

A Programming Environment for Network Processors

Michael E. Kounavis, Andrew T. Campbell, Stephen T. Chou and John B. Vicente
COMET Group, Columbia University, New York, NY 10025, USA

Abstract

There is growing interest in network processor technologies capable of processing packets at line rates. In this paper, we present the design, implementation and evaluation of *NetBind*, a programming environment for constructing data paths in network processor-based routers. NetBind balances the flexibility of network programmability against the need to process and forward packets at line speeds. To support dynamic binding of components with minimum addition of instructions in the critical path, NetBind modifies the machine language code of components at run time. To support fast data path composition, NetBind reduces the number of binding operations required for constructing data paths to a minimum set so that binding latencies are comparable to packet forwarding times. Data paths constructed using NetBind seamlessly share the resources of the same network processor. The NetBind source code described and evaluated in this paper is freely available on the Web [15] for experimentation.

1. Introduction

Recently, there has been a growing interest in network processor technologies [1-4] that can support software-based implementations of the critical path while processing packets at high speeds. Network processors use specialized architectures that employ multiple processing units to offer high packet-processing throughput. We believe that introducing programmability in network processor-based routers is an important area of research that has not been fully addressed as yet. The difficulty stems from the fact network processor-based routers need to forward minimum size packets at line rates and yet support modular and extensible data paths. Typically, the higher the line rate supported by a network processor-based router the smaller the set of instructions that can be executed in the critical path.

Data path modularity and extensibility requires the dynamic binding between independently developed packet processing components. While code modularity and extensibility is supported by programming environments running in host processors (e.g., high level programming language compilers and linkers), such capability cannot be easily offered in the network. Traditional techniques for realizing code binding, (e.g., insertion of code stubs or indirection through function tables), cannot be applied to network processors because these techniques introduce considerable overhead in terms of additional instructions in the critical path. One solution to this problem is to optimize the code produced by a binding tool, once data path composition has taken place. Code optimization algorithms can be complex and time-consuming, however. We believe that a binding tool for network processor-based routers needs to balance the flexibility of network programmability against the need to process and forward packets at line rates. This poses significant challenges.

In this paper, we present the design, implementation and evaluation of *NetBind*, a high performance, flexible and scalable programming environment for creating composable data paths in network processor-based routers. By “high performance” we mean that NetBind can produce data paths that forward minimum size packets at line rates without introducing significant overhead in the critical path. NetBind modifies the machine language code of components at run time, directing the program flow from one component to another. In this manner, NetBind avoids the addition of code stubs in the critical path.

By “flexible” we mean that NetBind allows data paths to be composed at a fine granularity from

components supporting simple operations on packet headers and payloads. NetBind can create packet-processing pipelines through the dynamic binding of small pieces of machine language code. A *binder* modifies the machine language code of executable components at run-time. As a result, components can be seamlessly merged into a single code piece. For example, in [14] we show how Cellular IP [6] data paths can be composed for network processor-based radio routers.

By “scalable” we mean that NetBind can be used across a wide range of applications and time scales. In order to support fast data path composition, NetBind reduces the number of binding operations required for constructing data paths to a minimum set so that binding latencies are comparable to packet forwarding times. In NetBind, data path components export symbols, which are used during the binding process. The NetBind binding algorithm does not inspect every instruction in the data path code but only the symbols exported by data path components. While the design of NetBind is guided by a set of general principles that make it applicable to a class of network processors, the current implementation of the tool is focused toward the Intel IXP1200 network processor. Porting NetBind to other network processors is for future work.

This paper is structured as follows. In Section 2 we discuss issues and design choices associated with dynamic binding. Specifically, we investigate tradeoffs that are associated with different design choices and discuss their implications on the performance of the data path. In Section 3, we present the design and implementation of NetBind. NetBind can create multiple data paths that can share resources of the same IXP1200 network processor. In Section 4, we use a number of NetBind created IPv4 [5] data paths to evaluate the performance of the system. We also evaluate the MicroACE system [7] developed by Intel to support binding between software components running on Intel IXP1200 network processors, and identify pros and cons in comparison to NetBind. In Section 5, we provide some concluding remarks.

2. Dynamic Binding in Network Processors

2.1 Network Processors

A common practice when designing and building high performance routers is to implement the fast path using Application Specific Integrated Circuits (ASICs) in order to avoid the performance cost of software implementations. ASICs are usually developed and tested using Field Programmable Gate Arrays, which are arrays of reconfigurable logic. Network processors represent an alternative approach to ASICs and FPGAs, where multiple processing units, (e.g., the microengines of Intel's IXP1200 [2] or the dyadic protocol processing units of IBM's PowerNP NP4GS3 [3]) offer dedicated computational support for parallel packet processing. Processing units often have their own on-chip instruction and data stores. In some network processor architectures, processing units are multithreaded. Hardware threads usually have a separate program counter and manage a separate set of state variables. However, each thread shares an arithmetic logic unit and register space. Network processors do not only employ parallelism in the execution of the packet processing code, rather, they also support common networking functions realized in hardware, (e.g., hashing [2], classification [3] or packet scheduling [2-3]). Network processors typically consume less power than FPGAs and are more programmable than ASICs.

2.2 Dynamic Binding Issues

In what follows, we discuss issues associated with the design of a programming environment for network processor-based routers supporting dynamic binding. The main issues associated with the design of a binding system for network processor-based routers can be summarized as:

- Headroom limitations;
- Register space and state management;
- Choice of the binding method;

- Data path isolation and admission control;

2.2.1 Headroom Limitations

Line rate forwarding of minimum size packets (64 bytes) is an important design requirement for routers. Given that the line speeds at which network processor-based routers operate are high, (e.g., in the order of hundreds of Mbps or Gbps), the amount of instructions that can be executed in the critical path is typically small, ranging between some tens to hundreds of instructions. The amount of instructions that can be executed in the critical path beyond minimal IPv4 forwarding is often called the *headroom*. Headroom is a precious resource in programmable routers. An efficient binding technique needs to minimize the amount of additional instructions introduced into the critical path. The code produced by a good dynamic binding tool should be as efficient, and as optimized, as the code produced by a static compiler or assembler.

2.2.2 Register Space and State Management

The exchange of information between data path components typically incurs some communication cost. The performance of a modular data path depends on the manner in which the components of the data path exchange parameters between each other. Data transfer through registers is faster and more efficient than memory operations. Therefore, a well-designed binding tool should manage the register space of a network processor system such that the local and global state information is exchanged between components as efficiently as possible.

For fast data path composition, register addresses need to be known to component developers in advance. A component needs to place parameter values into registers so they can be correctly accessed by the next component in the processing pipeline. There are two solutions to this problem. First, the binding mechanism can impose a consensus on the way register sets are used. Each component in a processing pipeline can place parameters into a predetermined set of registers. The purpose of each register, and its associated parameter, can be exposed as a programming API for the component. A second solution is more computationally intensive. The binding tool can scan all components in a data path at run time and make dynamic register allocations when the data path is constructed or modified. In this case, the machine language code that describes each component needs to be modified reflecting the new register allocations made by the binding tool. Such code optimization algorithm may be time consuming, and not suitable for applications requiring fast data path composition.

2.2.3 Choice of the Binding Method

Apart from the manner in which parameters are exchanged between components, the choice of the binding technique significantly impacts the performance of a binding algorithm. There are three methods that can be used for combining components into modular data paths. The first method is to insert a small code stub that implements a *dispatch* thread of control into the data path code. The dispatch thread of control would direct the program flow from one component to another based on a global binding state, and, on the parameters returned by each module. This method is more appropriate for static rather than dynamic binding and can impact the performance of the data path because of the overhead associated with the insertion of a dispatch code stub.

An enhancement on the first method for dynamic binding adds a small *vector table* to memory. The vector table contains the instruction store addresses where each component is located. A data path component obtains the address of the next component in the processing pipeline in one memory access time. In this approach, no stub code needs to be inserted in the critical path. The third binding method is more interesting. Instead of deploying a dispatch loop, or using a vector table, the binding tool can modify the components' machine language code at run time, adjusting the destination

addresses of branch instructions. No global binding state needs to be maintained with this approach. Each component can function as an independent piece of code having its own “exit points” and “entry points”. Exit points are instruction store addresses of branch instructions. These branch instructions make the program flow jump from one component to another. Entry points are instruction store addresses where the program flow jumps. The impact of this approach on network processor headroom can be significant since instructions that check global binding state are omitted.

2.2.4 Data Path Isolation and Admission Control

To forward packets without disruption, data paths sharing the resources of the same network processor hardware need to be isolated. In addition, an admission control process needs to ensure that the resource requirements of data paths are met. Resource assignments can be controlled by a system-wide entity. Resources in network processors include bandwidth, hardware contexts, processing headroom, on-chip memory, register space, and instruction store space.

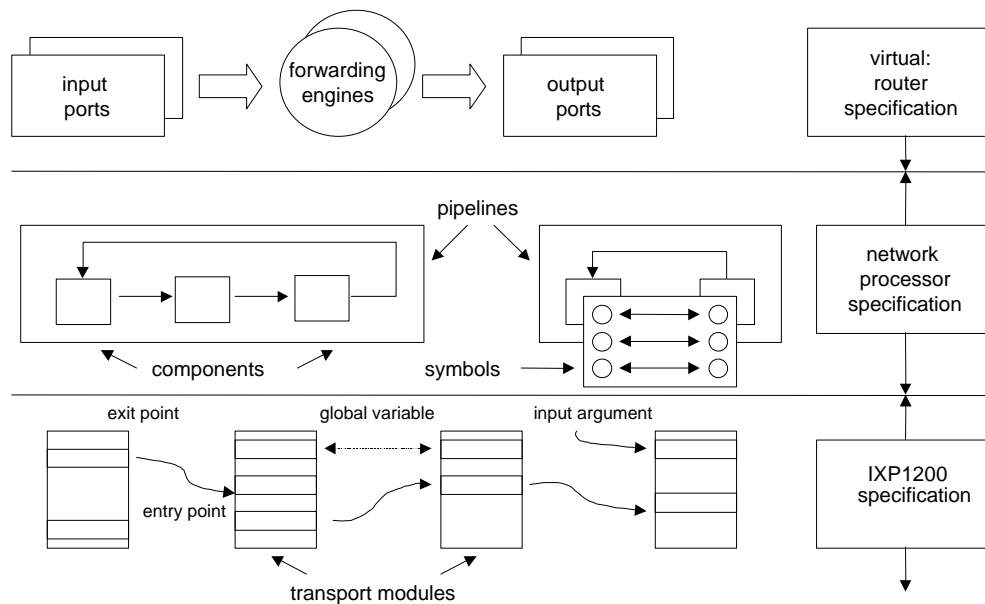


Figure 1: Data Path Specification Hierarchy

3. NetBind System

NetBind is a programming Environment we have developed that offers dynamic binding support for the IXP1200 network processor. Components are written in machine language code called *microcode* and grouped into processing pipelines that execute in the IXP1200 microengines. In what follows, we present the design and implementation of NetBind.

3.1 Data Path Specification Hierarchy

Before creating data paths, NetBind captures the structure and building blocks of data paths in a set of executable profiling scripts. NetBind uses multiple specification levels to capture the building blocks of packet forwarding services and their interaction. Figure 1 illustrates the different ways data paths are profiled in NetBind. First, a *virtual router specification* can be applied to any hardware architecture. The virtual router specification describes a virtual router as a set of input ports, output ports and forwarding engines. The components that comprise ports and engines are listed, but no additional information is provided regarding the contexts that execute the components and the way components create associations with each other. There is no information about timing and

concurrency in this specification.

A *network processor specification* augments the virtual router specification with information about the number of hardware contexts that execute components and about component bindings. The network processor specification exposes information about the hardware contexts that execute data paths in order to allow the programmer to control the allocation of computational resources in a network processor architecture (i.e., hardware threads and processing units).

Components are grouped into processing *pipelines*. A processing pipeline is a set of components that are executed by the same hardware contexts sequentially. Components exchange packets between each other in a “push” or a “pull” manner. Components that sequentially exchange packets in a push manner can be grouped into the same pipeline. A data path is split between at least two pipelines if components perform different operations on packets simultaneously. For example, components may need to place packets (or pointers to packets) into memory, while other components may need to concurrently process or remove packets from memory. In this case, the first set of components should be executed by a separate set of hardware contexts other than those, which execute the second set.

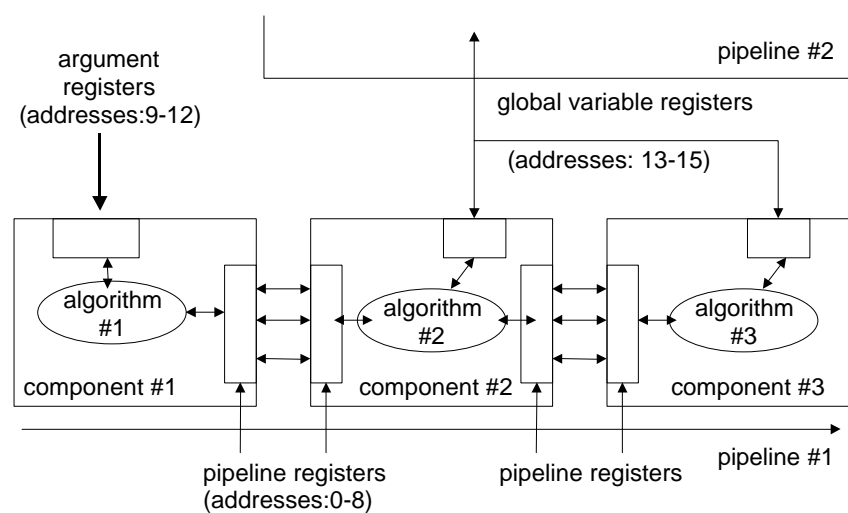


Figure 2: Register Allocations in NetBind

In the network processor related specification, components are augmented with *symbols* that are represented as strings. The profiling script specifies one-to-one bindings between symbols and between components. Symbols abstract binding properties of components such as register or instruction store addresses, which are used in the binding process.

An *IXP1200 specification* relates the components of a programmable data path with binding properties associated with the IXP1200 architecture. This type of specification shows how data paths are constructed for a specific network processor. Components are implemented as blocks of instructions (microwords) called *transport modules*. Each transport module supports a specific set of functions. Transport modules can be customized or modified during the binding process. Symbols are specified as “entry points”, “exit points”, “input arguments” or “global variables”.

Exit points are instruction store addresses of branch instructions. These branch instructions make the program flow jump from one transport module to another. Entry points are instruction store addresses where the program flow jumps. Input arguments are instruction store addresses of “immed” IXP assembler instructions [2] that load GPRs with numeric values. These numeric values, (e.g., Cellular IP timers, IPv4 interface addresses), customize the operation of transport modules. Global variables are GPRs that are accessed using the absolute addressing mode. These GPRs hold numeric values that

are shared across pipelines or data paths. For example, the SRAM address of a packet buffer in an IPv4 data path needs to be declared as global variable since this value is shared between the packet buffer and a scheduler.

3.2 Register Allocations

Register allocation realized in NetBind is shown in Figure 2. We observe that in data path implementations we have experimented with, registers are used in four ways. First, registers can hold numeric values used by a specific component. Each component implements an algorithm that operates on numeric values. When a component creates new values, it replaces old numeric values with the new ones in the appropriate microengine registers. We call these registers *pipeline registers*. The algorithm of each component places some numeric values into pipeline registers, which are used by the algorithm of the next component in the pipeline. In this manner pipeline registers are shared among all components of a pipeline. Pipeline registers are accessed using the context relative addressing mode. Once a component executes it becomes the owner of the entire set of pipeline registers. In this way, the registers used by different contexts are isolated.

Second, registers can hold input arguments. Input arguments are passed dynamically into components when pipelines are created from an external source, (e.g., the control unit of a virtual router [14]). Each component can place its own arguments into *input argument registers* overwriting the input arguments of the previous component. Similar to pipeline registers, input argument registers are accessed using context relative addressing. Third, some registers are shared among different pipelines or data paths. We call these registers *global variable registers*. Global variable registers are exported as global variable symbols. These registers need to be accessed by multiple hardware contexts simultaneously. For this reason, global variable registers are accessed using absolute addressing.

To reduce the time required for performing data path composition, NetBind uses static register allocation for pipeline and input argument registers and dynamic allocation for global variable registers. By static register allocation, we mean that register addresses are known to component developers and exported as a programming API for each component. By dynamic register allocation, we mean that register addresses are assigned at run time when data paths are created or modified. An admission controller assigns global variable registers to data paths on-demand. The microcode of components is modified to reflect the register addresses that have been allocated to components in order to hold global variables. NetBind uses static register allocation for pipeline and input argument registers in order to simplify the binding algorithm, and, to reduce the time needed for the creation a modular data path.

3.3 Binding and Admission Control

Figure 3 illustrates an example how NetBind performs dynamic binding. In this example, two components are placed into an instruction store. Figure 3 shows the instruction store containing the components and the instructions of components, which are modified during the binding process. The fields of instructions, which are modified by NetBind, are illustrated as shaded boxes in the figure. First, NetBind modifies the microwords that load input arguments into registers. Input argument values are specified using the NetBind programming API. Input argument values replace the initial numeric values used by the components. In the example of Figure 3, two pairs of “immed_w0” and “immed_w1” instructions are modified at run time, during the steps (1) and (2), as shown in Figure 3. The values of input arguments introduced into the microcode are 0x20100 and 0x10500 for the two components, respectively.

Second, the binder modifies the microwords where global variables are used. The “alu” instructions shown in Figure 3 load the absolute registers @var1 and @var2. The absolute registers @var1 and @var2 are global variable registers. An admission controller assigns the addresses of these global variable registers before binding takes place. The binder then replaces the addresses that are initially

used by the programmer for these registers, (i.e., 45 and 46 as shown in Figure 3), with a value assigned by the admission controller (47). In this manner, pipelines can use the same GPR for accessing shared information (step 3 in Figure 3).

Third, the binder modifies all branch instructions that are included in each transport module. The destination addresses of branch instructions are incremented by the instruction store addresses where transport modules are placed. Finally, the microwords that correspond to the exit points of transport modules are modified so that the program flow jumps into their associated entry points (step 4 in Figure 3). By modifying the microcode of components at run time, NetBind can create processing pipelines that are optimized adding little overhead in the critical path. This is a key property of the NetBind system that we discuss in the evaluation section.

In NetBind, admission control drives the assignment of system resources including registers, memory, instruction store space, headroom and hardware contexts. NetBind applies a simple “best fit” bin-packing algorithm to determine the most suitable microengines where data path components should be placed. If the “best fit” algorithm fails NetBind applies exhaustive search. Exhaustive search is a technique associated with significant complexity (i.e., $O(m!)$ as a function of the number of pipelines in the system). However, exhaustive search works effectively for small number of pipelines and can result in efficient resource allocations when the best fit algorithm fails.

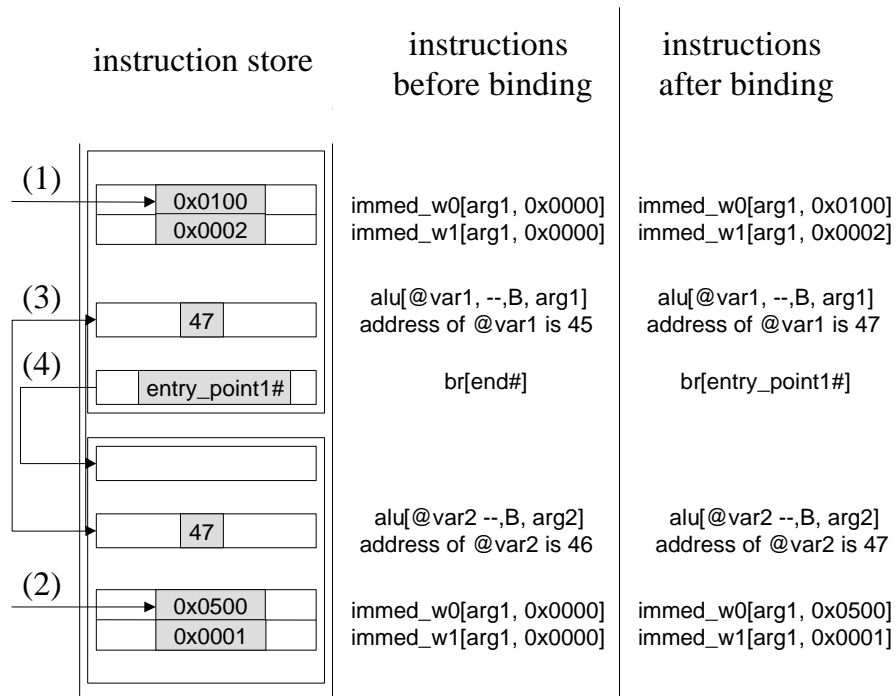


Figure 3: Dynamic Binding in NetBind

4. Realizing NetBind

We have implemented NetBind in C and C++ as a user space process in the StrongARM Core processor o IXP1200. The StrongARM Core processor runs a embedded ARM version of Linux. Transport modules can be developed using tools such as the Intel Developer Workbench [18]. The IXP1200 microassembler encapsulates the microcode associated with a component project into a “uof” file. The UOF file format is an Intel proprietary format. The UOF file format is suitable for

encapsulating statically compiled microcode, but does not include, dynamic binding information in the header. For this reason, we introduced a new file format, the TMD (Transport MoDule) format for encapsulating microcode. TMD header includes symbol information about input arguments, global variables entry points and exit points. For each symbol a symbol name, a value and an instruction store address where the symbol is used are provided. To evaluate NetBind, we have set up an experimental environment consisting of three “Bridalveil” development boards, interconnecting desktop and notebook computers in our lab. Bridalveil is an IXP1200 network processor development board running at 200 MHz and using 256 Mbytes of SDRAM off-chip memory. We evaluated the performance of a modular IPv4 data path discussed in Section 3.4 when no binding is performed (i.e., the data path is monolithic) and when the data path is created using NetBind and Inter’s MicroACE [17] programming environment. MicroACE follows a dispatch loop approach where some global binding state needs to be checked before each component is executed. We also implemented a simple binding tool based on the vector table binding technique, as discussed in Section 2. To analyze and compare the performance of different data paths, we executed these data paths on Intel’s “transactor” simulation environment and on the IXP1200 Bridalveil cards.

We observed that the dispatch loop binding technique, used by MicroACE, introduces the largest overhead, while the NetBind code morphing technique and the vector table technique demonstrate smaller overhead. The overhead of the worst case dispatch loop for a six component data path is 89 machine cycles which represents 36% of the network processor headroom. NetBind demonstrates the best performance in terms of binding overhead adding only 18 execution cycles when connecting six modules to construct the IPv4 data path. We have also measured the time to install a new data path using NetBind. Stopping and starting the microengines takes 60 μ s and 200 μ s respectively. The binding algorithm and the process of writing data path components into instruction stores takes 400 μ s to complete. Measurements were taken using the Bridalveil’s 200MHz StrongArm core processor. Loading six modules from a remotely mounted NFS server takes about 60ms to complete. Since the IXP1200 network processor prevents access to its instruction stores while the hardware contexts are running, we had to stop the microengines before binding, and restart them again after the binding was complete. We are currently studying better techniques for dynamic placement without disruption to executing data paths.

5. Conclusion

We presented the design, implementation and evaluation of the NetBind programming environment. While the community has investigated techniques for synthesizing kernel code [20] and constructing modular data paths and services [8, 9, 10, 13], the majority of the literature has been focused on the use of general-purpose processor architectures. Little work has investigated the development of programming environment for network processors. Our work on NetBind aims to address this gap. We proposed a binding technique that is optimized for network processor-based architectures, minimizing the binding overhead in the critical path, and, allowing network processors to forward minimum size packets at line rates. NetBind aims to balance the flexibility of network programmability against the need for high performance. We think this a unique part of our contribution. The NetBind source code, described and evaluated in this paper, is freely available on the Web (comet.columbia.edu/genesis/netbind) for experimentation. This research is supported by grants from the NSF CAREER Award ANI-9876299, and the Intel Research Council on “Signaling Engines for Programmable IXA Networks”.

6. References

- [1] Network Processing Forum, <http://www.npforum.org>
- [2] Intel IXP1200, <http://www.intel.com/IXA>
- [3] IBM Corporation, IBM PowerNP NP4GS3 Network Processor Datasheet, May 2001.
- [4] Intel Corporation, IXP1200 Network Processor Datasheet, Dec 2000.
- [5] Postel J., Editor, “Internet Protocol”, *Request For Comments 791*, September 1981.

- [6] Campbell, A. T., Gormez, J., Kim, S., Valko, A., Wan, C., Turanyi, Z., "Design Implementation, and Evaluation of Cellular IP", *IEEE Personal Communications*, vol. 7 No. 4, pg. 42-49, Aug 2000
- [7] Intel Corporation, Intel IXA SDK ACE Programming Framework Developer's Guide, June 2001
- [8] Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M., "The Click Modular Router", *ACM Transactions on Computer Systems* 18(3), Aug 2000, pg 263-297.
- [9] Wolf T., Turner, J., "Design Issues for High- Performance Active Routers", *IEEE Journal on Selected Areas in Communications*, March 2001.
- [10] Taylor, D., Turner, J., Lockwood, J., "Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers", *IEEE Open Architectures and Network Programming*, April 2001.
- [11] Spalink, T., Karlin, S., Peterson, L., "Evaluating Network Processors in IP Forwarding", Technical Report TR-626-00, Nov 15, 2000
- [12] Spalink, T., Karlin, S., Peterson, L., Gottlieb, Y., "Building a Robust Software-Based Router Using Network Processors", In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pg. 216-229, Oct 2001
- [13] Karlin, S., Peterson, L., "VERA: An Extensive Router Architecture", In Proceeding of the 4th International Conference on Open Architectures and Network Programming, pg 3-14, April 2001
- [14] Kounavis, M. E., Campbell A. T., Chou, S., Modoux F., Vicente, J., and Zhang H., "The Genesis Kernel: A Programming System for Spawning Network Architectures", *IEEE Journal on Selected Areas in Communication*, Vol. 19, No 3, pg. 511-526, March 2001.
- [15] The NetBind Project home page, available at: <http://www.comet.columbia.edu/genesis/netbind>
- [16] The Genesis Project home page, available at: <http://www.comet.columbia.edu/genesis>
- [17] Intel Corporation, Intel IXA SDK ACE Programming Framework Reference, June 2001.
- [18] Intel Corporation, IXP1200 Network Processor Development Tools User's Guide, Dec 2000.
- [19] Montz, A., Mosberger, D., O'Malley, S., Peterson, L., and Proebsting, T., "Scout, A Communication Oriented Operating System", *Operating System Design and Implementation*, 1994.
- [20] Pu, C., Massalin, H., Ioannidis, J., "The Synthesis Kernel", Springer Verlag, 1988.
- [21] Campbell, A.T., Kounavis, M.E., Vicente, J., Vilella, Miki, K. and De Meer, H., G., "A Survey of Programmable Networks", *ACM SIGCOMM Computer Communication Review*, Vol. 29, No. 2, pp. 7-24, April 1999.
- [22] Keshav S. "An Engineering Approach to Computer Networking", *Addison Wesley*, 1997
- [23] Bennett J. C. R., and Zhang H., "Hierarchical Packet Fair Queueing Algorithms", *IEEE/ACM Transactions on Networking*, 5(5):675-689, Oct 1997.
- [24] Stoica I., Zhang H., and Ng T. S. E., "A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service, in *Proceedings of SIGCOMM'97*, Cannes, France, 1997, pp. 249-262.
- [25] Pappu P. and Wolf. T. "Scheduling processing resources in programmable routers", *Proceedings of the Twenty-First IEEE Conference on Computer Communications (INFOCOM)*, New York, NY, June 2002.
- [26] Goyal, P., Vin, H. M., and Cheng H., "Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", *IEEE/ACM Transactions on Networking*, Vol. 5, No. 5, pp. 690-704, October 1997.
- [27] Johnson E. J., and Kunze A. R., "IXP1200 Programming", Chapter 13, *Intel Press*, 2002.