligning DNA sequences has been accelerated using this system [8]. Several efforts are directed at integrating CPUs and reconfigurable resources in the same device, such as the Xilinx Extensible Processing Platform [9] that includes an ARM Cortex A9 MPCore[1] processor with a small amount of programmable logic and hardened peripheral IP.

RAMP [10] describes an FPGA-based emulator of parallel architectures in order to allow hardware-software co-design. FARM [11] also develops a shared virtual memory-based FPGA accelerator, but the use-case emphasized there is of a prototyping platform for exploring various micro-architectural features. We believe that QAP introduces a wider range of use-cases, hardware prototyping being one of them. Additionally, our work presents a software architecture, Accelerator Abstraction Layer (AAL), which provides virtualization and abstraction features.

The OpenCL[1] programming environment [12], and hence various implementations of OpenCL, provide a runtime platform with some features described by the AAL middleware subsystem. However, AAL does not require that the underlying semantics of the hardware device be those of OpenCL, and provides a flexible development paradigm for implementing algorithms on the device.

VI. USAGES

QAP enables a range of new use-cases that take advantage of the low overhead of small message data transfer and cache-coherent system memory. Three notable use-cases are:

1) DirectIO: By implementing an Ethernet port directly onto the QAP module, it may serve both as a low-latency network device and a low-latency component of an end-to-end application. This use-case is of interest for high-frequency trading applications, in which short messages (100-300 bytes) must be received, partially parsed, and then filtered, forwarded, or otherwise processed with extremely low latency.

2) ASIC Emulation: HW-SW co-design is an essential means to jump-start software design in parallel with RTL development. The QAP platform provides a low cost, scalable module to emulate Intel QPI attached node-controller ASIC, telecommunications or military accelerators.

3) Zero Buffer Copy: In a traditional IO device SW architecture, the device driver provides the address of a pinned kernel buffer to the device. After the device finishes writing to the kernel buffer, the device driver copies data from the kernel buffer to the user buffer and provides a pointer to the user buffer to the application. With the QAP Virtual Memory Management engine, the user-to-kernel buffer copy is eliminated. The device driver maps the user buffer to the device virtual page, so that the device can write directly into the application's virtual address space. This saves the extra cost of a user-to-kernel buffer copy. Applications of this mechanism include various *deep packet inspection* applications and high-frequency trading (HFT) applications.

VII. FUTURE WORK

We are considering a number of research and development directions for the system architecture described in this paper. Efforts are currently underway to evaluate QAP thoroughly for workloads in computational finance (both for pricing models and for high-frequency trading) and genomics. Future work will also target applications in seismic imaging.

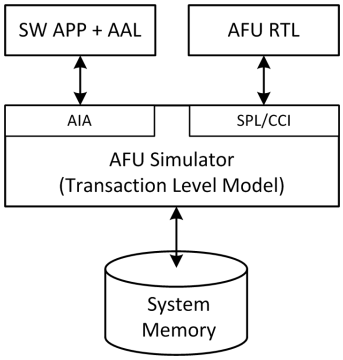

Fig. 8: AFU Simulator tool (under development)

We are actively following advances in the areas of rapid place-and-route and synthesis. The QAP protocol stack can be incorporated into a design as a black box IP block, enabling a reduced "time-to-market" productivity benefit to the designer. This could be vital in areas such as computational finance, where algorithms have a short life span [13].

We are currently developing an AFU Simulator, to accelerate debugging and integration of AFU designs in QAP (see Figure 8). The simulator exposes the AIA (see Figure 7) on the software side and a transaction-level model of CCI or SPL on the hardware side. The host application can switch between a QAP system and the simulator transparently since the AIA interface is consistent. Similarly the AFU RTL which transacts with CCI or SPL can be seamlessly migrated from simulation environment to the FPGA seated on a QAP system. The AFU Simulator uses the Synopsys VCS[1] or Mentor Graphics ModelSim[1] to implement the transaction level model. In the AFU Simulator, the QAP stack is not used, but rather emulated (in a non-cycle accurate manner) in order to provide uniform interfaces to both the AFU and the AIA.

Efforts are also underway to integrate the Impulse CoDeveloper [14] and AutoESL [15] programming environments with QAP. These are programming environments that support close dialects of the C programming language, allowing AFUs to be implemented more rapidly, and by non-hardware designers. Continued advances in High Level Language (HLL) technology (i.e. compilation optimized for area, timing and power, performance improvements, etc.) will drive wider adoption of QAP.

VIII. CONCLUSION

We have reported on an FPGA accelerator architecture, QAP, based on the Intel QPI system fabric. The QAP participates in the Intel QPI protocol and allows applications to use host system memory and cache to achieve low latency and high bandwidth with small message sizes. It also supports a block-oriented protocol to allow more traditional streaming applications to run with high bandwidth. We have also defined a software architecture to support applications split between QAP and host-resident components. Several exciting areas of research remain to be explored.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions of partner companies Altera, Impulse Accelerated Technologies, Nallatech, Pactron, and Xilinx, in the development of these technologies.

REFERENCES

[1] Intel Corporation, "An Introduction to the Intel QuickPath Interconnect," January 2009.

[2] D. Ziakas, A. Baum, R. Maddox, and R. Safranek, "Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures," in *2010 18th IEEE Symposium on High Performance Interconnects*, pp. 1–6, IEEE, 2010.

[3] B. Mutnury, F. Paglia, J. Mobley, G. Singh, and R. Bellomio, "QuickPath Interconnect (QPI) design and analysis in high speed servers," in *Electrical Performance of Electronic Packaging and Systems (EPEPS), 2010 IEEE 19th Conference on*, pp. 265–268, IEEE.

[4] D. J. Miller, P. M. Watts, and A. W. Moore, "Motivating future interconnects: a differential measurement analysis of pci latency," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, (New York, NY, USA), pp. 94–103, ACM, 2009.

[5] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood, "Coherent network interfaces for fine-grain communication," *SIGARCH Comput. Archit. News*, vol. 24, pp. 247–258, May 1996.

[6] G. Girão, B. C. de Oliveira, R. Soares, and I. S. Silva, "Cache coherency communication cost in a NoC-based MPSoC platform," in *Proceedings of the 20th annual conference on Integrated circuits and systems design*, SBCCI '07, (New York, NY, USA), pp. 288–293, ACM, 2007.

[7] T. Brewer, "Instruction Set Innovations for the Convey HC-1 Computer," *Micro, IEEE*, vol. 30, pp. 70 –79, march-april 2010.

[8] J. Allred, J. Coyne, W. Lynch, V. Natoli, J. Grecco, and J. Morrissette, "Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer," 2009.

[9] M. Santarini, "Zynq-7000 EPP Sets Stage for New Era of Innovations," *Xcell Journal*, vol. 75, pp. 8–13.

[10] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research Accelerator for Multiple Processors," *Micro, IEEE*, vol. 27, pp. 46 –57, march-april 2007.

[11] T. Oguntebi, S. Hong, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "FARM: A Prototyping Environment for Tightly-Coupled, Heterogeneous Architectures," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 221–228, 2010.

[12] The Khronos Group, "OpenCL 1.1 Specification." http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, June 2011.

[13] Bank for International Settlements, "Andrew G Haldane: The race to zero.." http://www.bis.org/review/r110720a.pdf, July 2011.

[14] Impulse Accelerated Technologies, "ImpulseC Datasheet." http://www.impulseaccelerated.com, 2007.

[15] Xilinx Corporation, "AutoESL High-Level Synthesis Tool." http://www.xilinx.com/tools/autoesl.htm, 2011.