Managing Heterogeneous Processor Machine Dependencies in Computer Network Applications*

Ralph Duncan, Peder Jungck, Kenneth Ross, Jim Frandeen, and Greg Triplett

CloudShield Technologies, A Science Applications International Corporation (SAIC) Company
212 Gibraltar Drive, Sunnyvale, CA 94089 USA
rduncan@cloudshield.com

Abstract. Executing complex network packet applications typically requires using network processors and parallel processing to handle packet transmission speeds of 1 gigabit per second and beyond. Heterogeneous computing approaches also employ specialized coprocessors, such as associative memory processors for flow matching and regular expression (regex) processors for packet payload searching. Our goals for this kind of heterogeneous processing are to free application developers from hardware-specific details and to develop systems in which we can deploy new hardware and software components in modular, plug-and-play fashion. Our initial contribution to realizing these goals involves (1) expressing classic packet operations in a C dialect as C/C++-style operators; (2) compiling user code into bytecodes for a packet-processing virtual machine that hides machine-specific details; and (3) interpreting the bytecodes with microcoded interpreters that orchestrate an ensemble of heterogeneous processors on the users' behalf.

1 Introduction

As computer networks continue their relentless speed increases the need for increasing packet processing speeds moves in tandem. Since the late 1990s network processing vendors have responded to the need for speed and flexibility by using *network processors* that are optimized for classic packet processing tasks. These processors are often used in conjunction with separate, specialized *coprocessors* [1, 2]. As Douglas Comer observed in a popular textbook [1], designers have generated a variety of architectures that typically use some combination of specialized instructions, parallel processing and pipelining. His 2005 survey of commercial architectures [2] had these categories:

* Intel is a trademark of Intel Corporation in the United States and/or other countries. Broadcom® is a registered trademark of Broadcom Corporation in the United States and/or other countries. Xilinx is a trademark of Xilinx Inc. in the United States and/or other countries. OCTEON® is a registered trademark of Cavium. AMCC is a registered trademark of Applied Micro Circuits Corporation. IDT is a trademark of Integrated Device Technology, Inc. packetC® is a registered trademark of CloudShield Technologies, Inc. in the United States and/or other countries.

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 280–289, 2014.

- Embedded processors with special instructions for packet processing
- Parallel processors, assisted by specialized coprocessors
- Parallel pipelines of either homogenous processors or heterogeneous ones

Our hardware approach falls into the category for parallel processors, augmented with coprocessors. Our basic building blocks are the following:

- Network processors (*microengines* by Intel) with a reduced instruction set computer (RISC) processor as their host
- Xilinx, Inc.® Virtex® 5 field programmable gate arrays (FPGAs) to perform specialized functions e.g., *ingress processing,* to identify packet *headers*
- Ternary content addressable memory (TCAM) chips (associative memory processors) to match key header fields' contents
- Regular expression *(regex)* processors to match packet *payload* contents to strings specified as regular expressions.

We use the kinds of processors often seen in commercial systems. Thus, our contribution is not a particular ensemble of specialized processors but the use of high-level languages to isolate chip-specifics and interpreters to manage heterogeneity:

- Expressing packet operations with high-level language operators
- Compiling programs into machine-independent bytecodes for packet processinge
- Interpreting bytecode programs via multiple copies of a microcoded interpreter, running in parallel and orchestrating diverse coprocessors' actions

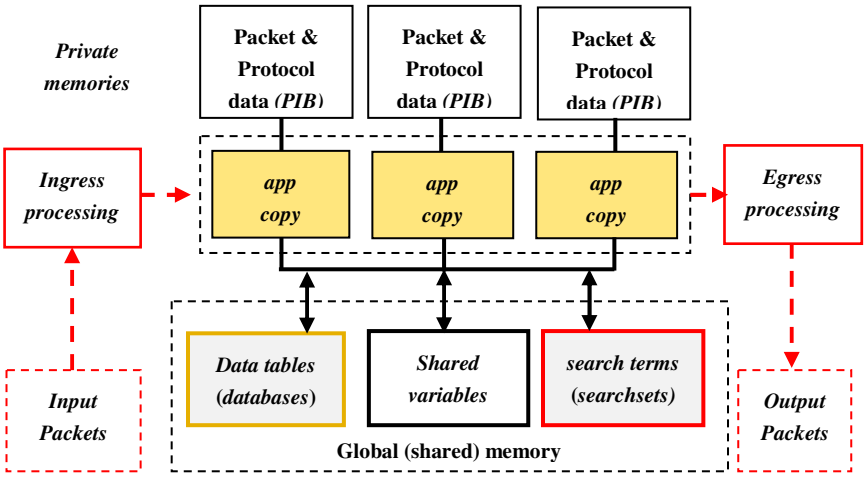

Fig. 1. Parallel packet processing model and language features (in italics). Copyright CloudShield Technologies, 2013.

The paper reviews our programming model, then traces our extended data types and operators from their high-level forms in packetC® [3] through domain-specific bytecodes to their implementation by specialized coprocessors. We present experiments then end by reviewing related work and summarizing.

2 Parallel Processing Model

Our ultimate goal is a flexible system in which we can swap one coprocessor for another or substitute a software component for a coprocessor. To support this, our model of parallel packet processing has the following characteristics.

- It uses coarse-grained parallelism at the packet level to hide machine specifics.
- The host system is expected to provide classic ingress and egress processing (see Fig. 1) but the model does not specify how this to be done.
- Capabilities are specified for matching packet header fields and searching packet payloads but their implementation is not.

Developers express parallelism at a coarse-grained level with a program that completely processes one packet at a time. Thus, we allocate each packet to a specific copy of the application (a *context*) and runs the contexts simultaneously in single program multiple data (SPMD) fashion. Using coarse-grain parallelism frees developers from fine-grain, processor-specific mechanics, like synchronizing low-level tasks.

3 packetC Language Overview

The first component of the approach consists of using high-level operators in the packetC language, rather than lower-level operators that reflect coprocessor specifics. packetC [3] extends C99 with data types and operators that provide classic packet processing functionality. Types used in the experiments section are sketched below.

Databases act like an array of structures divided into identically typed "data" and "mask" halves for *wildcarded* matching against packet contents [5] (typically, to match header fields). Users provide a base type from which the compiler constructs the data and mask halves. Users then match structures against those portions of the data half that have bits set to "on" in their corresponding mask.

```
struct stype { short dest; short src;};
database stype myDB[50];
// myDB element layout is { stype data; stype mask;};
rownum = myDB.match( myStruct ); // do masked search
```

Searchsets are aggregates of strings or regular expressions that are matched against the contents of the packet *payload* via C++-style *methods* [6]. The first of the searchset items to be found is reported in a result structure with position information.

```
searchset pets[3][3] = {"cat", "dog", "owl" };
SearchResult ansStruct;
try {  // match vs. a 3-byte slice of packet
   ansStruct = pets.match( currPkt[64:66] );
}
catch ( ERR_SET_NOMATCH ) {…}
```

4 Translation to Bytecodes

Our approach's second key consists of translating high-level packetC operations into machine-independent *bytecodes* (Fig 2). Just as packetC hides processor specifics from application developers, the bytecodes hide them from the tool-chain and runtime support system to isolate the effects of replacing accelerators with new processors or software equivalents. A single bytecode may express a complex operation that will be implemented by a specialized coprocessor. Due to space constraints we show only a human-readable version of the database match bytecode as a representative example.

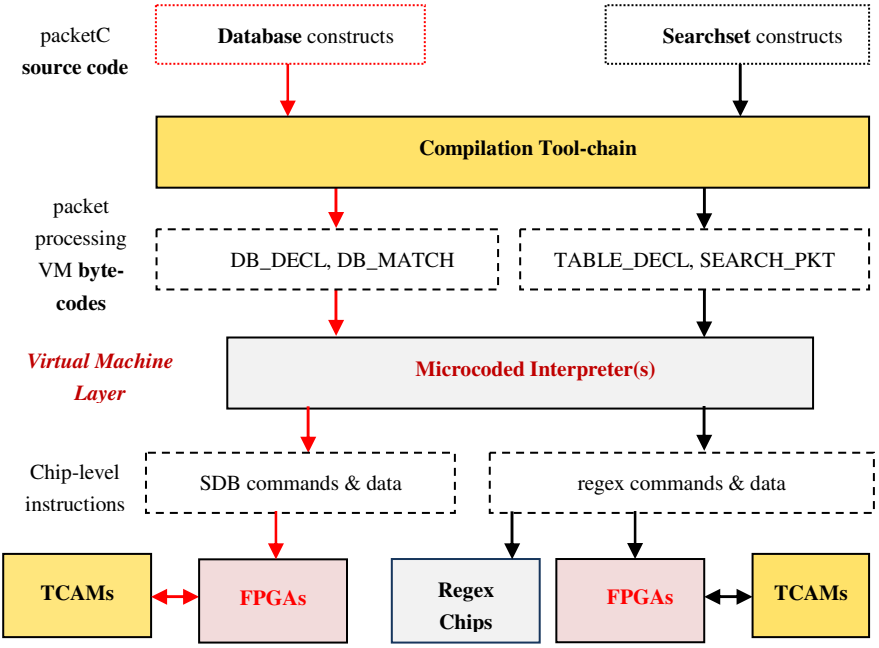

Fig. 2. Example bytecode translation flow for databases and searchsets. Copyright Cloud-Shield Technologies, 2013.

In the example below, a packetC *record* – a structure with data and mask halves – is matched against a database of like elements via a complex DBMATCH bytecode.

```
// packetC code – database match
struct stype { short dest; short src;};
database stype myDB[50];
rownum = myDB.match( myRec ); // match contents of myDB
// Bytecode – database decl, then MATCH operation
DBASE_DECL  myDB [500] 5 // 5 is the ID
DB_MATCH Id, maskKind, inData, inMask, rownum
//    Id: integer indicating with DB to use
//    maskKind: scenario indicator
//    inData: data to match
//    inMask: to combine w/ each element mask
//    rownum: 32bit integer to rec matching DB row #
```

The next section describes how we use interpreters to turn the bytecodes above into coprocessor actions.

5 Interpreter as Heterogeneous Processor Orchestrator

Our approach's third component consists of using microcoded interpreters to execute bytecodes and encapsulate coprocessor specifics. Such interpreters can effectively exploit machine-level resources and fine-grain operations to implement our bytecodes. For example, the interpreters exploit various data pathways to control specialized processors. Interpreters also isolate coprocessor-specific aspects of the communication. Specifically, an interpreter

- Sends a coprocessor input operands in some required format
- Sends an indication of which operation to perform
- Receives results and coprocessor error information, again, in prescribed formats

The complexity of these operations may vary considerably, depending on whether

- Operands consist of a small number of scalars or an aggregate (such as the packet *payload* treated as a byte array)
- The coprocessor performs a single operation or many (which must therefore be distinguished in communications)

An appreciation of how such dependencies and of how a bytecode interpreter manages them is best gained by concrete example. Thus, the next section describes one of our fielded system's processor components and shows in considerable detail how an interpreter drives operations on two coprocessor (accelerator) systems.

6 Implementing Interpreter and Coprocessor Communication

Our products, such as the PN41 [7], use multi-core network processing units (NPUs) optimized for packet operations. The PN41 uses an IXP 2800 with 16 *microengines* as its NPU Although the IXP is often programmed in fine-grain parallel fashion [8], we use the coarse-grained approach described in section 2, with 95 microengine *contexts* running an interpreter and other contexts providing house-keeping functions.

These interpreters interact with multiple kinds of coprocessor to implement operations originally coded as packetC operators, then translated to bytecode form.

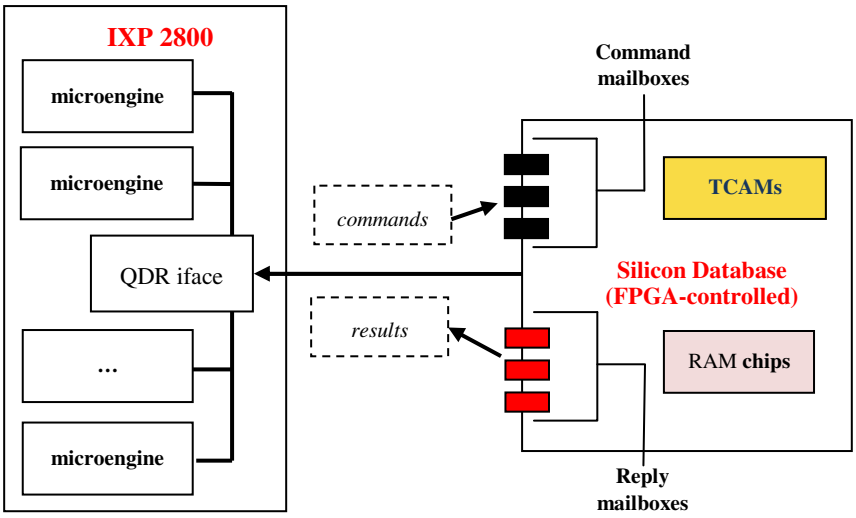

Fig. 3. PN41 Communications between IXP-based interpreters and TCAM chips. Copyright CloudShield Technologies, 2013.

In our first example a Xilinx, Inc.® Virtex® 5 FPGA controls a *silicon database* (SDB) that uses Broadcom NL 5512 TCAM chips to match packetC *databases*. When an interpreter encounters a **match** operation, these actions occur (Fig. 3):

- The interpreter uses a *quad data rate* (QDR) interface to send a match command (in a control language) to a command mailbox associated with the SDB controller.
- The interpreter passes the data to compare and ID of the database to use.
- Depending on the operation, the interpreter may pass a bit mask to use.
- The SDB replies with acknowledgement signals and the results data.

The TCAM performs a given match on all the relevant database entries in parallel, a significant performance benefit. However, moving operands and results back and forth, as well as managing the operation can consume significant time. In practice, optimizing this data movement is an art and science in its own right.

Our second example involves using IDT PAX.port 2500 *content inspection engines* (regex processors) to implement packetC *searchset* **find** operations, typically to search for terms in the packet's *payload* [10]. Such a searchset is an array composed entirely of strings or regular expressions.

Searchset contents are precompiled into a set of regex rules and loaded into a *pattern memory*. As Fig. 4 shows, an interpreter running on a microengine ships command data and the bits within which to search (typically, a slice of the packet payload) to the subsystem via another specialized FPGA. The regex subsystem returns the index of the first searchset element to be found (if any), as well as location data.

The next section presents two experiments that show that this kind of microcoded interpretation with coprocessors achieves high speeds.

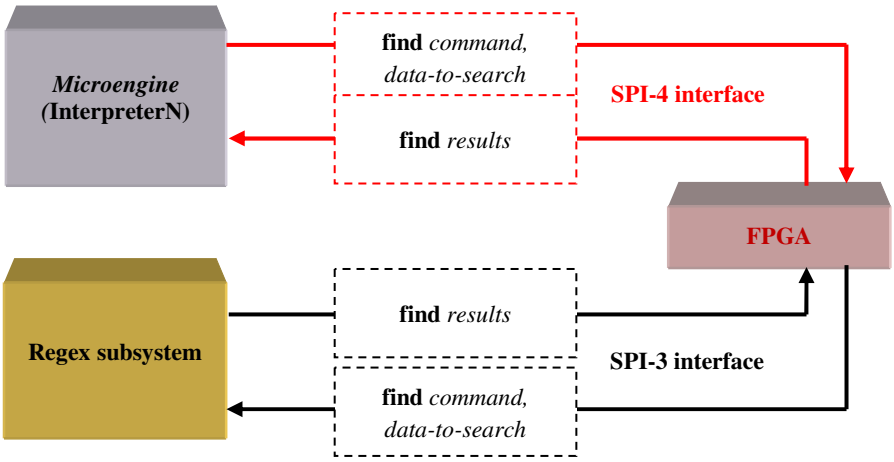

Fig. 4. Searching packet data via a regex system. Copyright CloudShield Technologies, 2013.

7 Experiments Emphasizing Coprocessors

The experiments used a CloudShield PN41 [7] 10 gigabit Ethernet blade to run packetC applications and an IXIA® XM12 traffic generator [11] to produce network traffic at a maximum of 10 gigabits per second (Gbps). Each experiment involved microcode interpreting bytecodes and invoking coprocessors for key operations.

Fig 5(a) shows an experiment that matched a sequence of bytes from a specific packet payload location of each packet against a packetC *database* of 30,000 stored patterns. This database and the high-level **match** operation used the bytecodes and TCAM subsystem shown in sections 4 and 6 above (see also Fig. 3). For this scenario the system produces a maximum output rate of 8.85 Gbps. (See [6] for details).

The experiment shown in Fig 5(b) involved searching each entire packet for the presence of one or more patterns, defined by a packetC *searchset* of 10,000 simple patterns – each containing a wild card substring. This searchset and the high-level **find** operation used the bytecodes and regex subsystem shown in sections 4 and 6 above (see Fig. 4). The maximum speed achieved, 5.9 Gbps, is expected, since the regex subsystem has a maximum speed of roughly 5 Gbps and most of this application's time is spent in the subsystem. For further details, the packetC source constructs and related experiments are described in an analysis of searchsets [6].

A traditional presentation might report comparative metrics for alternative techniques, typically from experimental prototypes. In this paper we report the data points that we have, which are from a commercial product line with more than 10 years in the fields. Our claim is not that techniques we describe are superior in performance to particular alternatives but, instead, that the portability and retargetability benefits of this approach are gained while performing deep packet inspection (DPI) at speeds of two to nine Gbps, which we believe to be state-of-the-practice DPI at this time.

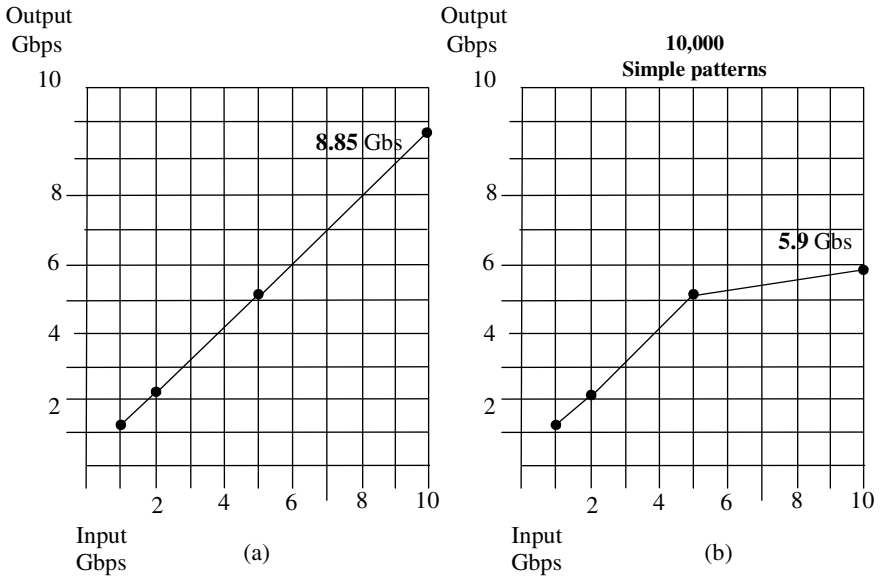

Fig. 5. (a) Using a TCAM coprocessor to match a bit pattern. (b) Using a regex coprocessor to search for string matches. Copyright CloudShield Technologies, 2013.

8 Related Work

Intel Corporation®'s Microengine C [8] is a C dialect targeted to the IXP network processor family. Its processing model involves breaking programs into multiple threads that are partitioned across *microengines*. The user manages communications among threads and swaps out tasks. Microengine C reflects a variety of machine-specific IXP features, such as memory and register classes.

J. Wagner and R. Leupers describe a processor-specific C dialect geared to the Infineon network processor in [12]. Language's extensions let the user map protocol headers to special registers and manipulate those values via operands with arbitrary bit-widths. A collection of compiler *intrinsics* lets users exploit these capabilities.

L. George and M. Blume described the NOVA language for the IXP network processor in [13]. NOVA has features for specifying header representation, including a *layout* construct for packed and unpacked forms and an *overlay* construct for alternative organizations within a layout. Network Protocol Description Language Vin, et al. proposed the Baker programming language [14], a C dialect augmented by data-flow concepts, which emphasized pipelining packet data from function to function.

Cavium Networks, Inc. makes Octeon® networking and communications processor families [15] that use specialized processors for regular expression processing and fast matching. Their software approach appears to use C/C++, augmented by application programming interfaces (APIs) and libraries. AMCC® has also produced network processors (e.g., nP3700) that feature multiple network processing cores and specialized coprocessors, including regex processors [16] and TCAMs [17].

MicroEngineC and C for the Infineon® chip extend C to include machine-specific details, whereas packetC disguises machine particulars. Like AMCC and Cavium, our architecture uses multi-core processing engines, supplemented by coprocessors. However, we use different approaches to translating and running application programs on those coprocessors. Although we cannot be certain, our approach appears to be the only one on current, packet processing systems that uses microcoded interpreters or high-level bytecodes to trigger coprocessor operations for classic network operations.

9 Summary

The key elements of our approach to parallel, heterogeneous programming are

- Expressing user operations in terms of high-level packetC operators that act on extended, C-style data types.
- Compiling user programs to machine-independent bytecodes that describe a packet processing virtual machine.
- Using microcoded interpreters to process bytecodes and trigger coprocessor operations as needed.

packetC high-level operators have proven easy to use and to teach. Both packetC and bytecodes have survived chipset and NPU changes without requiring tool-chain redesign or user code recompilation. The interpreter scheme requires detailed microcode, NPU and coprocessor expertise but only from tool-chain developers, not application code developers. Freed from arcane processor details, users can concentrate on solving problems in terms of high-level networking application domain constructs.

We believe the techniques described above, both individually and in combination, offer significant benefits to other heterogeneous processing practitioners.

Acknowledgements. Mary Pham, Scott Tillman, Greg Triplett, Jim Frandeen and Minh Nguyen made key contributions to coprocessor subsystems, loaders, and the interpreters. The export approval number for the paper is 13-SAIC-0517-1012.

References

1. Comer, D.: Network Systems Design Using Network Processors, Intel 2xxx Version. Prentice Hall, Upper Saddle River (2006)
2. Comer, D.: Network Processors: Programmable Technology for Building Network Systems. The Internet Protocol Journal 7(4) (2004)
3. Jungck, P., Duncan, R., Mulcahy, D.: packetC Programming. Apress, New York (2011)
4. Duncan, R., Jungck, P., Ross, K.: A paradigm for processing network protocols in parallel. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010, Part I. LNCS, vol. 6082, pp. 52–67. Springer, Heidelberg (2010)
5. Duncan, R., Jungck, P., Ross, K.: packetC language and parallel processing of masked databases. In: Proceedings of the 39th Intl. Conf. on Parallel Processing, San Diego, USA, September 13-16, pp. 472–481. IEEE (2010)

6. Duncan, R., Jungck, P., Ross, K., Tillman, S.: Packet Content matching with packetC Searchsets. In: Proceedings of the 16th Intl. Conf. on Parallel and Distributed Systems, Shanghai, China, December 8-10, pp. 180–188 (2010)
7. International Business Machines Corporation. IBM Blade Center PN41. Product datasheet available from IBM Systems and Technology Group, Route 100, Somers, New York, USA 10589 (2008)
8. Johnson, E.K., Kunze, A.R.: IXP 2400/2800 Programming: The Complete Microengine Coding Guide. Intel Press, Hillsboro (2003)
9. Broadcom Corporation, NL5064, NL5128, NL5256 and NL5512 Proprietary Interface, Knowledge-based Processors, http://www.broadcom.com/products/Knowledge-Based-Processors/Layers-2–4/NL5000-Family (retrieved on April 30, 2013)
10. Chao, H.J., Liu, B.: High Performance Switches and Routers, pp. 562–564. John Wiley and Sons, Hoboken (2007)
11. XIA. XM12 High Performance Chassis, http://www.ixia.com.com/pdfs/datasheets/ch_optixia_xm12.pdf (retrieved on July 25, 2013)
12. Wagner, J., Leupers, R.: C compiler design for a network processor. IEEE Trans. on CAD 20(11), 1–7 (2001)
13. George, L., Blume, M.: Taming the IXP network processor. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, California, USA, pp. 26–37. ACM (June 2003)
14. Vin, H.M., Mudigonda, J., Jason, J., Johnson, E.J., Ju, R., Kunze, A., Lian, R.: A Programming Environment for Packet-processing Systems: Design Considerations. In: Network Processor Design: Issues and Practices, vol. 3, pp. 145–173. Morgan Kaufmann (2004)
15. Cavium Networks, Inc. Octeon Multi-Core Processor Family, http://www.cavium.com/OCTEON_MIPS64.html (retrieved December 13, 2012)
16. Integrated Device Technology, Inc. IDT Strengthens Collaboration with AMCC by Delivering NSE Development Kit with AMCC Network Processor Evaluation System. News release of (March 1, 2004), http://ir.idt.com/releasedetail.cfm?ReleaseID=414510 (retrieved December 12, 2012)
17. Melnick, R., Morris, K.: AMCC nPcore NISC Architecture. In: Franklin, M.A., Crowley, P., Hadimioglu, H., Onufryk, P.Z. (eds.) Network Processor Design: Issues and Practices, vol. 2, pp. 327–342. Morgan Kaufmann (2003)