SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; (in press)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.662

Programming the data path in network processor-based routers

Michael E. Kounavis[1,*,†], Andrew T. Campbell[2], Stephen T. Chou[3] and John Vicente[4]

[1]*Intel Research and Development, Intel Corporation, Hillsboro, OR 97124, U.S.A.*
[2]*Electrical Engineering Department, Columbia University, 500 West 120th Street, 1312 S. W. Mudd Building, New York, NY 10025, U.S.A.*
[3]*Computer Science Department, Columbia University, 1214 Amsterdam Avenue, New York, NY 10027, U.S.A.*
[4]*Intel Corporation, Folsom, CA, U.S.A.*

SUMMARY

There is growing interest in network processor technologies capable of processing packets at line rates. Network processors are likely to be an integral part of next generation high-speed router and switch architectures, replacing the application-specific integrated circuits (ASICs) that are used in routers today. In this paper, we present the design, implementation and evaluation of NetBind, a high-performance, flexible and scalable binding tool for dynamically constructing data paths in network processor-based routers. The methodology that underpins NetBind balances the flexibility of network programmability against the need to process and forward packets at line speeds. To support the dynamic binding of components with the 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 with packet forwarding times. Data paths constructed using NetBind seamlessly share the resources of the same network processor. Resources are assigned during the binding process. We compare the performance of NetBind to the MicroACE system developed by Intel and show that NetBind provides better performance in comparison to MicroACE with smaller binding overhead. The NetBind source code described and evaluated in this paper is freely available on the Web (http://www.comet.columbia.edu/genesis/netbind) for experimentation. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: network processors; programmable networks; service creation; binding; code morphing

*Correspondence to: Michael E. Kounavis, Intel Research and Development, Intel Corporation, Hillsboro, OR 97124, U.S.A.
†E-mail: michael.e.kounavis@intel.com

Contract/grant sponsor: NSF CAREER Award; contract/grant number: ANI-9876299
Contract/grant sponsor: Intel Research Council

Received 30 October 2002
Revised 21 October 2003
Accepted 21 October 2003

INTRODUCTION

There has been a growing interest in network processor technologies [1–3] that can support software-based implementations of the critical path while processing packets at high speeds. Network processors employ multiple processing units to offer high packet-processing throughput. We believe that introducing programmability into 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 are 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), introduce some overhead when applied to network processors in terms of additional instructions in the critical path. This overhead can be significant in some network processors. 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. For example, code optimization algorithms may need to process each instruction in the data path code several times resulting in $O(n)$ or higher complexity as a function of the number of instructions in the critical path. Such algorithms may not be suitable for applications that require fast data path composition (e.g. data path composition on a packet-by-packet basis). 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* [4], a high-performance, flexible and scalable binding tool for creating modular 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 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 [5] 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 with packet forwarding times. In NetBind, data path components export symbols, which are used during the binding process. Each symbol is associated with some specific instruction executed in the critical path (e.g. a branch instruction or an arithmetic logic unit instruction that uses a critical register). Not all

instructions in the critical path are exported as symbols, however. The number of symbols exported by data path components, h, is typically much smaller than the total number of instructions executed in the critical path, n. The NetBind binding algorithm does not inspect every instruction in the data path code, only the symbols exported by data path components. Because of the fact that h is much smaller than n, the time it takes to inspect every exported symbol is typically much smaller than the time it takes to inspect every instruction in the data path code. For this reason, the NetBind binding algorithm can compose packet processing pipelines very quickly, in the order of microseconds. The NetBind binding algorithm is associated with $O(h)$ complexity. 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 an area for future work.

DYNAMIC BINDING IN NETWORK PROCESSORS

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 (FPGAs), 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 IXP network processors [2] or the dyadic protocol processing units of IBM's PowerNP processors [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 [3]). Network processors typically consume less power than FPGAs and are more programmable than ASICs.

Our study on the construction of modular data paths is focused on Intel's IXP1200 network processor. The IXP1200 network processor incorporates seven RISC CPUs, a proprietary bus (IX bus) controller, a PCI controller, control units for accessing off-chip SRAM and SDRAM memory chips, and an on-chip scratch memory. The internal architecture of the IXP1200 is illustrated in Figure 1. In what follows, we provide an overview of the IXP1200 architecture. The information presented here about the IXP1200 network processor is sufficient to understand the design and implementation of NetBind. For further details about the IXP1200 network processor, see [7]. One of the IXP1200 RISC CPUs is a StrongARM Core processor running at 200 MHz. The StrongARM Core can be used for processing slow path exception packets, managing routing tables and other network state information. The other six RISC CPUs, called 'microengines' are used for executing the fast path code. Like the StrongARM Core, microengines run at 200 MHz. Each microengine supports four hardware contexts sharing an arithmetic logic unit, register space and instruction store. Each microengine has a separate instruction store of size 1 K instructions (i.e. 4 K bytes) called the 'microstore'. Unlike the StrongARM

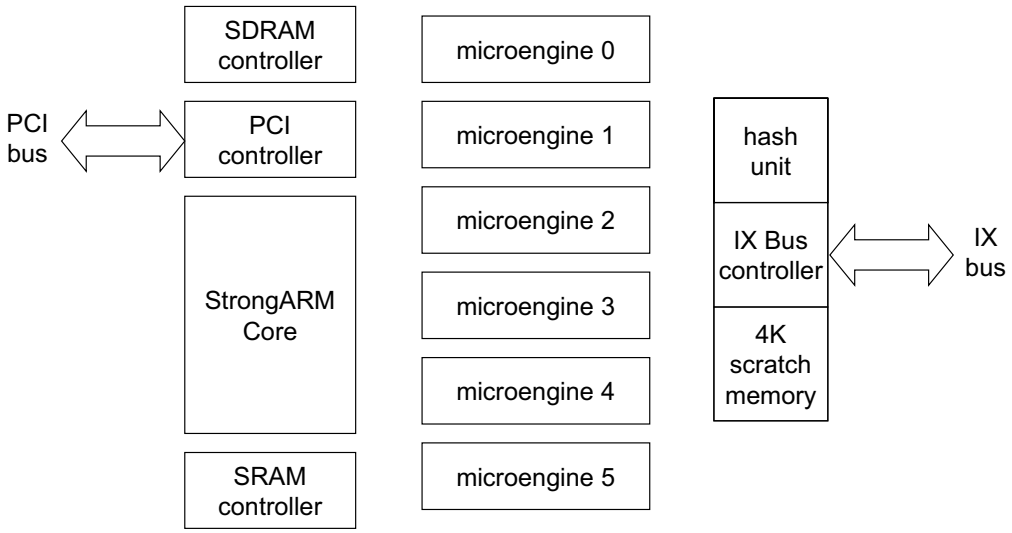

Figure 1. Internal architecture of the IXP1200 network processor.

Core processor, microengines do not use instruction or data caches because of their associated performance implications. For example, using a data cache in the critical path is effective only when it is easy to determine whether some portion of the data path structures (e.g. classification, forwarding or scheduling data structures) is used more often that others. Each microengine incorporates 256 32-bit registers. Among these registers, 128 registers are general purpose registers (GPRs) and 128 are memory transfer registers. The register space of each microengine is shown in Figure 2. Registers are used in the following manner. GPRs are divided into two banks (i.e. banks A and B, shown in Figure 2), of 64 registers each. Each GPR can be addressed in a 'context relative' or 'absolute' addressing mode. By context relative mode, we mean that the address of a register is meaningful to a particular context only. Each context can access one quarter of the GPR space in the context relative addressing mode. By absolute mode, we mean that the address of a register is meaningful to all contexts. All GPRs can be accessed by all contexts in the absolute addressing mode.

The memory transfer registers are divided between SRAM and SDRAM transfer registers. SRAM and SDRAM transfer registers are further divided among 'read' and 'write' transfer registers. Memory transfer registers can also be addressed in context-relative and absolute addressing modes. In the context relative addressing mode, eight registers of each type are accessible on a per-context basis. In the absolute addressing mode, all 32 registers of each type are accessible by all contexts.

The IXP1200 uses a proprietary bus interface called the 'IX bus' interface to connect to other networking devices. The IX bus is 64-bit wide and operates at 66 MHz. In the evaluation boards we use for experimenting with NetBind, the IXP1200 is connected to four fast Ethernet (100 Mbps) ports. The IX bus controller (shown in Figure 1) incorporates two FIFOs for storing minimum size packets,

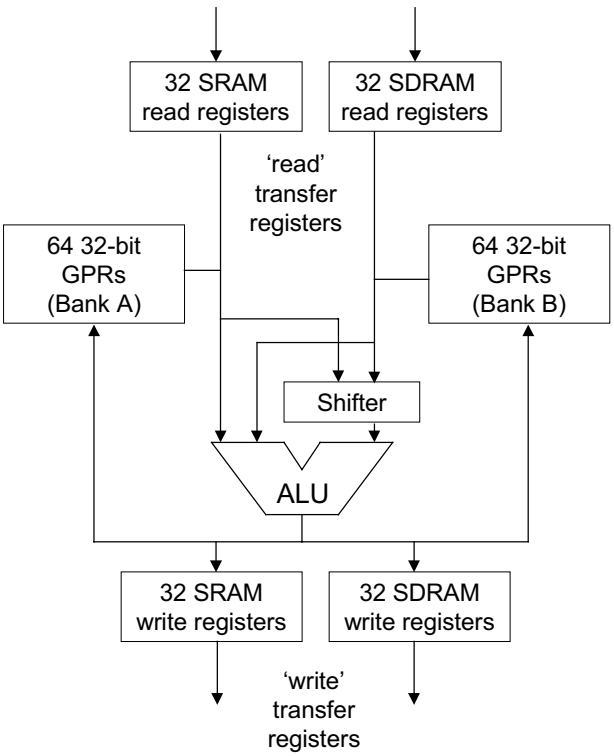

Figure 2. Microengine registers of IXP1200.

a hash unit and a 4K 'scratch' memory unit. The scratch memory is used for writing or reading short control messages, which are exchanged between the microengines and the StrongARM Core.

Dynamic binding issues

There are many different techniques for introducing new services into software-based routers [5,8–13]. At one end of the spectrum, the code that implements a new service can be written in a high-level, platform-independent programming language (e.g. Java) and be compiled at run time producing optimized code for some specific network hardware. In contrast, data paths can be composed from packet processing components. Components can be developed independently from each other, creating associations at run time. This dynamic binding approach reduces the time for developing and installing new services, although it requires that algorithmic components are developed and tested in advance. In what follows, we discuss issues associated with the design of a dynamic binding system for network processor-based routers.

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;
- processor handoffs;
- instruction store limitations; and
- complexity of the binding algorithm.

Headroom limitations

Line rate forwarding of minimum size packets (64 bytes) is an important design requirement for routers. Routers that can forward minimum size packets at line rates are typically more robust against denial-of-service attacks, for example. Line rate forwarding does not mean zero queuing. Rather, it means that the output links of routers can be fully utilized when routers forward minimum size packets.

A necessary condition for achieving line rate forwarding is that the amount of time dedicated to the processing of a minimum size packet does not exceed the packet's transmission or reception times, assuming a single queuing stage. If multiple sequential queuing stages exist in the data path, then the processing time associated with each stage should not exceed the packet's transmission or reception times.

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 number of instructions that can be executed in the critical path is typically small, ranging between some tens to hundreds of instructions. The number of instructions that can be executed in the critical path without violating the line rate forwarding requirement is often called *headroom*.

Headroom is a precious resource in programmable routers. Traditional binding techniques used by operating systems or high-level programming languages are not suitable for programming the data path in network processor-based routers because these techniques waste too much headroom and, thereby, limit the performance of the router. This is because such binding techniques burden the critical path with unnecessary code stubs, or with time-consuming memory read/write operations for accessing function tables. 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.

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.

The number of parameters which are used by a component determines the number of registers a component needs to access. If this number is smaller than the number of registers allocated to a component, the entire parameter set can be stored in registers for fast access. The placement of some component state in memory impacts the data path's performance. Although modern network processors support a large number of registers, register sets are still small in comparison with the amount of data that needs to be managed by components. Transfer through memory is necessary when components are executed by a separate set of hardware contexts and processing units. A typical case is when queuing components store packets into memory, and a scheduler accesses the queues to select the next packet for transmission into the network.

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 interface 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 a code optimization algorithm may be time consuming, and may not be suitable for applications requiring fast data path composition.

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 directs the program flow from one component to another based on some global binding state and on the parameters returned by each module. This method is costly. The minimum amount of state that needs to be checked before the execution of a new component is an identifier to the next module and a packet buffer handle exchanged between components. Checking this amount of state requires at least four compute cycles. If a data path is split between six components, then the total amount of overhead introduced in the critical path is 24 compute cycles, which is a significant part of the network processor headroom in many different network processors. For example, in IXP network processors that target the OC-192 line rate [14,15], the headroom can be as small as 57 compute cycles. In this case the binding overhead accounts for 42% of the headroom. The dispatch loop approach 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* into 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. When a new component is added, only the content of the vector table needs to be modified. Although this approach is more suitable for dynamic binding, it involves at least one additional memory read operation for each component in the critical path. In commercial network processors that target OC-48 and OC-192 speeds [14,15] memory access latencies can be several times larger than the network processor headroom (e.g. 100–400 compute cycles). As a result it is difficult to apply the vector table technique in order to support modular data paths at such speeds.

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. The third binding method introduces less overhead in terms of additional processing latency in the critical path. For these reasons, we have used the third binding method in NetBind. In this paper we refer to this method as 'code morphing' method because it modifies the machine language code of components at run time.

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.

One way to support isolation between data paths is to assign each data path to a separate processing unit, or a set of hardware contexts, and to make sure that each data path does not execute code that exceeds the network processor headroom. Determining the execution time of programs given some specific input is typically an intractable problem. However, it has been shown that packet processing components can have predictable execution times [16]. One solution to the problem is to allow code modules to carry their developer's estimation of the worst-case execution time in their file headers. The time reported in each file's header should be trusted, and code modules should be authenticated using well-known cryptographic techniques. To determine the worst-case execution time for components, developers can use upper bounds for the time it takes to complete packet processing operations.

Bandwidth can be partitioned using packet scheduling techniques. Packet scheduling techniques (e.g. deficit round robin [17], weighted fair queuing [18], or start time fair queuing [19]), can be implemented either in the network processor hardware or as part of a packet-processing pipeline. Hierarchical packet scheduling algorithms [20] can be used for dividing bandwidth between a hierarchy of coexisting data paths. The implementation of packet scheduling algorithms in network processors is beyond the scope of this paper.

Processor handoffs

Sometimes the footprints of data paths can be so large that they cannot be placed in the same instruction store. In this case, the execution of the data path code has to be split across two or more processing units. Another case arises when multiple data paths are supported in the same processor. In this case, the available instruction store space of processing units may be limited. As a result, the components of a new data path may need to be distributed across multiple instruction stores. Third, the execution of a data path may be split across multiple processing units when 'software pipelining' [21] is employed for increasing the packet rate that can be achieved in a network processor architecture. Software pipelining is a technique that divides the functionality of a data path into several stages. Pipeline stages can potentially run in different processing units. Multiple stages can be executed at the same time forwarding the packets of different data flows.

We call the transfer of execution from one processing unit to another (that takes place when a packet is being processed), 'processor handoff'. Processor handoffs impact the performance of data paths and need to be taken into account by the binding system. The performance of processor handoffs depends on the type of memory used for communication between processing units. A dynamic binding system should try to minimize the probability of having high-latency processor handoffs in the critical path. This is not an easy task and requires a search to be made on all possible ways to place the code of data paths into the instruction stores of processing units.

Instruction store limitations

Instruction store limitations represent a constraint on the number of data paths or processing functions that can be simultaneously executed in the same network processor. A solution to this problem would be to have the binding system fetch code from off-chip memory units into instruction stores on an on-demand basis. This solution, however, can significantly impact the performance of the critical path because of the overhead associated with accessing memory units.

Complexity of the binding algorithm

The last consideration for designing a dynamic binding system is the complexity of the binding algorithm. In many cases, the complexity of a binding algorithm affects the time scales over which the binding algorithm can be applied. A complex binding algorithm needs time to execute, and is typically not suitable for applications that require fast data path composition. Keeping the binding algorithm simple while producing high-performance data paths is an important design requirement for a good binding system. Applications that require real-time binding include classification, forwarding and traffic management. The performance of such data path algorithms depends on the properties of classification databases [22,23], routing tables and packet scheduling configurations [24]. These properties typically change at run time calling for advanced service creation environments for programming the data path efficiently. A dynamic binding system should ideally support a wide range of packet-processing applications ranging from the creation of virtual networks over slow time scales to the fast creation of customized data paths, after disasters occur. Disasters typically result in rapid changes on the input traffic characteristics and topologies of communication networks. To accommodate increased traffic demands or to reroute traffic to alternate links once some part of

the communication infrastructure is physically damaged, communication networks need to be highly adaptive, calling for new techniques and software methodologies for rapid service creation.

NETBIND DESIGN

NetBind is a binding tool we have developed that offers dynamic binding support for the IXP1200 network processor. NetBind consists of a set of libraries which can modify IXP1200 instructions, create processing pipelines or perform higher-level operations such as data path admission control. Components are written in machine language code called *microcode*. NetBind groups components into processing pipelines that execute on the microengines of IXP1200. In what follows, we present the design and implementation of NetBind.

Design principles

The most fundamental principle in the design of NetBind is that binding is performed by avoiding the use of code stubs or indirection due to their associated performance penalty. As a consequence, NetBind modifies the machine language code of components at run time in order to construct consistent packet processing pipelines. Since the minimum step required for advancing the program sequence from one component to another is a branch operation, NetBind modifies the destination addresses of branch instructions connecting components. In this way, NetBind allows the program sequence to continue to the next component in a pipeline, after the previous component's code is executed.

NetBind uses no global binding state. Because of this, NetBind requires that components expose registers used for inter-component communication as a programming interface. Registers are exported as binding symbols. NetBind splits the register space into regions and imposes a consensus on how registers are used. Parameters exchanged between components do not need to be stored as global binding state. In addition, no global biding state needs to be checked every time a new component's code executes. This reduces the overhead introduced in the critical path significantly, as discussed in the evaluation section. Registers are used in three different ways in the components we experimented with. First, components use registers to hold input argument values, as explained in detail below. Second, components use registers to exchange information produced and consumed during the execution of a packet processing pipeline. Third, components use registers to share information between each other. We distinguish registers between 'input argument', 'pipeline' and 'global variable' registers depending on the way registers are used. By exporting registers as input argument, pipeline and global variable, components can communicate with each other with the least possible communication cost.

A second principle followed in the design of NetBind is that binding is performed by inspecting a much smaller number of instructions than the body of code executed in the critical path. This second principle results in fast data path composition. To reduce the number of instructions inspected during binding, NetBind requires components to allocate register addresses statically. Statically allocated register addresses are exported as part of each component's programming interface. The only exception to this rule concerns the allocation of global variable register addresses. Global variables represent a shared resource because they are accessed by multiple simultaneously running hardware threads and referenced using the absolute addressing mode. As a result, global variables addresses need to be allocated dynamically on an on-demand basis to packet-processing pipelines.

Data path specification hierarchy

Before creating data paths, NetBind captures their structure and building blocks in a set of executable profiling scripts. NetBind uses multiple specification levels to capture the building blocks of packet forwarding services and their interaction. NetBind uses multiple specification levels in order to offer the programmer the flexibility to select the amount of information that can be present in data path profiling scripts. Some profiling scripts are generic, and thus applicable to any hardware architecture. Some other profiling scripts can be specific to network processor architectures, potentially describing timing and concurrency information associated with components. A third group of scripts can be specific to a particular chip such as the IXP1200 network processor.

Figure 3 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 is generic and can be applied to many different types of programmable routers (e.g. PC-based programmable routers [8], software [9] or hardware [10] plugin-based routers or network processor-based routers [11,12]).The purpose of the virtual router specification is to capture the composition of a router in terms of its constituent building blocks and their interaction, without specifying the method that is used for component binding. Programmable routers can be constructed using a variety of programming techniques ranging from higher-level programming languages (e.g. Java) to hardware plugins based on FPGA technologies. The virtual router specification can be used for service creation in a heterogeneous infrastructure of programmable routers of many different types. 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 (i.e. hardware threads and processing units).

Components are grouped into processing *stages*. A processing stage is a part of a software pipeline and consists of a set of components that are executed by one or multiple 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 are grouped into the same stage of a pipeline. A data path software pipeline is split between at least two stages 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.

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

The network processor specification is dynamically created from the virtual router specification. The virtual router specification does not include information about symbol bindings. Because of

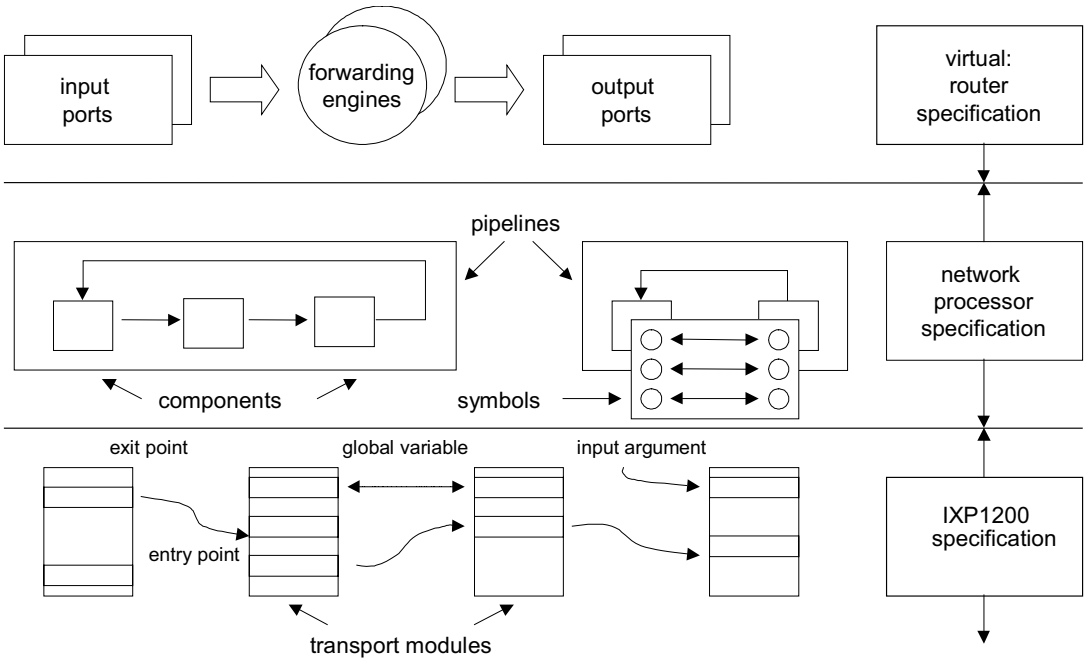

Figure 3. Data path specification hierarchy.

this reason, the virtual router specification is not sufficient to describe the interaction between components at the network specification level. To convert the virtual router specification into the network processor specification, NetBind queries a database that stores information about symbol bindings. Groups of symbol bindings are being stored for each component binding used in the data path. The NetBind binding approach assumes that the number of components and symbols used for constructing data path algorithms is limited (i.e. in the same order of magnitude as the classes and methods found in a higher-level programming language API). If this assumption is true, it may not be difficult for component developers to specify exactly which symbol bindings should characterize the interaction between components in the data path.

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

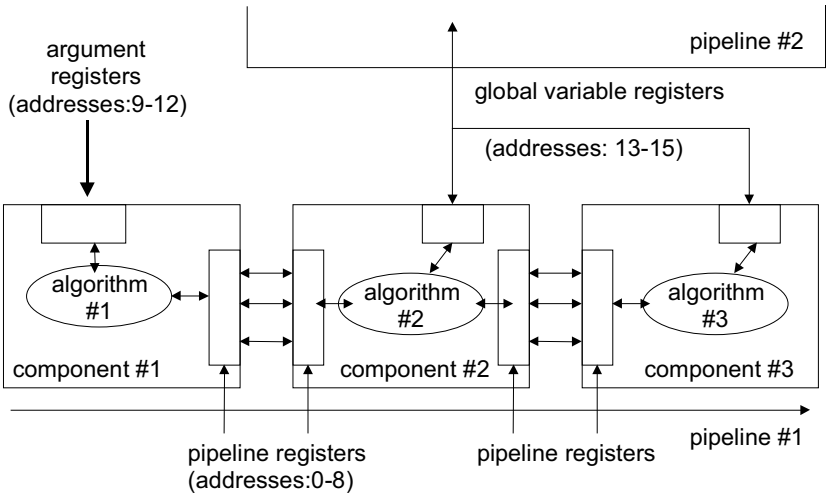

Figure 4. Register allocations in NetBind.

where the program flow jumps. Input arguments are instruction store addresses of 'immed' IXP assembler instructions [25] 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. Network processor related specifications are automatically translated into IXP12000 related specifications by the NetBind binding system.

Register allocations

Register allocations realized in NetBind are shown in Figure 4. We observe that in data path implementations we have experimented with, registers are used in three 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 values 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 of the 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 [5]). 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 in a context relative addressing manner. Examples of input arguments include the SRAM address where a linked list of packet buffer descriptors is stored, the SDRAM address where a routing table is located or the scratchpad address where a small forwarding MIB is maintained.

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 the absolute addressing mode.

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 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. Otherwise, the NetBind binding algorithm would have to inspect all instructions in the data path code, locate the instructions where registers are used and assign register addresses dynamically.

In the current implementation of NetBind, we use 18 GPRs per context as pipeline registers (addresses 0–8 of banks A and B) and eight GPRs per context as input argument registers (addresses 9–12 of banks A and B). Memory transfer registers are all pipeline registers. Each context contributes six GPRs in order to be used as global variable registers. In this manner, a pool of 24 global variable registers are shared among pipelines or data paths. Dynamic register allocation is used for global variable registers, because these registers are shared between different pipelines or data paths and it is not possible for their addresses to be known in advance. In our Spawning Networks Testbed, as part of the Genesis Project, we have used NetBind to build IPv4 [26] and Cellular IP [6] virtual routers. Static register allocation is sufficient for programming this set of diverse data paths.

Our characterization of registers as 'input argument', 'pipeline' and 'global variable' reflects register usage in the components we experimented with. Alternative designs may define different types of register usage. What is important in the design of NetBind is that we allow components to communicate without checking the global binding state every time a new component is executed. This is accomplished by allowing components to expose registers as a programming API. Registers can be characterized in many different ways in the API depending on the programmer's needs.

Binding algorithm

The NetBind binding algorithm operates on components and modifies their microcode instructions at run time in order to compose packet-processing pipelines. A description of the binding algorithm is given below.

`NETBIND-BIND` (*component-list*, *binding-list*)
1. **for** i = 1 *to number-of-components* **do**
2. *place each component into the same microstore*

3. **for** i = 1 *to number-of-components* **do**
4. **for** j = 1 *to number-of-input-arguments for the i-th component* **do**
5. *j-th-input-argument* ← *exported-symbol-value for this argument*
6. **for** j = 1 *to number-of-global-variables for the i-th component* **do**
7. *j-th-global variable* ← *exported-register-address for this variable*
8. **for** i = 1 *to number-of-bindings* **do**
9. *exit-point for the i-th binding* ← *exported-entry point for this exit point*

Lines 1 and 2 describe the placement of components into a microstore. Lines 3–7 describe the assignment of input argument values and global variable register addresses. Lines 8 and 9 describe the bindings between exit points and entry points. Assuming that lines 1 and 2 can be executed in bounded time (e.g. by a DMA transfer unit or other hardware acceleration unit) and that the component list passed as input contains m input arguments, n global variables and l bindings, the complexity of the binding algorithm is $O(m+n+l)$. This is because lines 3–5 is executed m times overall, lines 6 and 7 are executed n times, and lines 8 and 9 are executed l times. The total number of symbols that are processed during the binding process is $h = m+n+l$. As a result, the complexity of our binding algorithm is $O(h)$ as stated earlier.

Figure 5 illustrates an example of how NetBind performs dynamic binding. In this example, two components are placed into an instruction store. Figure 5 shows the instruction store containing the components and the instructions of components, which are modified during the binding process. The fields of instructions that 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 5, two pairs of 'immed_w0' and 'immed_w1' instructions are modified at run time, during the steps (1) and (2), as shown in Figure 5. 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 5 load the absolute registers @var1 and @var2. The absolute registers @var1 and @var2 are global variable registers. An admission controller assigns address values for 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 5) 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 5).

If the destination addresses of branch instructions are not adjusted during the placement of transport modules into instruction stores, the binder needs to modify all branch instructions in each transport module. The destination addresses of branch instructions are incremented by the instruction store addresses where transport modules are placed. This step can also be performed during placement of transport modules into instruction stores. 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 5). 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 which we discuss in the evaluation section.

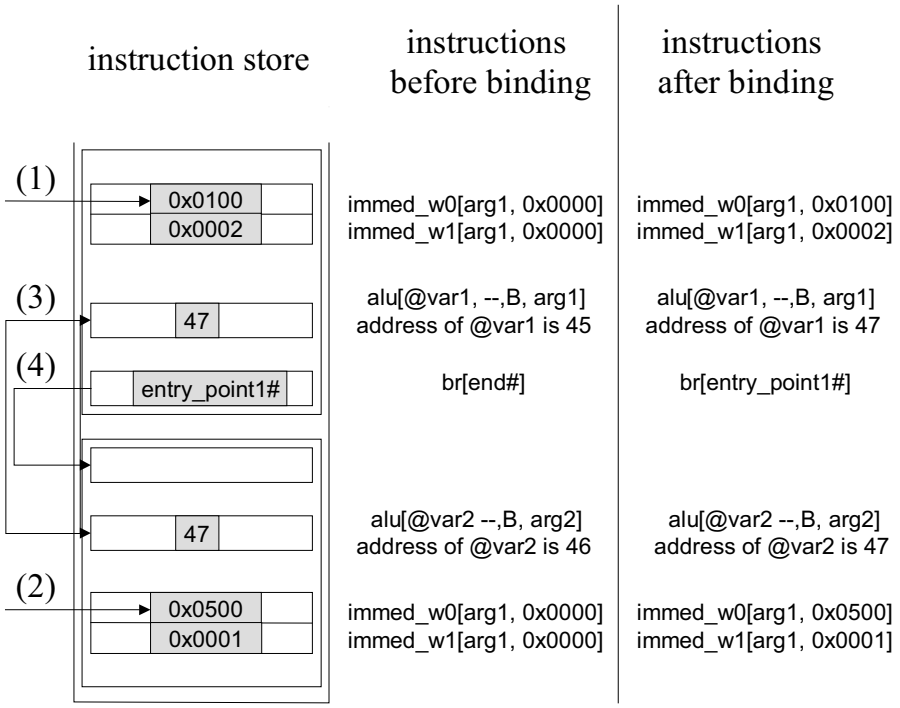

Figure 5. Dynamic binding in NetBind.

The admission control algorithm

In NetBind, admission control drives the assignment of system resources including registers, memory, instruction store space, headroom and hardware contexts. Resource management in network processors is a difficult problem. The difficulty stems from the fact that network processors use multi-processor architectures that expose many different types of resources to applications (e.g. multiple types of memory units, registers, packet-processing units and hardware contexts). We have not fully investigated the resource management problem in network processors as yet. Instead, we designed a simple heuristic algorithm for admission control. NetBind applies a 'best-fit' bin-packing algorithm to determine the most suitable microengines where data path components should be placed. The best-fit bin-packing algorithm determines the microengine where a packet-processing pipeline should be placed by calculating the remaining resources (i.e. leftover) after the placement of the pipeline in each microengine. In order to determine the remaining resources associated with each resource assignment, NetBind calculates a weighted average, taking every type of resource (i.e. memory and instruction store space, hardware contexts and global variable registers) into account. Each type of resource weighs

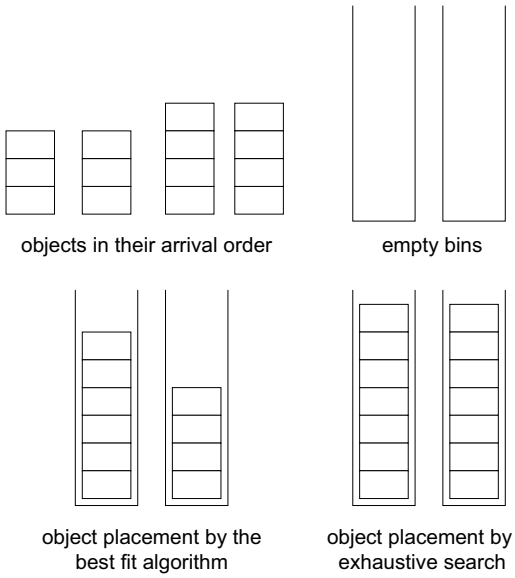

Figure 6. Bin-packing example.

equally on the calculation of the leftover resources. The microengine selected is the one that results in the smallest amount of leftover after the placement of the pipeline.

Maintaining records of resource usage is essential to supporting isolation between data paths. If the 'best-fit' algorithm fails NetBind applies an 'exhaustive search' algorithm to determine the microengines where data path components should be placed. 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 a small number of pipelines and can result in efficient resource allocations when the best-fit algorithm fails.

Real applications do need exhaustive search. Best fit may fail to find the optimal allocation. In what follows we provide an example that illustrates why exhaustive search is necessary. In the example shown in Figure 6, microengines are abstracted and referred to as 'bins' and pipelines are abstracted and referred to as 'objects' for the sake of simplicity. Let us assume that we have two bins of size seven and two objects which need to be placed into the bins of size three each. The best fit algorithm places the two objects in the first of the bins. If an object of size four needs to be added into the bins as well, the best fit algorithm places the object in the second bin. If a fourth object, also of size four needs to be added into the bins, then the best fit algorithm determines that there is not enough space in the system to accommodate the new object. This happens because the leftover in the first bin is one and in the second bin is three. Exhaustive search in this case determines that there exists a placement configuration which can accommodate all four objects. In the optimal configuration each of the bins contains one object of size four and one object of size three.

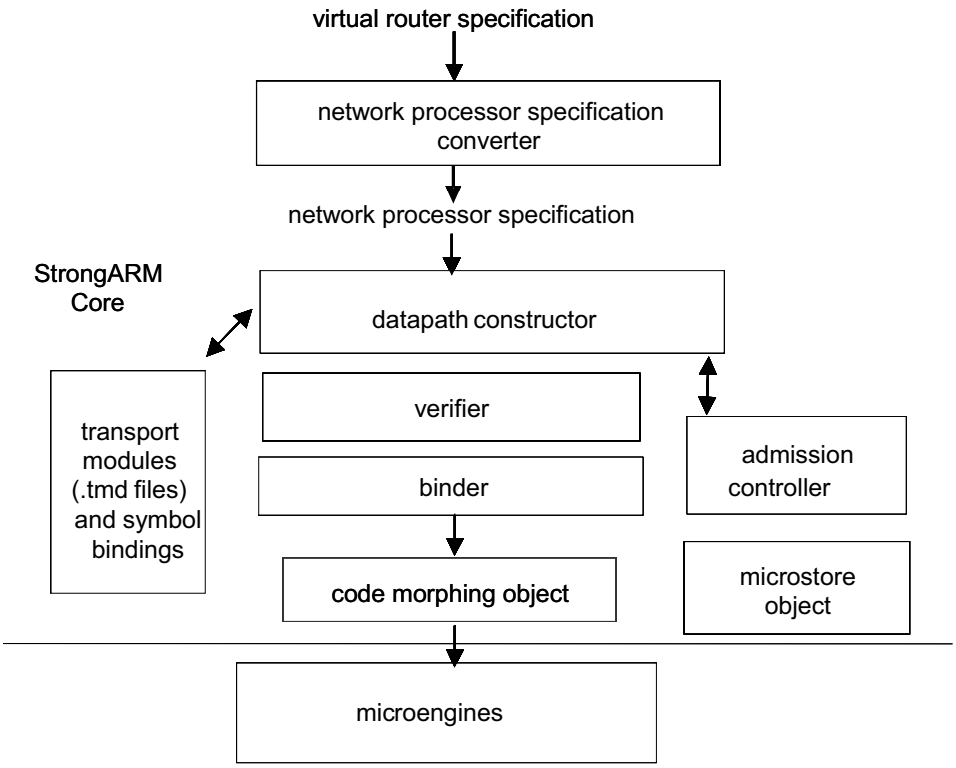

Figure 7. The NetBind binding system.

Before realizing a data path, NetBind examines whether the candidate data path exceeds the network processor headroom. Each transport module carries an estimate of its worst-case execution time. The execution time for each component is determined from deterministic bounds on the time it takes for different microengine instructions to complete. With the exception of memory operations, the execution time of microengine instructions is deterministic and can be obtained using Intel's transactor simulation environment [27].

NETBIND IMPLEMENTATION

We have implemented NetBind in C and C++ as a user space process in the StrongARM Core processor of IXP1200. The StrongARM Core processor runs an embedded ARM version of Linux. A diagram of the components of the NetBind system is shown in Figure 7. The NetBind binding system consists of the following.

- *A network processor specification converter object*, which converts a virtual router specification to a network processor specification. Information about symbol bindings required for the conversion is obtained from a database of transport modules and symbol bindings.
- *A data path constructor object*, which coordinates the binding process, accepting as input the network processor specification of a data path. The data path constructor parses the transport module (.tmd) files that contain data path components and converts the network processor-related specification of a data path into an IXP1200-related specification.
- *An admission controller object*, which determines whether the resource requirements of a data path can be met. The admission controller performs resource allocation for every candidate data path. If the resource requirements of a data path can be met, the admission controller assigns a set of microengines, hardware contexts, global variable registers, memory and instruction store regions to the new data path. Once admission control takes place, the data path is created.
- *A verifier object*, which verifies the addresses and values of each symbol associated with a data path before binding takes place. The verifier object also checks the validity of the resource allocation made by the admission controller. The verifier object is useful for debugging since it makes sure that no incorrect or malicious microcode is written into the instruction stores of IXP units. Using a verifier object one can avoid resetting the system manually every time the data path constructor makes a mistake.
- *A binder object*, which performs low-level binding functions, such as, the modification of microwords for binding or the loading of transport modules into instruction stores. The binder is 'plug-and-play' and can either create new data paths or modify existing ones at run time.
- *A code morphing object*, which offers a set of methods that parse IXP1200 microinstructions and modify the fields of these instructions. The code morphing object is used by the binder.
- *A database of transport modules and symbol bindings*. Transport modules encapsulate component code and are accessed by the data path constructor. Symbol bindings are used during the conversion of the virtual router specification into the network processor specification.
- *A microstore object*, which can initialize or clear the instruction stores on an IXP1200 network processor. The microstore object can also read from, or write into, any address of the instruction stores.

In order to implement the binder object we had to discover the binary representations for many IXP1200 microassembler instructions. This was not an easy task since the opcodes of microassembler instructions do not have fixed lengths and they are not placed in fixed locations inside each instruction's bit set.

Transport modules can be developed using tools such as the Intel Developer Workbench [27]. 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 header includes information about the number of microcode pages associated with a Workbench project, the manner in which the instruction stores are filled and the size of pages associated with a project's microcode. Since we have had no access to the source code of the standard development tools provided by Intel [27] (i.e. the Intel Developer Workbench, the transactor simulation environment and the microassembler), we have created our own microassembler extensions on top of these tools. Our microassembler extensions have been used for creating the transport modules of components.

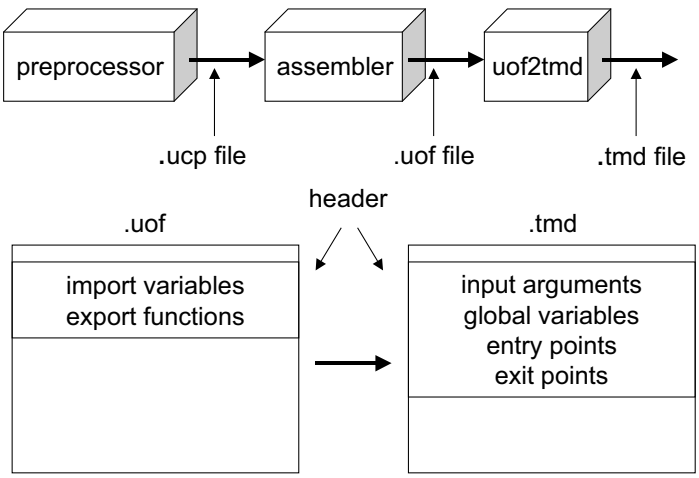

Figure 8. Microassembler extensions.

The UOF file format is suitable for encapsulating statically compiled microcode. The UOF file format 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. Our microassembler extensions are shown in Figure 8. A utility called *uof2tmd* takes a .uof file as input and produces a '.tmd' file as output. The 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. Currently, the information included in the TMD header is obtained from the UOF header.

SERVICE CREATION USING NETBIND

Programming virtual routers

We have used NetBind to create programmable virtual routers that seamlessly share the resources of the communications infrastructure. A software model reflecting the structure and building blocks of a programmable virtual router are illustrated in Figure 9. A 'routelet' operates as an autonomous virtual router forwarding packets at layer three from a set of input ports to a set of output ports. Multiple routelets can share the resources of a network processor. The dynamic instantiation of a set of routelets across the network hardware, and the formation of virtual networks on-demand are broader research goals addressed in the Genesis Project [28] and not dealt with in this paper. Each routelet has its own data path, state and control plane, which are fully programmable. The data path can run in the microengines of a network processor, such as the IXP1200. The data path consists of input ports, output

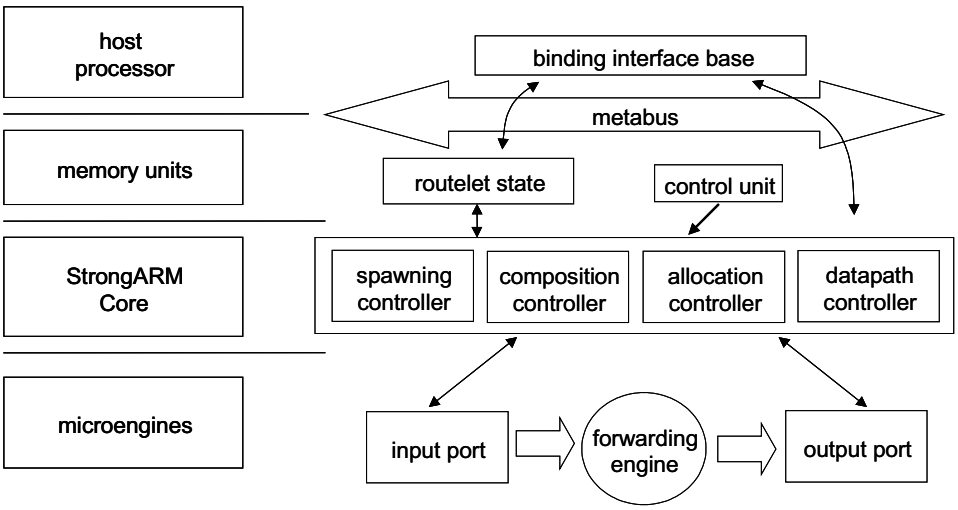

Figure 9. Routelet.

ports and forwarding engines, which comprise atomic transport modules. Transport modules perform simple operations on packets such as decrementing the Time to Live (TTL) field of an IP header, or more complex operations such as packet classification and scheduling. The control plane can run on network processors or host processors as well. For example, in the routelet architecture, as shown in Figure 9, a *control unit* runs on the StrongARM core of the IXP1200, whereas additional control plane objects run in host processors. The control unit provides low-level operating system support for creating routelets, modifying data paths at run time or managing a routelet's state. The control unit has been implemented using the NetBind libraries. Each routelet maintains its own state, which is distributed over the memory units of IXP1200.

A metabus supports a hierarchy of distributed objects that realize a number of virtual network specific communication algorithms including routing, signaling, quality of service control and management. At the lowest level of this hierarchy binding interface base objects provide a set of handlers to a routelet's controllers and resources allowing for the programmability of a range of internetworking architectures using the programming environment. The binding interface base separates the implementation of the finite state machine, which characterizes communication algorithms (e.g. the RIP finite state machine, the RSVP finite state machine, etc.) from the implementation of the mechanisms that transmit signaling messages inside the network. Communication algorithms can be implemented as interactions of distributed objects, independent of the network transport mechanisms. Distributed objects that comprise network architectures (e.g. routing daemons, bandwidth brokers, etc.) are not aware of the existence of routelets. Distributed objects give the 'illusion' of calling methods on local objects whereas in practice call arguments are

'packaged' and transmitted over the network via one or more routelets. This abstraction is provided by the metabus.

We have chosen to realize the metabus abstraction as an orblet, a virtual object request broker (ORB) derived from the CORBA [29] object-programming environment. Current ORB implementations are tailored toward a single monolithic transport service. This limitation makes existing CORBA implementations unsuitable for programming virtual network architectures that may use different transport environments. To resolve this issue we have implemented the 'acceptor–connector' software pattern [30] in the orblet. The acceptor–connector pattern wraps low-level connection management tasks (e.g. managing a TCP connection) with a generic software API. The orblet can use a range of transport services on-demand in this case. To use a specific transport service, the orblet dynamically binds to an inter-ORB protocol engine supported by the programming environment. We have created an IIOP protocol engine for interacting with IP-based routelets. Isolation between the programming environments of virtual networks is realized in two ways: First, each metabus uses a separate transport environment for object interaction where the transport environment is dependent on the spawned network architecture. Second, each metabus offers dedicated naming services to the spawned network architecture.

We believe that virtual networks can be programmed, deployed and provisioned on-demand using appropriate distributed operating system support and network processor hardware. While virtual networks share resources, they do not necessarily use the same software for controlling those resources. Programmable virtual networks can be created by selecting customized protocols for transport, signaling, control, and management and can be used for investigating the cost and performance of network architectures. Our work on programmable virtual networking assumes a network-wide service creation infrastructure of network processors. This assumption may not be true in the near future. It is difficult for network processors today to match the very high speeds (e.g. some terabit/s) at which ASICs operate in the network core. While it may take some time until silicon is developed for the network core, network processors can effectively run at the network edge. Most of the sophistication in packet processing required at the network edge can be programmed using existing network processor hardware. Therefore, virtual routers can effectively run at the network edge using commercial network processors.

IPv4 data path

The data path associated with an IPv4 virtual router implemented using NetBind is illustrated in Figure 10. This data path comprises seven transport modules. In the figure, the first six modules run on microengine 0 and are executed by all four contexts by the microengine:

- a virtual network demultiplexor (`receiver.tmd`) module receives a packet from the network performs virtual network classification and places the packet into an SDRAM buffer;
- an IPv4 verifier (`ipv4_verifier.tmd`) module verifies the length, version and TTL fields of an IPv4 header;
- a checksum verifier (`ipv4_checksum_verifier.tmd`) module verifies the checksum field of an IPv4 header;
- an IPv4 header modifier (`ipv4_hdr_modifier.tmd`) module decrements the TTL field of an IPv4 packet header and adjusts the checksum accordingly;

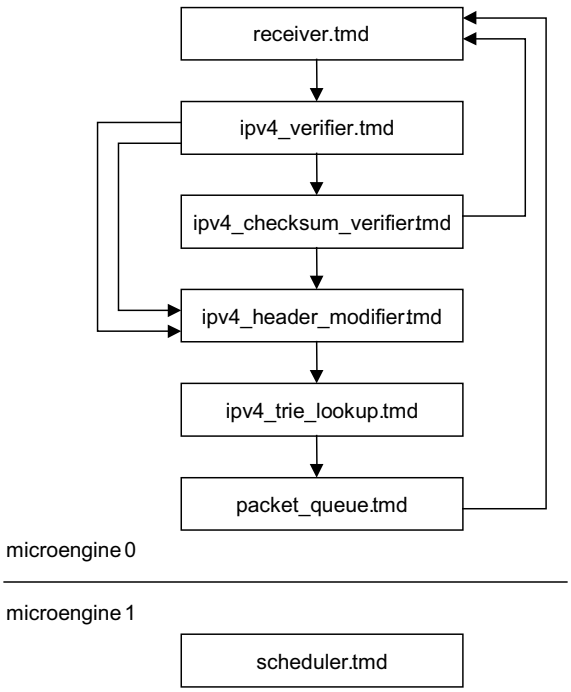

Figure 10. An Ipv4 data path.

- an IPv4 trie lookup (`ipv4_trie_lookup.tmd`) module performs an IPv4 route lookup; and
- a packet queue (`packet_queue.tmd`) module dequeues a packet.

In addition, a seventh module runs on a separate microengine (microengine 1). This module is a dynamic scheduler that assigns the transmission of enqueued packets to hardware contexts. Context 0 executes the scheduler, whereas contexts 1, 2 and 3, transmit packets to the network.

The network processor specification for the router presented above is shown in Figure 11. The specification is written in a scripting language called *ni*. This scripting language has been defined and developed as part of the NetBind testing environment. The syntax of ni commands is similar to the syntax of IXP1200 microcode instructions. The first string in a ni command identifies the operation that is executed. A number of operands follow. Operands are declared inside brackets and are separated by commas. The final part of a command consists of one or multiple optional tokens. For a detailed description of the commands and syntax of the ni scripting language see [31].

In the first part of the network processor specification input arguments are declared. Input arguments are declared as symbols. Each symbol is associated with a string name. Optional tokens follow the declaration of input arguments. These tokens specify numerical values assigned to input arguments.


```
//declaration of input arguments
decl[symbol, inp_packet_buff_base], 0x20100
decl[symbol, inp_scratchpad_mib], 0x0c0
decl[symbol, inp_scratchpad_pwp], 0x200
decl[symbol, inp_route_base_64k], 0x20000
decl[symbol, inp_route_base_256], 0x30000
decl[symbol, inp_route_base_trie], 0x30100
decl[symbol, inp_ftable_base], 0x8100

//declarations for the packet processing pipeline:
//mpacket receiver:
decl[comp, mpr. mpr. ./receiver.tmd], inp_descripter_base, inp_packet_buff_base, inp_scratchpad_mib, glo_rdready_sem#,
            glo_rec_sem#, glo_dma_sem#, entry_router initializer, exit_to_verifier#
//ipv4 verifier
decl[comp, ipv4_verifier, ipv4_verifier. ./ipv4_verifier.tmd], inp_descriptor_base, inp_packet_buff_base,
              entry_verificationbegin#, exit_to_checksum#, exit_to_end_verify#, exit_to_end_verify2#
//ipv4_checksum_verifier
decl[comp, ipv4_checksum_verifier, ipv4_checksum_verifier, ./ipv4_checksum_verifier.tmd]. entry_ipv4_cksum_verify#,
               exit_chksum_verify#, exit_to_router#
//ipv4_header_modifier
decl[comp, ipv4_header_modifier, ipv4_header_modifier, ./ipv4_header_modifier.tmd], inp_descriptor_base, inp_packet_buff_base,
             entry_ipv4_header_modifier#, exit_ipv4_header_modifier#
//ipv4_trie_lookup
decl[comp, ipv4_trie_lookup, ipv4_trie_lookup, ./ipv4_trie_lookup.tmd], inp_descriptor_base, inp_packet_buff_base,
               inp_route_base_64k, inp_route_base_256, inp_route_base_trie, inp_ftable_base, entry_routelookup#, exit_routelookup#
//packet_queue
decl[comp, packet_queue, packet_queue, ./packet_queue.tmd], inp_descriptor_base, inp_scratchpad_mib,
                 inp_scratchpad_pwp, entry_packet_queue#, exit_packet_queue#

//declaration of bindings for the packet processing pipeline
decl[binding, b0], mpr, exit_to_verifier#, ipv4_verifier, exntyr_verificationbegin#
decl[binding, b1], ipv4_verifier, exit_to_checksum#, ipv4_checksum_verifier, entry_ipv4_cksum_verify#
decl[binding, b2], ipv4_verifier, exit_to_end_verify1#, ipv4_header_modifier, entry_ipv4_header_modifier#
decl[binding, b3], ipv4_verifier, exit_to_end_verify2#, ipv4_header_modifier, entry_ipv4_header_modifier#
decl[binding, b4], ipv4_checksum_verifier, exit_chksum_verify#, ipv4_header_modifier, entry_ipv4_header_modifier#
decl[binding, b5], ipv4_checksum_verifier, exit_to_router#, mpr, entry_routerinitialize#
decl[binding, b6], ipv4_header_modifier, exit_ipv4_header_modifier#, ipv4_trie_lookup, entry_routelookup#
decl[binding, b7], ipv4_trie_lookup, exit_routelookup#, packet_queue, entry_packet_queue#
decl[binding, b8], packet_queue, exit_packet_queue#, mpr, entry_routeinitialize#

//global variables
decl[gv, var0], mpr, glo_rdready_sem#
decl[gv, var1], mpr, glo_rec_sem#
decl[gv, var2], mpr, glo_dma_sem#

//declaration of the packet processing pipeline that received and enqueues packets
decl[pline, ipv4_recv, 4], mpr, ipv4_verifier, ipv4_checksum_verifier, ipv4_header_modifier, ipv4_trie_lookup, packet_queue,
                 b0, b1, b2, b3, b4, b5, b6, b7, b8, var0, var1, var2

//declaration of the packet transmitting pipeline
decl[comp, scheduler, scheduler, ./scheduler.tmd]
decl[pline, ipv4_omit, 4], scheduler

//aggregate structures
decl[va, aggr0], ipv4_recv, var0
decl[va, aggr1], ipv4_recv, var1
decl[va, aggr2], ipv4_recv, var2

//declaration of the datapath
decl[dpath, ipv4], ipv4_recv, ipv4_xmit, aggr0, aggr1, aggr2
```

Figure 11. Network processor specification of the Ipv4 data path.

For example, the location of a 64K table used in a 16-bit trie lookup is declared as an input argument named 'inp_route_base_64k'. This input argument is used for customizing the IPv4 trie lookup module. The numerical value assigned to this argument is 0x20000, which is an SRAM address location. A different table location could be specified by introducing a different numerical value in the declaration of the symbol inp_route_base_64k.

In the second part of the network processor specification the components that constitute the processing stages and pipelines of the IPv4 data path are declared and their binding symbols specified. Each component is listed using a 'decl' command. Symbols associated with each component are specified as optional tokens. Symbols include input arguments, global variables, entry points and exit points as mentioned earlier. For example, the component that performs IPv4 routing lookup (ipv4_trie_lookup) is listed followed by eight symbols as shown in Figure 11. The first symbol (inp_descriptor_base) is an input argument specifying the SRAM location of a list of packet buffer descriptors used by the IPv4 data path. The second symbol (inp_packet_buff_base) is an input argument specifying an SDRAM location where packet buffers are stored. The next three symbols (i.e. inp_route_base_64k, inp_route_base_256 and inp_route_base_trie) are input arguments specifying SRAM locations of tables constituting a trie data structure. This trie data structure is used for performing a longest prefix matching (LPM) search on the destination IP address of each packet. The next symbol (inp_ftable_base) is an input argument specifying an SDRAM location where next hop information is stored. Finally, the last two symbols (i.e. entry_routelookup# and exit_routelookup#) specify the module's entry and exit points, respectively. The entry point symbol (entry_routelookup#) refers to the first instruction of the trie lookup module, where the IPv4 routing lookup process begins. The exit point symbol (exit_routelookup#) refers to a branch instruction executed at the end of the IPv4 routing lookup process, where the sequence of control jumps to another module.

After the declaration of components, bindings are declared. Nine bindings are declared in the example of Figure 11 named b0–b8. The declaration of bindings reflects the sequence of control and inter-module relationship characterizing the IPv4 data path of Figure 10. The execution of the IPv4 data path begins with the execution of a virtual network demultiplexor module (receiver.tmd). After the execution of this module a packet is passed into an IPv4 verifier module (binding b0). If the packet header contains valid version and TTL fields, the sequence of control jumps to the checksum verifier module (binding b1). However, if the packet header does not contain valid header fields, an exception is raised and the sequence of control jumps to the IPv4 header modifier module bypassing the checksum verifier module (bindings b2 and b3). After the checksum verifier module is executed the sequence of control jumps to the IPv4 header modifier module (binding b4). This module decrements the TTL field of the packet header and recomputes the checksum. In case the verification of the checksum field fails an exception is raised and the sequence of control jumps back to the virtual network demultiplexor module (binding b5). Packets with valid checksum fields are passed to the IPv4 header modifier module and then to the IPv4 trie lookup module (binding b6). Once routing lookup takes place, packets are placed in appropriate queues (binding b7) and the sequence of control jumps back to the virtual network demultiplexor (binding b8). Each binding is declared using four tokens. Tokens specify the source component, exit point, destination component and entry point associated with each binding.

Following the declaration of bindings, global variables are declared. Next, packet-processing pipelines are declared as collections of components, bindings and global variables. The IPv4 data path

described in the specification of Figure 11 consists of two pipelines: an 'ipv4_recv' pipeline which places packets into queues and an 'ipv4_xmit' pipeline which schedules the transmission of packets into the network. In the last statement of the specification the IPv4 data path is declared as a collection of pipelines and global variables.

Example of a component

In what follows we describe how one of the components of the IPv4 data path is implemented. We present the checksum verifier module as it is implemented as part of the NetBind source code distribution [4].

The checksum verifier module verifies the checksum field of an IPv4 header. The checksum verifier module consists of 23 microcode instructions and can be executed by any number of hardware contexts between one and four in a single IXP1200 microengine. The checksum verifier module exports two pipeline registers and can be linked with other modules using two entry points and two exit points. The microcode that implements the checksum verifier module is shown in Figure 12. In the first part of the code, entry and exit points are declared using the '.export_func' directive of the IXP1200 microassembler. Next, pipeline and local registers are declared.

Pipeline register names are distinguished from local register names because pipeline register names begin with the underscore ('_') character. Two pipeline registers are declared and exported as a programming API for the component. The '_descriptor_address' pipeline register holds the address of a packet buffer descriptor. This packet buffer descriptor indicates the location of a buffer where a packet is stored in the SDRAM memory unit. The '_exception' pipeline register holds a flag that indicates errors or failure during the reception of a packet.

After the declaration of pipeline and local registers, memory transfer registers are declared. Register addresses are allocated statically. Eight SDRAM transfer registers are declared in the code (i.e. $$xdfer0–$$xdfer7). These registers hold the fields of the IPv4 packet header that are examined by the checksum verifier module. The first instruction of the module is marked with the label 'entry_cksum_verify#' indicating that this instruction is the module's entry point. The checksum verify process divides every word in the packet header into two 16-bit parts and adds the parts of every word producing a sum. The carry resulting from each addition is added into this sum. If the checksum field in the packet header is correct, then the result from all additions is zero. If the value of the checksum field is not correct, then an exception is raised and the buffer space for the packet is de-allocated. The packet is discarded, in this case. The program sequence executes the branch instruction labeled 'exit_to_router#'. This is an exit point leading to the beginning of the packet receiving code. Otherwise, if the value of the checksum field is correct, then program sequence executes the branch instruction labeled 'exit_chksum_verify#', which is an exit point leading to the next component in the packet processing pipeline. Both exit points jump to the same label 'end#' in the microcode of Figure 12. This happens because the component code of Figure 12 reflects the component state before binding. At development time all exit points of components branch to the same instruction which is a 'nop' (i.e., no operation) instruction separating transport modules. During binding the destination addresses of branch instructions are modified. The destination addresses of branch instructions are set to the entry points of the components where the sequence of control jumps to, as explained earlier.

```
//entry points
.export_func oneway 0123 entry_ipv4_cksum_verify# 0 0
.export_func oneway 0123 entry_ipv4_end_verify# 0 0

//exit points
.export_func oneway 0123 entry_ipv4_chksum_verify# 0 0
.export_func oneway 0123 entry_to_router# 0 0

//REGISTER ALLOCATIONS:
.areg _exception 3
.areg local_a6 6
.breg local_b8 8
.areg _descriptor_address 0

.$$reg $$xdfer0 0
.$$reg $$xdfer1 1
.$$reg $$xdfer2 2
.$$reg $$xdfer3 3
.$$reg $$xdfer4 4
.$$reg $$xdfer5 5
.$$reg $$xdfer6 6
.$$reg $$xdfer7 7

.xfor_order $$xdfer0 $$xdfer1 $$xdfer2 $$xdfer3 $$xdfer4 $$xdfer5 $$xdfer6 $$xdfer7

entry_ipv4_cksum_verify#
// Checksum is in the two least significant bytes of
//transfer register 4

alu[local_b8, 0, +16, $$xdfer1]
alu[local_b8, local_b8, +, $$xdfer2]
alu[local_b8, local_b8, +carry, $$xdfer6, >>16]
alu[local_b8, local_b8, +carry, $$xdfer3]
alu[local_b8, local_b8, +carry, $$xdfer4]
alu[local_b8, local_b8, +carry, $$xdfer5]

alu[local_b8, local_b8, +carry, 0]
// add in previous carry
id_field_w_clr[local_a6, 1100, local_b8]
// get high 16 of the total
alu_shf[local_b8, local_a6, +, local_b8, <<16]
// add low 16 bits to upper 16
alu[local_a6, 1, +carry, local_a6, <<16]
// add last carry +1, local_a6 B op=0
alu[local_b8, local_b8, *, local_a6, <<16]
// add 1<<16 to 0xffff to get zero result

br-0[entry_ipv4_end_verify#]
alu[_exception, --, B, 0x0A] //IP_BAD_CHECKSUM
entry_ipv4_end_verify#:

alu[- -, - -, B, _exception]
br=0[exit_chksum_verify#]

//else if the packet is not correct
//free the buffer space and start all over
//again

immed_w0[local_b8, 0x0000]
immed_w1[local_b8, 0x0010]

alu_shf[- -, local_b8, OR, 0x0000]
// merge ov bit with freelist id/bank

sram[push, - -, _descriptor_address, 0, 0], indirect_ref, ctx_swap

exit_to_router#:
br[end#]

nop
exit_chksum_verify#:
br[end#]
end#:
nop
```

Figure 12. Checksum verifier module.

Hardware context and register allocations

The IPv4 data path of Figure 10 is split into two pipelines as shown earlier. The first pipeline implements the reception of packets from the network. This pipeline is executed by four hardware contexts each serving a different port of the Bridalveil development board (see below). The input FIFO slots of the IXP1200 network processor are evenly divided between the four contexts and assigned to each port in a static manner. On the transmit side, however, the scheduling of contexts is done dynamically. We use three hardware contexts to transmit packets into the network on an on-demand basis and one context to assign packet transmission tasks to the transmit contexts. The same approach has been followed in the IPv4 data path reference design distributed by Intel.

Hardware context management is a difficult problem. There is no single solution that satisfies all types of applications and network processor hardware. Implementing network algorithms in software typically requires sophisticated hardware context assignment and synchronization. For example, in a router system where the number of output ports is much larger than the number of hardware contexts available for transmission, static allocation of contexts may not be efficient. In other router systems such as the Bridalveil development boards, static allocation of contexts may result in better performance [11,12]. Another case where static allocation of contexts is inefficient is packet scheduling. A scheduler may need to select the next eligible packet for transmission from a large number of queues or connections. In this case, assigning a single thread for serving each queue may not be possible. In a software implementation of a hierarchical packet scheduler, it is preferable if single-level schedulers are served by different contexts. In this way the dequeuing process at each level of the hierarchy can complete without waiting for packets to be enqueued from previous levels. These contexts need to be assigned dynamically to different levels of the hierarchy.

These examples indicate that it is better if binding systems offer the flexibility to programmers to select hardware context allocation and management policies among a range of choices. We believe that programmers should have the flexibility to select the context allocation and management policies which are more suitable for the algorithms programmed in the data path. In NetBind contexts can be assigned to input ports, output ports and other types of resources (e.g. queues) either statically or dynamically. At the admission control level, the resource requirements of each component are presented as the total number of contexts needed. Port and queue allocations are hidden from the admission controller. The benefit of this approach is that the binding and admission control processes are simplified because port and queue allocations are not taken into account. The disadvantage of this approach is that additional responsibility is placed on the component developer which needs to know details about the way ports and queues operate in different router systems. In addition, a strong trust relationship is assumed between programmers at different levels of the specification hierarchy of Figure 3. The programmers at the virtual router specification level assume that component bindings are successfully resolved at the network specification level. Similarly, programmers at the network processor specification level assume that bindings are successfully resolved at the IXP1200 specification level and that components manage hardware contexts, ports and queues efficiently.

While port and queue allocations are hidden from the binder and admission controller, register allocations are exposed. Pipeline and input argument registers are assigned statically as

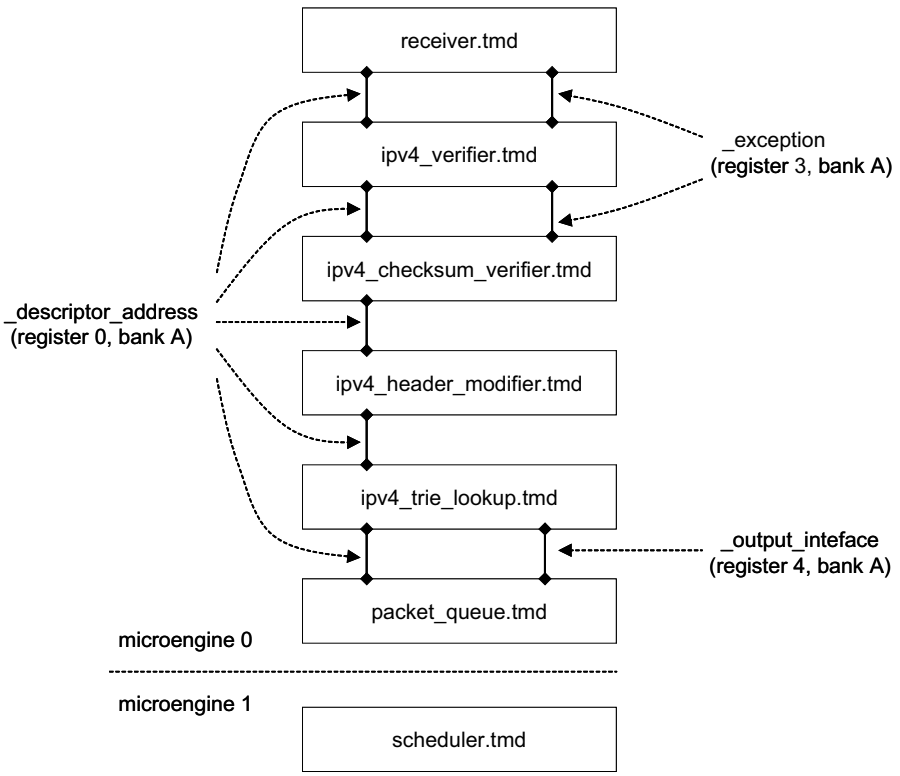

Figure 13. Pipeline register allocations in the Ipv4 data path.

discussed earlier, while global variable registers are assigned dynamically. In Figure 13 we illustrate the path which a packet follows through the router system of Figure 10, and the pipeline registers used in this path.

EVALUATION

Experimental environment

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 [32] is an IXP1200 network processor development board running at 200 MHz and using 256 Mbytes of SDRAM off-chip memory. The Bridalveil board is a PCI card that can be connected to a PC running Linux. The host processor of the PC and the IXP1200 unit can communicate over the PCI bus. The PC

serves as a file server and boot server for the Bridalveil system. PCI bus network drivers are provided for the PC and for the embedded ARM version of Linux running on the StrongARM Core processor. In our experimental environment, each Bridalveil card is plugged into a different PC. In each card, the IXP1200 chip is connected to four fast Ethernet ports that can receive or transmit packets at 100 Mbps.

To evaluate NetBind we tested the NetBind code using the Bridalveil system where we could also execute MicroACE. MicroACE is a systems architecture provided by Intel for composing modular data paths and network services in network processors. The MicroACE adopts a static binding approach to the development of modular data paths based on the insertion of a 'dispatch loop' code stub in the critical path.

The dispatch loop is provided by the programmer and directs the program flow through the components of a programmable processing pipeline. MicroACE is a system that can be used for programming microengines and the StrongARM Core. In what follows we provide an overview of MicroACE.

MicroACE is an extension of the ACE [32] framework that can execute on the microengines of IXP1200. In MicroACE, an application defines the flow of packets among software components called 'ACEs' by binding packet destinations called 'targets' to other ACEs in a processing pipeline. A series of concatenated ACEs form a processing pipeline. A MicroACE consists of a 'microblock' and a 'core component'. A microblock is a microcode component running in the microengines of IXP1200. One or more microblocks can be combined into a single microengine image called 'microblock group'. A core component runs on the StrongARM Core processor. Each core component is the counterpart of a microblock running in the microengines. The core component handles exception, control and management packets.

The binding between microblocks in the ACE framework is static and takes place offline. The targets of a microblock are fixed and cannot be changed at run time. For each microengine, a microcode 'dispatch loop' is provided by the programmer, which initializes a microcode group and a pipeline graph. The size of a microblock group is limited by the size of the instruction store where the microcode group is placed.

Before a microblock is executed, some global binding state needs to be examined. The global binding state consists of two variables. A 'dl_next_block' variable holds an integer identifying the next block to be executed, whereas a 'dl_buffer_handle' variable stores a pointer to a packet buffer exchanged between components. Specialized 'source' and 'sink' microblocks can send or receive packets to/from the network. Since the binding between microblocks takes place offline, a linker can preserve the persistency of register assignments.

Dynamic binding analysis

MicroACE and NetBind have some similarities in their design and realization, but they also have differences. MicroACE allows the programmer to construct processing pipelines in the StrongARM Core and the microengines. NetBind, on the other hand offers a set of libraries for the microengines only. MicroACE supports static linking allowing programmers to use registers according to their own preferences. MicroACE does not impose any constraints on the number and purpose of registers used by components. NetBind, on the other hand, supports dynamic binding between components that can take place at run time. To support dynamic binding NetBind imposes a number of constraints on the purpose and number of registers used by components.

The main difference between NetBind and MicroACE in terms of performance comes from the choice of the binding technique used in each scheme. MicroACE follows a dispatch loop approach where some global binding state needs to be checked before each component is executed. In NetBind there is no explicit maintenance of global binding state. Components are associated with each other at run time through the modification of their microcode. The modification of microcode at run time, which is fundamental to our approach, poses a number of security challenges, which our research has not yet addressed. Another problem with realizing dynamic binding in the IXP1200 network processor is that microengines need to temporarily terminate their execution when data paths are created or modified. Such termination may disrupt the operation of other data paths potentially sharing the resources of the same processing units. We are investigating methods to seamlessly accomplish dynamic binding when multiple data paths share the resources of the same network processor as part of our future work.

To compare NetBind against MicroACE we estimate the available headroom for the microengines of the IXP1200 network processor. In what follows, we discuss a methodology on how headroom can be calculated in a multi-threaded network processor architecture. We assume that a single pipeline is used for forwarding packets. We also assume that each port of a network processor can forward packets at line speed C, and that a minimum size packet is $m$bits. The maximum time a packet is allowed to spend on the network processor hardware without violating the line rate forwarding requirement is

$$T = \frac{m}{C} \tag{1}$$

During this time microengine resources are shared between $n$hardware contexts. It is fair to assume that each thread gets an equal share of the microengine resources on average, and that each packet is processed by a single thread only. Therefore, the maximum time a thread is allowed to spend processing a packet without violating the line rate-forwarding requirement is

$$\frac{T}{n} = \frac{m}{n \cdot C} \tag{2}$$

Typically, microengine resources do not remain utilized all the time. For example, there may be cases when hardware contexts attempt to access memory units at the same time. In these cases, all contexts remain idle until a memory transfer operation completes. Therefore, we need to multiply the maximum time calculated above with a utilization factor ρ in order to have a good estimation of the network processor headroom. A final expression for the network processor headroom H is given below, expressed in microengine cycles, where t_c is the duration of each cycle:

$$H = \frac{\rho \cdot m}{n \cdot C \cdot t_c} \tag{3}$$

The utilization factor was measured when our system was running the IPv4 data path, and was found to be equal to 0.98. We doubled the percentage of idle time to have an even worse estimation of the utilization factor, resulting in $\rho = 0.96$. After substituting $C = 100$ Mbps, $m = 64$ bytes, $n = 4$, and $t_c = 5$ ns into (3), we find that the headroom Hin the IXP1200 Bridalveil system is 246 microengine cycles.

We evaluated the performance of the IPv4 data path when no binding is performed (i.e. the data path is monolithic) and when the data path is created using NetBind and MicroACE. We also implemented a simple binding tool based on the vector table binding technique. We used this tool in our experiments in order to compare the three binding techniques presented earlier in a quantitative manner.

Table I. Binding instructions in the data path.

Binding technique	Per component binding instructions			
	Register operation	Conditional branch	Unconditional branch	Memory (scratch) transfer
NetBind	1	1	N/A	N/A
Dispatch loop (MicroACE)	2	1	2	N/A
Vector table	N/A	N/A	1	1

Table II. Component sizes and symbols in the IPv4 data path.

Module name	Size	Input argument	Global variation	Entry points	Exit points
Receiver (initialization)	48	N/A	N/A	N/A	N/A
Receiver	100	3	3	3	3
Ipv4_verfier	17	2	0	1	3
Ipv4_checksum_verifier	16	0	0	2	1
Ipv4_header_verifier	23	2	0	1	1
Ipv4_trie_lookup	104	6	0	1	1
packet_queue	44	3	0	1	1
Aggregate (excluding initialization)	304	16	3	9	10

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. Table I shows per component binding instructions in the data path. Table II shows the additional instructions that are inserted in the data path for each binding technique. Our implementation of the IPv4 data path is broken down into two processing pipelines: a 'receiver' pipeline consisting of six components (from 'receiver' to 'packet_queue' in Table II) and a 'transmitter' pipeline consisting of single component ('scheduler.tmd' in Figure 9). The receiver and transmitter pipelines are executed by microengines 0 and 1, respectively. Table II shows the number of instructions, input arguments, global variables, entry points, and exit points associated with each component of the receiver pipeline.

We added a 'worst-case' generic dispatch loop to evaluate the MicroACE data path. The dispatch loop includes a set of comparisons that determine the next module to be executed on the basis of the value of the 'dl_next-block' global variable. The loop is repeated for each component. We added five instructions for each component in the dispatch loop: (i) an 'alu' instruction to set the 'dl_next_block' variable to an appropriate value; (ii) an unconditional branch to jump to the beginning of the dispatch loop; (iii) an 'alu' instruction to check the value of the 'dl_next_block' variable; and (iv) a conditional and an unconditional branch to jump to the appropriate next module in the processing pipeline.

The vector table binding technique works as follows. At the initialization stage, we retrieve the entry point locations using 'load_addr' instructions. The vector is then saved into the 4K scratch memory of

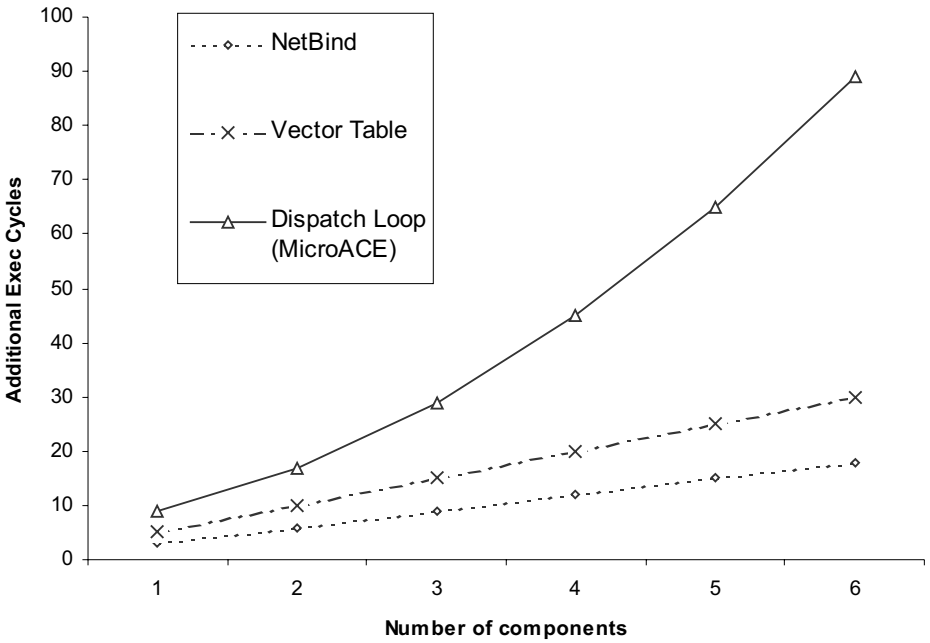

Figure 14. Dynamic Binding Overhead.

IXP1200. For each component, we insert a scratch 'read' instruction to retrieve the entry point from the vector table and a 'rtn' instruction to jump to the entry point of the next module. The performance of the vector table scheme is heavily dependent on the speed of a memory access. If the vector for the next component in the pipeline can be retrieved in advance, the overhead of binding can be reduced drastically to five cycles per binding. The advantage of having a multithreaded network processor is that memory access latencies can be hidden if the processor switches context when performing time consuming memory transfer operations.

Figure 14 shows additional execution cycles for each binding technique. From the figure, we observe 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.

Figure 14 illustrates that the vector table technique is not that much worse than NetBind in the IXP1200 network processor. However, memory latencies are much larger in network processors that target higher speeds than fast Ethernet. In commercial network processors that target OC-48 and OC-192 speeds memory access latencies can be several times larger than the network processor headroom,

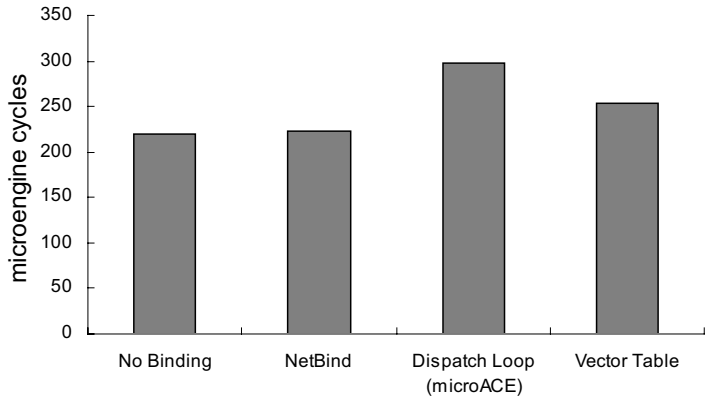

Figure 15. Per-packet execution time.

as stated earlier. For example it is reported in [33] that memory access latencies can be as large as 100–300 compute cycles in the IXP2400 and IXP2800 network processors. In the IXP2800 network processor the headroom for a single microengine can be as small as 57 compute cycles. The vector table technique requires at least twice the processing headroom for fetching a single component in this network processor. For this reason it is not easy to support modular line rate forwarding with the vector table technique. Memory access latencies are likely to be even larger in next generation network processors targeting faster line rates (e.g. OC-768). The NetBind approach is suitable for this type of processors because it adds a binding overhead of two compute cycles per component which is insignificant.

Figures 15 and 16 show per-packet execution times and packet processing throughput for the three binding techniques, and a monolithic data path. The term 'MAC packets' used in Figure 16, refers to minimum size Ethernet packets of 64 bytes each. The measurements were taken for the case where the data path is split between six components, and packets are forwarded at the maximum possible rate from four input ports to the same output port. To forward packets at the maximum possible rate independent of the input port speed, we applied the technique suggested in [12], where iterations of the input forwarding process forward the same packet from the input FIFO slot avoiding port interaction. NetBind has 2% overhead (execution time) over the monolithic implementation. The vector table and dispatch loop implementation have 12 and 32% overhead, respectively, over the performance of the pipeline created by NetBind. Collecting measurements on an IXP1200 system is more difficult than we initially thought. Since microengines do not have access to an onboard timer, measurements have to be collected by a user program running on the StrongARM core processor. We created a program consisting of a timing loop that is repeated for accuracy. The program stops executing and prints out the time difference measured after a significant number of iterations have taken place.

We have measured the time to install a new data path using NetBind. Stopping and starting the microengines takes 60 and 200 μs, respectively. The binding algorithm and the process of writing data path components into instruction stores takes 400 μs to complete. Measurements were taken using the Bridalveil's 200 MHz StrongArm core processor.

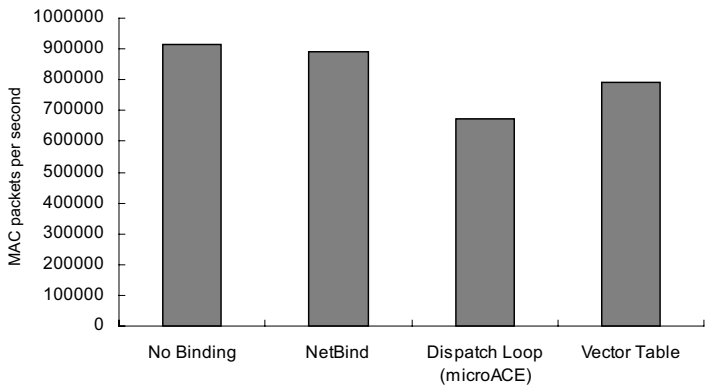

Figure 16. Packet processing throughput.

Loading six modules from a remotely mounted NFS server takes about 60 ms 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.

NetBind limitations

NetBind offers significant improvement in terms of data path performance and composition time over the alternative choices of dispatch loops and indirection. However, NetBind imposes a number of constraints on the programmer. The most significant limitation of NetBind is that the register APIs exported by consecutive components in a packet-processing pipeline need to match with each other. The physical register addresses exported by a component need to be the same as the physical register addresses exported by every subsequent component in packet-processing pipelines. This is a significant restriction. For example, a routing lookup component needs to place the identifier to an outgoing interface into the register where every subsequent component (e.g. a packet queue) expects to find this value.

While this seems to be a significant restriction, we believe that NetBind allows programmers to effectively write code for a small but significant class of data path applications. NetBind is most efficient for applications that manipulate packet header fields, such as packet classification, forwarding and traffic management. The number of parameters that need to be exchanged between components for this class of applications is typically small and well defined. For example, parameters exchanged between components may include packet descriptors, interface identifiers, classifier actions, packet time stamps or eligibility times. In order to effectively write code using NetBind, component developers need to determine the set of components which interact with their own data path components. Next, component developers need to make sure that the register APIs which their components export match

with the register APIs of subsequent components in packet-processing pipelines. This may not be difficult if the number of parameters exchanged between components is limited and well defined.

Another limitation of the NetBind approach is that a strong trust relationship is assumed between developers at different levels of the data path specification hierarchy. As mentioned earlier, the programmers at the virtual router specification level assume that component bindings are successfully resolved at the network specification level. Similarly, programmers at the network processor specification level assume that bindings are successfully resolved at the IXP1200 specification level and that components manage hardware contexts, ports and queues effectively.

The design of the NetBind tool is independent of the target line rate. This is because the binding system can construct data paths from many different packet receiving and transmitting components designed for different network interfaces and speeds. Port and queue allocations are hidden from the binder and admission controller. While this practice places extra burden on the programmer it simplifies the binding process and makes the system applicable to a range of network processors. We have not investigated forwarding at Gigabit speeds (e.g. OC-48, OC-192). However, there is strong indication that NetBind can be applied to these line rates due to the simplicity and generality of the binding process. Porting NetBind to network processors that target the OC-48 and OC-192 line rates is left for future work.

RELATED WORK

Programmable routers represent an active area of research. Click [8] is an extensible architecture for creating software routers in commodity operating systems. A Click router is constructed by selecting components called 'elements' and connecting them into directed graphs. Click components are implemented using C++ and, thus, inherit the binding overhead associated with using a higher-level programming language to construct data paths. The software [9] and hardware [10] plugins projects are investigating extensibility in programmable Gigabit routers. These routers are equipped with port processors, allowing for the insertion of software/hardware components, where hardware plugins are implemented as reconfigurable logic. The work on Scout OS [34] is addressing the problem of creating high-performance data paths using general-purpose processors. The installation of packet forwarders in network processors has been discussed in [12]. Packet forwarders discussed in [12] are rather monolithic in nature and their installation system does not support binding.

Run-time machine language code generation and modification has been proposed as part of the work on the Synthesis Kernel [35]. The Synthesis Kernel aims for improving kernel routine performance in general-purpose processors as well.

CONCLUSION

We have presented the design, implementation and evaluation of the NetBind software system, and compared its performance to the Intel MicroACE system, evaluating the binding overhead associated with each approach.

While the community has investigated techniques for synthesizing kernel code and constructing modular data paths and services, the majority of the literature has been focused on the use of

general-purpose processor architectures. Little work has been done using 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 [4] for experimentation.

ACKNOWLEDGEMENTS

The work was done while the author was a PhD student of Columbia University, New York. This research was supported by grants from the NSF CAREER Award ANI-9876299 and the Intel Research Council on 'Signaling Engines for Programmable IXA Networks'.

REFERENCES

1. Network Processing Forum. http://www.npforum.org [9 March 2005].
2. Intel Exchange Architecture, programmable network processors for today's modular networks. *White Paper*, Intel Corporation, 2000. Available at: http://www.intel.com/design/network/white_paper.htm.
3. IBM Corporation. IBM Power Architecture. http://www-03.ibm.com/technology/power [9 March 2005].
4. The NetBind Project home page. http://www.comet.columbia.edu/genesis/netbind [9 March 2005].
5. Kounavis ME, Campbell AT, Chou S, Modoux F, Vicente J, Zhang H. The Genesis Kernel: A programming system for spawning network architectures. *IEEE Journal on Selected Areas in Communication* 2001; **19**(3):511–526.
6. Campbell AT, Gormez J, Kim S, Valko A, Wan C, Turanyi Z. Design implementation, and evaluation of cellular IP. *IEEE Personal Communications Magazine* 2000; **7**(4):42–49.
7. Intel Corporation. *IXP1200 Network Processor Datasheet*, December 2000.
8. Kohler E, Morris R, Chen B, Jannotti J, Kaashoek M. The Click modular router. *ACM Transactions on Computer Systems* 2000; **18**(3):263–297.
9. Wolf T, Turner J. Design issues for high-performance active routers. *IEEE Journal on Selected Areas in Communications* 2001; **19**(3):404–409.
10. Taylor D, Turner J, Lockwood J. Dynamic Hardware Plugins (DHP): Exploiting reconfigurable hardware for high-performance programmable routers. *Proceedings of the 4th IEEE International Conference on Open Architectures and Network Programming*, Anchorage, Alaska, April 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001.
11. Spalink T, Karlin S, Peterson L. Evaluating network processors in IP forwarding. *Technical Report TR-626-00*, Princeton University, Princeton, NJ, 15 November 2000.
12. Spalink T, Karlin S, Peterson L, Gottlieb Y. Building a robust software-based router using network processors. *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001. ACM Press, 2001; 216–229.
13. Karlin S, Peterson L. VERA: An extensive router architecture. *Proceedings of the 4th IEEE International Conference on Open Architectures and Network Programming*, April 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 3–14.
14. Intel IXP2400 network processor product information. http://www.intel.com/design/network/products/npfamily/ixp2400.htm [9 March 2005].
15. Intel IXP2800 network processor product information. http://www.intel.com/design/network/products/npfamily/ixp2800.htm [9 March 2005].
16. Pappu P, Wolf T. Scheduling processing resources in programmable routers. *Proceedings of the 21st IEEE Conference on Computer Communications (INFOCOM)*, New York, NY, June 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002.
17. Shreedhar M, Varghese G. Efficient fair queueing using deficit round robin. *Proceedings of SIGCOMM'95*. ACM Press, 1995.
18. Demers A, Keshav S, Shenker S. Analysis and simulation of a fair queueing algorithm. *Proceedings of the ACM SIGCOMM'89. Journal of Internetworking Research and Experience* 1990; **1**(1):3–26.
19. Goyal P, Vin HM, Cheng H. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking* 1997; **5**(5):690–704.

20. Bennett JCR, Zhang H. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking* 1997; **5**(5):675–689.
21. Johnson EJ, Kunze AR. *IXP1200 Programming*, ch. 13. Intel Press, 2002.
22. Kounavis ME, Kumar A, Vin H, Yavatkar R, Campbell AT. Directions for packet classification in network processors. *Proceedings of the 2nd Workshop on Network Processors*, Anaheim, CA, February 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003.
23. Gupta P, McKeown N. Packet classification on multiple fields. *Proceedings ACM SIGCOMM'99*, Harvard University, September 1999. ACM Press, 1999.
24. Rexford J, Greenberg A, Bonomi F. Hardware-efficient fair queueing architectures for high-speed networks. *Proceedings of IEEE INFOCOM*, March 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996; 638–646.
25. Intel Corporation. *IXP1200 Network Processor Programmer's Reference Manual*, December 2000.
26. Postel J (ed.). Internet Protocol. *Request For Comments 791*, September 1981.
27. Intel Corporation. *IXP1200 Network Processor Development Tools User's Guide*, December 2000.
28. The Genesis Project home page. http://www.comet.columbia.edu/genesis [9 March 2005].
29. Vinoski S. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine* 1997; **35**(2):46–55.
30. Schmidt D, Cleeland C. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine* 1999; **37**(4):54–63.
31. NetBind Software Programmer's Guide. http://www.comet.columbia.edu/genesis/netbind/documentation [9 March 2005].
32. Intel Corporation. *Intel IXA SDK ACE Programming Framework Reference*, June 2001.
33. Lakshmanamurthy S, Liu K-Y, Pun Y, Huston L, Naik U. Network processor performance analysis methodology. *Intel Technology Journal* 2002; **6**(3):19–28.
34. Montz A, Mosberger D, O'Malley S, Peterson L, Proebsting T. Scout, a communication oriented operating system. *Operating System Design and Implementation*. USENIX Association, 1994.
35. Pu C, Massalin H, Ioannidis J. *The Synthesis Kernel*. Springer: Berlin, 1988.