Developing a Flexible Interface for RapidIO, Hypertransport, and PCI-Express

Christian Sauer[1], Matthias Gries[2], Jose Ignacio Gomez[3], Scott Weber[2], Kurt Keutzer[2]
[1]*Infineon Technologies, Corporate Research, Munich, Germany*
[2]*University of California, Electronics Research Lab, Berkeley*
[3]*Universidad Complutense de Madrid, Spain*
{*sauer, gries, jgomez, sjweber, keutzer*}*@eecs.berkeley.edu*

Abstract

In communication centric application domains flexible interfaces are required to support the variety of recently emerged and still evolving high-speed serial interconnect standards. To develop such interfaces, typical application scenarios have to be identified, and common functionality and specific characteristics of the selected standards need to be analyzed. The results of this analysis can then guide the exploration of the design space. In this paper, we present a methodology which is tailored to the characteristics of serial interconnects. We use Click models for analysis and abstraction of the communication protocols and explore flexible interfaces within a cycle-accurate architecture development framework. Our results show the feasibility of implementing a flexible interface on a packet engine similar to those used in network processors. Based on our findings, we believe that flexible interfaces will form a new family of building blocks for future Systems-on-Chip.

1 Introduction

Embedded systems require a number of dedicated interfaces to communicate with their environment. Some of these interfaces, e.g. memory and network interfaces, are well-covered by I/O standards. Others, for instance the switch fabric interface in network processors, still lack sufficient standardization. The recent enhancements to packet-oriented, link-to-link communication protocols, such as RapidIO, Hypertransport, and PCI-Express, are aimed at addressing this issue. Their extensions, e.g. peer-to-peer communication, message passing semantics, low pin count, and scalable bandwidth make them reasonable candidates for the switch fabric interface [5]. It is, however, unclear, which of these interfaces should be supported. Not only do these interfaces represent different market alliances, but they also provide (or will as they evolve) comparable features at a similar performance/cost ratio, as Figure 1 illustrates. Thus, the question arises: Can we support these interfaces with one, sufficiently flexible solution?

The goal of this paper is to establish a methodology for clarifying trade-offs involved in implementing RapidIO, Hypertransport, and PCI-Express. We are especially interested in understanding, how much we can gain from the fact that existing network processors already deal with packet-based protocols in a flexible way.

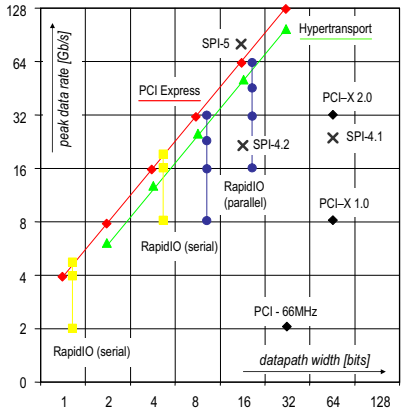

Figure 1. Bandwidth and pin count of network processor interfaces.

For this purpose, we first compare key characteristics and tasks of the protocols. Then, we describe our approach and the associated design space exploration method. Finally, we deploy the methodology to study the feasibility of implementing the communication standards on network processor building blocks. Before we conclude, we discuss related work.

2 Comparison of protocols

In Figure 1, we see the peak performances over the data pin count for RapidIO, Hypertransport, and PCI-Express. The throughput for all three protocols is comparable and scales similarly. Since all three communication protocols are packet oriented and based on point-to-point links, their interfaces are also fairly similar with respect to function and structure.

Structure. We structure the interfaces in three layers fol-

Table 1. Comparative description of the different interfaces.

	PCI-Express	RapidIO, serial	Hypertransport
Physical Layer	• Serial LVDS interface (2.5 GHz) • 1, 2, 4, 8, 16, or 32 lanes • Clock recovery, compensation • Striping and de-skewing for multiple lanes • 8/10b coding • Framing. Start/end symbol and command sequences generation. • Scrambling to reduce EMI	• Serial LVDS if (1.25, 2.5, 3.125 GHz) • 1 or 4 lanes • Clock recovery, compensation • Striping and de-skewing for multiple lanes • 8/10b coding • Framing. Start symbol and command sequences generation.	• Parallel LVDS if (0.4 - 1.6 GHz) • 2 - 32 data, 1 control, 1 - 4 clock pins • Clock compensation • Striping and de-skewing for multiple lanes • No coding • No framing performed.
Link Layer	• 16b CRC protection per packet and optional 32b CRC end-to-end • Ack/Nack protocol per link. • Packet classification and assembly • 6 Byte data link layer packets	• 16b CRC protection for data packets, 5b CRC for control symbols • Ack/Nack protocol per link. Error feedback. • Packet classification and assembly • 3 Byte data link layer packets	• 32b CRC protection for every 512 bytes (not per packet) • No hardware error recovery. • Packet classification (no assembly since packets have known sizes). • No data link layer packets
Transport Layer	• Three types of traffic: posted, non-posted and completion, with separate queues for each type. • 8 traffic priorities via virtual channels. • Data payload up to 4096 Bytes. Typical header: 12-16 Bytes. • Credit based flow control. Independently managed per traffic class. • Up to 32 outstanding transactions • Address validation • Configuration space	• No traffic type dependent storage • 4 traffic priorities with separate queues for each of them. • Data payload up to 256 Bytes. Typical header: 6 Bytes. • Global credit based flow control. • Up to 256 outstanding transactions (only 32 not acknowledged) • Address validation • Configuration space	• Three types of traffic: posted, non-posted and completion, with separate queues for each type. • Optionally, a second level of priority may be defined (isochronous). • Different packets for commands (4 or 8 Bytes) and data (up to 64 Bytes). • Credit based flow control. Independently managed per traffic class. • Up to 32 outstanding transactions per UnitID (one device may have multiple UnitIDs) • Address validation • Configuration space

lowing the OSI/ISO model and the PCI-Express specification: the *Physical Layer*, which transmits data over the physical link, the *Data Link Layer*, which manages the direct link and reliably transmits information across it, and the *Transaction Layer* which establishes communication streams, controls the flow of information and interacts with the device core.

Function. Table 1 presents a detailed comparison of the three protocols. Both PCI-Express and RapidIO define serial physical layers based on LVDS and encode clock and sideband signals into the bitstream. Hypertransport only defines a parallel version with still low pin count, which requires an additional clock and a control signal. We are not interested in the electrical differences of the PHY layers, they are addressed elsewhere [4]. For the context of this paper it is sufficient to notice the identical pin-count required for each of them.

Based on the comparison, we conclude that the three protocols share most of their functional elements, which motivates a joint architecture for supporting them all together. There are, however, certain implementation details, e.g. the packet formats, that are specific to the protocols and thus require flexibility of such an architecture. The acknowledge protocol, for instance, is one example of such differences. PCI-Express uses a sliding window scheme to confirm transactions. RapidIO requires every packet to be acknowledged individually. In case of failure, RapidIO reports the cause back to the transmitter. The current version of Hypertransport is not using the protocol.

The packet-oriented nature of our protocols relates them strongly to the network processing domain. Typical packet-processing tasks, such as bit field extraction, classification and packet forwarding, are required for our protocols as well. Thus, typical features of network processors might be reusable.

3 Design flow

In order to assess the feasibility of the three communication protocols on one common flexible IP block, we perform an exploration of implementation alternatives. The Y-Chart, shown in Figure 2 is a common scheme for this task [8]. Application and architecture specifications are independent and kept separately, so that an explicit mapping step leads to an analyzable (virtual) prototype. A consecutive profiling and performance analysis step then provides feedback for optimizing architecture and mapping.

We describe the application, i.e. the functionality of the protocols, in an architecture-independent way using the

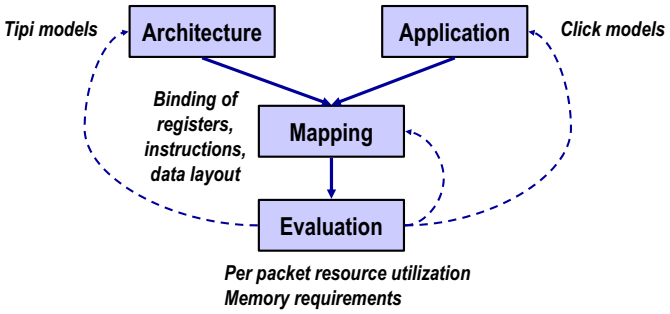

Figure 2. The Y-Chart for design space exploration using Click and Tipi.

Click model of computation [9]. Architecture models are designed in our Tipi framework [15] to provide bit- and cycle-accurate simulation results with a path to hardware generation. We start the exploration with a programmable packet processing engine as they are used in network processors.

Click for packet based interfaces. We choose Click for several reasons: Click models are executable, and capture inherent parallelism in packet flows and (in-)dependencies among elements. Click's abstraction level and the extensible element library allow us to focus on interface specifics.

In Click applications are composed from elements linked by directed connections. Elements describe common computational network operations. Connections specify the flow of packets between elements. Only packets are communicated between elements, state is kept local within elements. In order to use Click for our purposes three issues had to be resolved: 1) We explicitly share state using tokens in order to achieve the proper granularity of the elements; 2) Non-packet data types, such as state tokens and symbols (used by the physical layer), are represented by Click packets; 3) Interfaces require elements with different input and output rates, for instance, to convert link layer packets into a sequence of symbols.

Figure 3 shows the critical path of a PCI Express device [12] modeled in Click as an illustrative example. We capture the complete data- and control flow for the steady state.

Tipi to explore irregular architectures. Tipi is a correct-by-construction framework for designing architectures that can be any single-clock synchronous system [15]. In Tipi, architectures are composed in a schematic editor by connecting ports on components with relations. The architect deals primarily with the design of the data path, since the Tipi framework automatically extracts primitive operations, synthesizes required control, and generates cycle-precise and bit-true simulators and hardware descriptions.

The micro-architecture is exported to the programmer in the form of a domain-specific instruction set architecture

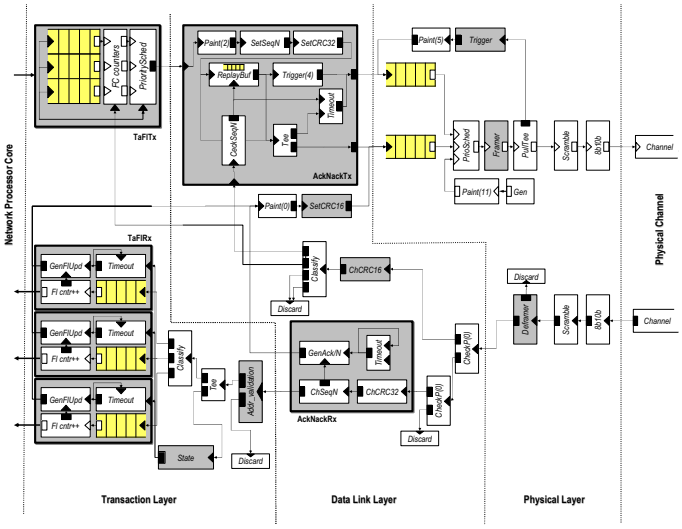

Figure 3. PCI Express end-point interface.

(ISA). In a separate mapping step, the actual data layout decisions and the register allocation take place.

Design space exploration. Based on Tipi, we cover the design space using constructive estimation, in which selected designs are individually synthesized to characterize the design space. The Tipi framework particularly supports a path-oriented search of the design space. That means, we start with a particular design, in our case a packet-processing core, and apply incremental changes to improve the performance. The exported ISA is adapted automatically by Tipi. We follow a knowledge-based approach, where we use profiling results to select customized instructions and dedicated functional units to support protocol processing.

4 Feasibility analysis

In order to derive computational requirements we profile our protocols: We annotate each of the Click elements with the number of executed assembler instructions for a packet processing engine modeled in Tipi. We then follow the different processing paths for receiving and transmitting packets. The result is a profile of executed instructions together with the number of registers used for each of these paths through the protocol layers. The profile also represents the execution time on one processing engine, assuming that there is no backlog in the queues.

4.1 Micro-architecture and instruction set

The start point for our analysis is an application-specific processing engine targeted at packet processing. Such engine clearly requires instructions for bit-level masking and logical operations [11], support for several threads of execution and a sufficient number of general-purpose registers (GPRs) per thread. The data path is 32 bit as in all ma-

jor network processors. Since we consider reliable protocols, register-mapped timers are mandatory in order to provide timed retransmissions. Fig. 4 shows part of the micro-architecture in our Tipi design framework.

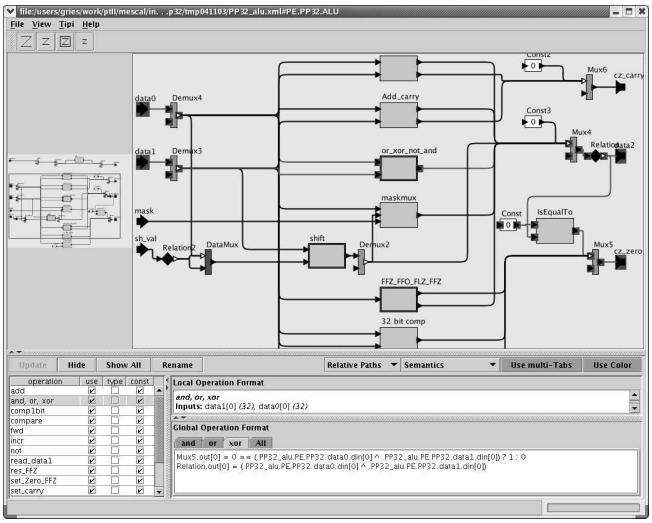

Figure 4. Processing engine in Tipi.

The following instruction classes are used to derive the execution profile of our Click elements. The application-specific, register-to-register instruction set supports bit-level masks together with logical, arithmetic, load, and store operations in order to quickly access and manipulate header fields.

- *Arithmetic operations (A):* add, sub take one cycle.
- *Logical operations (L):* and, or, xor, shift, compare take one cycle.
- *Data transfer operations:*
  - Load word (ldr) from memory: two cycles latency on embedded RAM.
  - Load immediate (ldi), move between registers (mvr): take one cycle.
  - Store word (str) to memory: three cycles latency on embedded RAM.
- *Branch instructions (B):* two cycles latency (no consideration of delay slots).

Our Click elements are annotated with a corresponding sequence of assembler instructions that are needed to perform the task of the element. These programs of course depend on particular implementation and mapping decisions.

4.2 Mapping and implementation details

Queue management. In the case of PCI-Express with packet sizes ranging from 6 to 4096 Bytes, headers are split from their payload and stored in separate queues. The packet descriptor, including the header, can be managed in statically reserved arrays, whereas the payload queues need support for segments and linked lists in order to efficiently run enqueue and dequeue operations. By contrast, the maximum size of a Hypertransport packet is only 64 Bytes. Here, we also manage payload statically. All queues are stored in the data memory of the packet processing engine.

Optimizations. Click elements mapped to the same thread of the processing engine share temporary registers, e.g. a pointer to the current packet descriptor does not need to be explicitly transferred to the next Click element. Code from Click elements within push/pull chains without branches can be concatenated so that jumps are avoided.

4.3 Results

Profiling Click elements. The instruction execution profile using the instruction classes introduced earlier for the major Click elements for PCI-Express are listed in Table 2. The profiles in the table reflect the computation requirement for one packet. The requirement for the scheduler is low, since queue operations are counted separately. Classifications are simple, since they only rely on small fields for packet and transaction types, so that they can be implemented using lookup tables. On the other hand, Ack/Nack Tx/Rx counts are high, because they include the calculation of the CRC. The CRC is implemented based on logical operations and depends on the packet length. Although we looked at a rather small packet size, we recognize that the calculation of the CRC is by far the most dominant element. Thus, the CRC implementation is the first candidate for hardware optimization. We therefore add an additional CRC unit to the Tipi implementation. The same is true for

Table 2. Execution profile for PCI-Express Click elements.

Instruction class	A	L	ldr	ldi	str	B
Click element/subfunction, 64 Byte data packet						
flow ctrl	9	18	4	0	2	0
Ack/Nack Tx	69	325	145	64	6	0
prio sched	0	1	1	0	0	0
framer	5	5	18	1	1	1
deframer	0	12	0	0	0	3
check paint	1	1	1	0	0	0
Ack/Nack Rx	67	322	148	64	1	1
classify	2	1	2	0	0	1
flow ctrl rx	8	8	6	3	5	0
Ack/Nack packet-specific						
AckNackGen	0	0	0	4	1	0
Ack/Nack Tx ack	7	9	2	0	1	1
Ack/Nack Tx nack	4	9	22	1	3	1
Flow control packet-specific						
flow ctrl update	1	4	1	4	0	0
ctrl hasRoom	6	8	4	0	2	0
ctrl newCredit	8	2	3	0	2	3
common						
Calc CRC	64	320	144	64	0	0
descr enqueue	2	2	0	0	4	0
descr dequeue	2	2	4	0	0	0
payload enqueue	3	3	1	1	16	0
payload dequeue	3	2	16	1	1	0

the framer/deframer functionality. Although much less demanding, the framer is still as complex as flow control operations, but can easily be implemented in hardware.

The analysis of the Hypertransport elements leads to the same conclusions, which is why we only provide major differences to the PCI-Express implementation in Table 3.

Table 3. Execution profile for Hypertransport.

Instruction class	A	L	ldr	ldi	str	B
flow ctrl rx	4	4	2	3	1	0
get data	2	0	2	0	8	0
end chain	1	3	1	0	0	0
payload enqueue	2	2	0	0	15	0
payload dequeue	2	2	15	0	0	0
compactor	1	9	1	0	2	0

Sensitivity on RAM access and clock speed. Using the profiling results, we now derive actual execution times for different hardware scenarios. Since the interaction with the data memory is fine-granular, the RAM timing is particularly important. We therefore look at three scenarios: (I) ideal memory which returns data within one cycle, (II) on-chip SRAM running at the core speed, e.g. employing DDR interfaces, using the latency values two and three clock cycles for reads and writes, respectively, and (III) off-chip memory with a 10 cycle access latency, representing off-chip SRAM and non-burst access. Assuming our engine to run at 1 GHz, we calculate the ratio between execution time and the inter-arrival time of packets at 2.0 GBit/s goodput, which is sketched on the vertical axis in the following diagrams. This ratio represents the ideal pipeline depth of processing elements to fulfill the throughput requirements. We calculate the execution times in dependence on different packet sizes. The results for PCI-Express receiving a data packet, the most complex application scenario, are plotted in Figure 5 as an example. This figure clearly underpins that

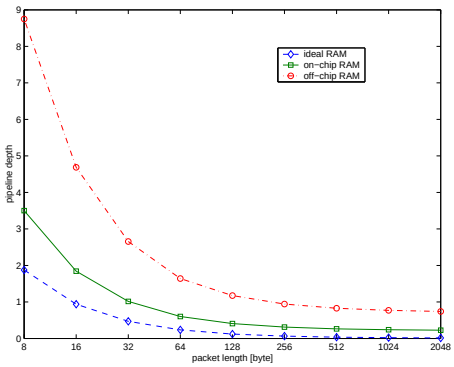

Figure 5. Rel. execution time depending on packet length and RAM access time for PCI-Express.

a tight interaction with the memory is required to get a feasible solution. As a rule of thumb, we see that one packet engine at 1 GHz is required to handle a lane at 2.5 GBit/s using on-chip memories. We also recognize that the access times to the RAM must be small and therefore multi-threading is not an option to hide latency.

For completeness, the cycle counts for all transfer scenarios for PCI-Express are listed in Table 4 for a data packet length of 64 Byte. Cases A, B, and D depend on the data packet size. In Case D, a retransmission of one data packet is initiated if a Nack packet is received. The cycle counts

Table 4. PCI-Express cycle counts by scenario.

Case A	Transmission of data packet	111
Case B	Reception of a data packet	154
Case C	Transmission of Ack/Nack packet	39
Case D	Reception of Ack/Nack packet	63 / 107
Case E	Transmission of flow control packet	39
Case F	Reception of flow control packet	103

for Hypertransport for the supported cases A, B, E, and F are within a 15% margin of the PCI-Express results.

4.4 Discussion

The results of this analysis reveal, that an instruction set specific to network processing is beneficial and mandatory for programmable solutions in our application domain. Furthermore, network processor queue management blocks can be reused, although smaller segments and minimum packet sizes must be supported. A tight coupling between data RAM and processing engine is required to support minimum-size packets with a feasible pipelining depth of processing engines. This also implies that multiple threads can no longer be used to hide memory latency. Further architecture optimization is required for better performance/cost trade-offs. The CRC, for instance, has a data-dependent execution time and is an ideal candidate for a hardware implementation.

5 Related work

The rising interest in new interconnect standards has recently been covered by several articles [5, 1, 3]. Although in [1] some details have been compared, no comprehensive picture has been drawn of the essential elements of the interfaces. A thorough survey of I/O adapters and network technologies in general is presented in [13]. The paper particularly focuses on requirements for server I/O.

A major obstacle to a flexible solution is obviously in the different physical characteristics of the protocols. This issue, however, is already addressed by initiatives such as UXPi and individual companies, such as Rambus, [10]. We are interested in the higher level interface and protocol aspects.

The choice to use Click as modeling environment has been particularly motivated by the network domain. State machine based approaches, e.g. [6], are best suited for control dominated systems and suffer from their inability to graphically express data flow. Spread sheet-based models do not provide any mapping path to hardware platforms. They are not executable and thus are confined to analysis only. The use of object-oriented modeling (see, e.g., [2]) is becoming more and more common. Although object-oriented formalisms contain several features to produce detailed models, they are not intended to be executable as such. High-level Petri nets (such as Coloured Petri Nets [7]) are also suited to model protocols, since they have an expressive inscription language and also structuring features, including hierarchy. They are graphical, able to express concurrency, and data flow. However, this expressiveness is not required to model protocols. We thus prefer to use a more restrictive model of computation by which efficient paths to implementation exist for network applications, see, e.g., [14].

6 Conclusion

The goal of this paper is to establish a method of clarifying trade-offs involved with implementing a flexible interface for popular interconnect standards, namely RapidIO, Hypertransport, and PCI-Express. In order to identify common functionality and structures, we have first modeled the behavior in Click. Based on our findings, we modeled a flexible processing architecture in Tipi similar to those in network processors. We have evaluated the feasibility of implementing the communication standards on existing network processor infrastructure by a profiling analysis. The results of our work are:

- We found Click to be natural and expressive in specifying the functionality of the communication standards investigated. Our analysis has been particularly eased by using the abstraction level of Click elements and their interaction.
- The Tipi framework has been a productive environment to model our processing element. The current version provided good support for assembler programming and manual mapping decisions. We expect even faster turnaround times as the compiler generation matures.
- The evaluation of network processor building blocks revealed the feasibility of implementing the required protocol processing. Flexible interfaces based on programmable engines can form a new family of building blocks for next generation Systems-on-chip.

In summary, our results encourage the use of Click and Tipi for the flexible implementation of our protocols. This flow enables design space exploration for determining different perfomance/cost trade-offs.

Acknowledgment

This work was supported by Infineon Technologies, the Microelectronics Advanced Research Consortium (MARCO), the Spanish government grant TIC 2002-750 and is part of the efforts of the Gigascale Systems Research Center. The authors would like to thank Andreas Herkersdorf for his invaluable feedback.

References

[1] D. Bees and B. Holden. Making interconnects more flexible. *EE Times*, Sept. 2003.

[2] G. Booch. *Object-oriented analysis and design, with applications*. Benjamin/Cummings, 2nd edition, 1994.

[3] N. Cravotta. RapidIO versus Hypertransport. *EDN*, June 2002.

[4] K. Donnelly. Common physical layer issues underlie new i/o standards. EE Times.

[5] I. Elhanany, K. Busch, and D. Chiou. Switch fabric interfaces. *IEEE Computer*, Sept. 2003.

[6] D. Harel. StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

[7] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts of *EATCS Monographs in Computer Science*. Springer-Verlag, 1992.

[8] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. 11th Int. Conf. on Application-specific Systems, Architectures and Processors, Zurich, Switzerland, 1997.

[9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3), Aug. 2000.

[10] R. Merritt. Dueling I/O efforts gear up to revamp comms. *EE Times*, Oct. 2003.

[11] X. Nie, L. Gazsi, F. Engel, and G. Fettweis. A new network processor architecture for high-speed communications. In *Workshop on Signal Processing Systems (SiPS)*, Oct. 1999.

[12] PCI Special Interest Group. PCI Express base specification, rev. 1.0a. www.pcisig.com, Apr. 2003.

[13] R. Recio. Server I/O networks past, present, and future. *ACM SIGCOMM Workshop on Network-I/O Convergence*, Aug. 2003.

[14] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A programming model for the Intel IXP1200. In *Network Processor Design: Issues and Practices*, volume 2. Morgan Kaufmann, 2003.

[15] S. J. Weber, M. W. Moskewicz, M. Loew, and K. Keutzer. Multi-View Operation-Level Design – Supporting the Design of Irregular ASIPS. Technical Report UCB/ERL M03/12, University of California, Berkeley, April 2003.