

# NetBind: A Binding Tool for Constructing Data Paths in Network Processor-Based Routers

Andrew T. Campbell, Stephen T. Chou, Michael E. Kounavis, Vassilis D. Stachtos and John Vicente

**Abstract**-- There is growing interest in network processor technologies capable of processing packets at line rates. In this paper, we present the design, implementation and evaluation of *NetBind*, a 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. Data paths constructed using *NetBind* seamlessly share the resources of the same network processor. We compare the performance of *NetBind* to the *MicroACE* system developed by Intel to support binding between software components running on Intel IXP1200 network processors. We evaluate these alternative approaches in terms of their binding overhead, and discuss how this can affect the forwarding performance of IPv4 data paths running on IXP1200 network processor-based routers. We 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 ([comet.columbia.edu/genesis/netbind](http://comet.columbia.edu/genesis/netbind)) for experimentation.

## I. INTRODUCTION

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

Data path modularity and extensibility requires the dynamic binding between independently developed packet processing components. Traditional techniques for realizing dynamic binding, (e.g., insertion of code stubs or indirection through function tables), cannot be applied to

network processors because these techniques introduce considerable overhead in terms of additional instructions in the critical path. One solution to this problem is to optimize the code produced by a binding tool, once data path composition has taken place. Code optimization algorithms can be complex and time-consuming, however, and thus not 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.

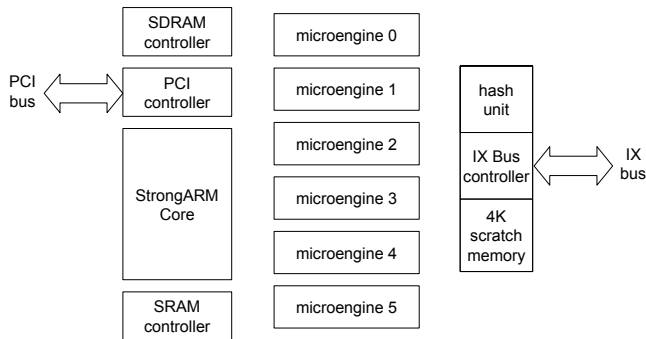
In this paper, we present the design, implementation and evaluation of *NetBind*, a high performance, flexible and scalable binding tool for creating composable data paths in network processor-based routers. By “high performance” we mean that *NetBind* can produce data paths that forward minimum size packets at line rates without introducing significant overhead in the critical path. By “flexible” we mean that *NetBind* allows data paths to be composed at a fine granularity from components supporting simple operations on packet headers and payloads. By “scalable” we mean that *NetBind* can be used across a wide range of applications and time scales. *NetBind* can be used to dynamically program a diverse set of data paths. For example, in [14] we show how Cellular IP [6] data paths can be composed for network processor-based radio routers. *NetBind* can create packet-processing pipelines through the dynamic binding of small pieces of machine language code. In *NetBind*, data path components export symbols, which are used during the binding process. 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. In order to support fast data path composition, *NetBind* reduces the number of binding operations required for constructing data paths to a minimum set so that binding latencies are comparable to packet forwarding times. While the design of *NetBind* is guided by a set of general principles that make it applicable to a class of network processors, the current implementation of the tool is focused toward the Intel IXP1200 network processor. Porting *NetBind* to other network processors is for future work.

This paper is structured as follows. In Section II we present an overview of network processor architectures, and IXP1200 in particular, and discuss issues and design choices associated with dynamic binding. Specifically, we investigate tradeoffs that are associated with different

---

*Andrew T. Campbell, Stephen T. Chou, Michael E. Kounavis and Vassilis D. Stachtos are members of the COMET Group, Columbia University, {campbell, schou, mk, vs}@comet.columbia.edu. John Vicente is affiliated with the Intel Corporation and is also member of the COMET Group, Columbia University, John.Vicente@intel.com.*

design choices and discuss their implications on the performance of the data path. In Section III, we present the design and implementation of NetBind. NetBind can create multiple data paths that can share resources of the same IXP1200 network processor. In Section IV, we use a number of NetBind created IPv4 [5] data paths to evaluate the performance of the system. We also evaluate the MicroACE system [7] developed by Intel to support binding between software components running on Intel IXP1200 network processors, and identify pros and cons in comparison to NetBind. Section V discusses the related work, and finally, in Section VI, we provide some concluding remarks.



**Figure 1: Internal Architecture of IXP1200**

## II. DYNAMIC BINDING IN NETWORK PROCESSORS

### A. Network Processors

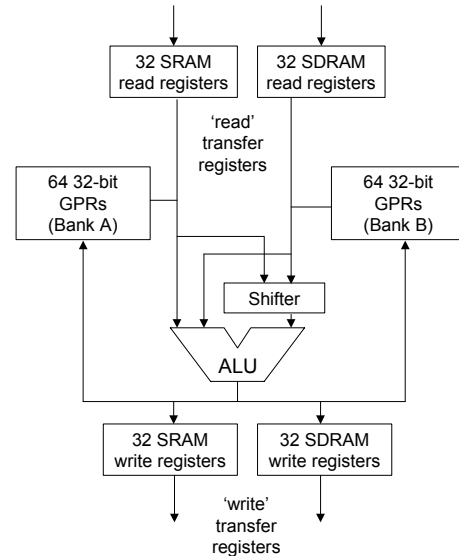
#### 1) Features

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. Network processors represent an alternative approach where multiple processing units, (e.g., the microengines of Intel's IXP1200 [2] or the dyadic protocol processing units of IBM's PowerNP NP4GS3 [3]), offer dedicated computational support for parallel packet processing. Processing units often have their own on-chip instruction and data stores. In some network processor architectures, processing units are multithreaded. Hardware threads usually have a separate program counter and manage a separate set of state variables. However, each thread shares an arithmetic logic unit and register space. Network processors do not only employ parallelism in the execution of the packet processing code, rather, they also support common networking functions realized in hardware, (e.g., hashing [2], classification [3] or packet scheduling [2-3]).

#### 2) The IXP1200 Network Processor

Our study on the construction of modular data paths is focused on the Intel IXP1200 network processor. The IXP1200 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 on the IXP1200 is sufficient to understanding the design and implementation of NetBind. For further details on the IXP1200 see [2].



**Figure 2: Microengine Registers**

One of the IXP1200 RISC CPUs is a StrongARM Core processor running at 166 or 200 MHz. The StrongARM Core can be used for processing slow path exception packets, managing routing tables and other network state information. The StrongARM Core uses a 16K instruction cache and an 8K data cache. The other six RISC CPUs, called “microengines” are used for executing the fast path code. Like the StrongARM Core, microengines run at 166 or 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 1K or 2K called “microstore”.

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 fourth 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 “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 experiment with NetBind, the IXP1200 is connected to eight fast Ethernet (100 Mbps) ports and two gigabit Ethernet (1Gbps) ports. The IX bus controller (shown in Figure 1) incorporates two FIFOs for storing minimum size packets, a hash unit and a 4K “scratch” memory unit. The scratch memory is used for writing or reading short control messages that are exchanged between the microengines and the StrongARM Core.

### B. Dynamic Binding Issues

There are many different techniques for introducing new services into software-based routers [21]. 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 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.

#### 1) 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 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 amount of instructions that can be executed in the critical path is typically small, ranging between some tens to hundreds of instructions. The amount of instructions that can be executed in the critical path beyond minimal IPv4 forwarding is often called the *headroom*.

Headroom is a precious resource in programmable routers. 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 would 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.

#### 2) Register Space and State Management

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 that 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 to the amount of data that is needed 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 to the network.

Register addresses need to be known to component developers in advance. A component needs to place parameter values into registers so they can be correctly accessed by the next component in the processing pipeline.

There are two solutions to this problem. First, the binding mechanism can impose a consensus on the way register sets are used. Each component in a processing pipeline can place parameters into a predetermined set of registers. The purpose of each register, and its associated parameter, can be exposed as a programming API for the component.

A second solution is more computationally intensive. The binding tool can scan all components in a data path at run time and make dynamic register allocations when the data path is constructed or modified. In this case, the machine language code that describes each component needs to be modified reflecting the new register allocations made by the binding tool.

### 3) *Choice of the Binding Method*

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

An enhancement on the first method for dynamic binding adds a small *vector table* to memory. The vector table contains the instruction store addresses where each component is located. A data path component obtains the address of the next component in the processing pipeline in one memory access time. In this approach, no stub code needs to be inserted in the critical path. When a new component is added the content of the vector table needs only 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.

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.

### 4) *Data Path Isolation and Admission Control*

To forward packets without disruption, data paths sharing the resources of the same network processor hardware need to be isolated. In addition, an admission control process

needs to ensure that the resource requirements of data paths are met. Resource assignments can be controlled by a system-wide entity. Resources in network processors include bandwidth, hardware contexts, processing headroom, on-chip memory, register space, and instruction store space.

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 components in advance is a difficult problem. One solution 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 reasonable 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 or weighted fair queuing [22]), can be implemented either in the network processor hardware, or, as part of a packet-processing pipeline. Hierarchical packet scheduling algorithms [23-24] can be used for dividing bandwidth between a hierarchy of coexisting data paths. The implementation of packet scheduling algorithms for network processors is beyond scope of this paper. Details about how we implemented data path isolation and admission control in NetBind can be found in Section III.

### 5) *Processor Handoffs*

Sometimes the footprints of data paths can be too large so 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.

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. Dynamic binding should try to minimize the probability of having processor handoffs in the critical path. This is not an easy task and requires a search to be made on all the possible ways to place the code for data paths into the instruction stores of the processing units.

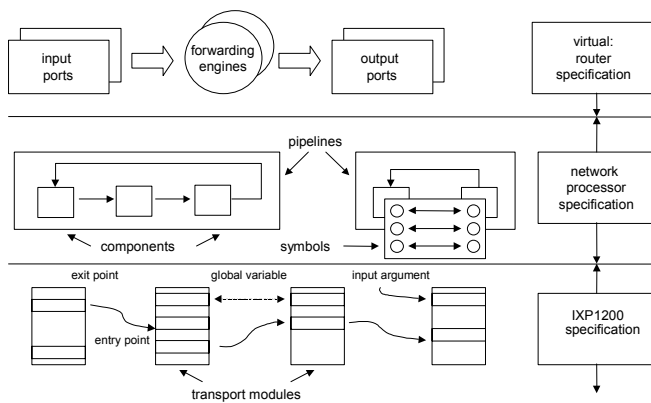
### 6) *Instruction Store Limitations*

Processor handoffs are caused by the fact that the instruction stores of a network processor have limited space. 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.

### 7) 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. A dynamic binding system should ideally support a wide range of packet processing applications ranging from, the creation of virtual networks over low time scales, to the fast creation of customized data paths, after a disaster occurs.



**Figure 3: Data Path Specification Hierarchy**

## III. NETBIND SYSTEM

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

### A. Design

#### 1) Data Path Specification Hierarchy

Before creating data paths, NetBind captures the structure and building blocks of data paths in a set of executable profiling scripts. Some profiling scripts can be 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 three different ways to profile data paths in NetBind. 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 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. Components are grouped into processing *pipelines*. A processing pipeline is a set of components that are executed by the same hardware contexts sequentially.

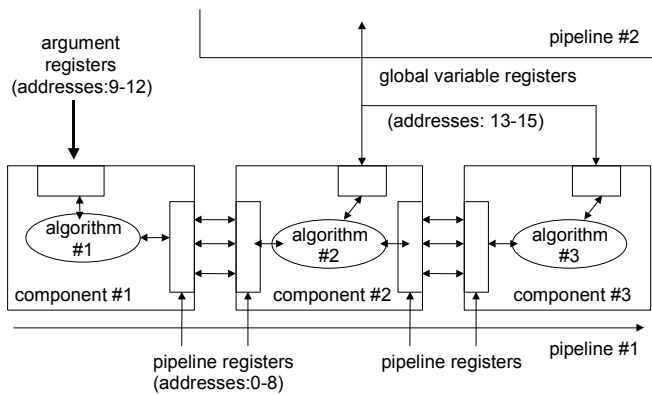
Components exchange packets between each other in a “push” or a “pull” manner. Components that sequentially exchange packets in a push manner can be grouped into the same pipeline. A data path is split between at least two pipelines if components perform different operations on packets simultaneously.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

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

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

Exit points are instruction store addresses of branch instructions. These branch instructions make the program

flow jump from one transport module to another. Entry points are instruction store addresses where the program flow jumps. Input arguments are instruction store addresses of “immed” IXP assembler instructions [2] that load GPRs with numeric values. These numeric values, (e.g., Cellular IP timers, IPv4 interface addresses), customize the operation of transport modules. Global variables are GPRs that are accessed using the absolute addressing mode. These GPRs hold numeric values that are shared across pipelines or data paths. For example, the SRAM address of a packet buffer in an IPv4 data path needs to be declared as global variable since this value is shared between the packet buffer and a scheduler.



**Figure 4: Register Allocation in NetBind**

## 2) Register Allocations

Register allocation realized in NetBind is shown in Figure 4. We observe that in data path implementations we have experimented with, registers are used in four ways. First, registers can hold numeric values used by a specific component. Each component implements an algorithm that operates on numeric values. When a component creates new values, it replaces old numeric values with the new ones in the appropriate microengine registers. We call these registers *pipeline registers*. The algorithm of each component places some numeric values into pipeline registers, which are used by the algorithm of the next component in the pipeline. In this manner pipeline registers are shared among all components of a pipeline. Pipeline registers are accessed using the context relative addressing mode. Once a component executes it becomes the owner of the entire set of pipeline registers. In this way, the registers used by different contexts are isolated.

Second, registers can hold input arguments. Input arguments are passed dynamically into components when pipelines are created from an external source, (e.g., the control unit of a virtual router [14]). Each component can place its own arguments into *input argument registers* overwriting the input arguments of the previous component. Similar to pipeline registers, input argument registers are accessed using context relative addressing. 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 absolute addressing.

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

In the current implementation of NetBind, we use eighteen 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). Each context contributes six GPRs in order to be used as global variable registers. In this manner, a pool of twenty-four global variable registers are shared among pipelines or data paths. Memory transfer registers are all pipeline registers. In our Spawning Networks Testbed, as part of the Genesis Project, we have used NetBind to build IPv4 [5] and Cellular IP [6] virtual routers. Static register allocation has proven sufficient for programming this set of diverse data paths, as discussed in Section III-C.

## 3) Binding Algorithm

Figure 5 illustrates 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, which are modified by NetBind, are illustrated as shaded boxes in the figure.

First, NetBind modifies the microwords that load input arguments into registers. Input argument values are specified using the NetBind programming API. Input argument values replace the initial numeric values used by the components. In the example of Figure 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 the addresses of these global variable registers before binding takes place. The binder then replaces the addresses that are initially used by the programmer for these registers, (i.e., 45 and 46 as shown in Figure 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).

Third, the binder modifies all branch instructions that are included in each transport module. The destination addresses of branch instructions are incremented by the instruction store addresses where transport modules are placed. Finally, the microwords that correspond to the exit points of transport modules are modified so that the program flow jumps into their associated entry points (step 4 in Figure 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 that we discuss in the evaluation section.

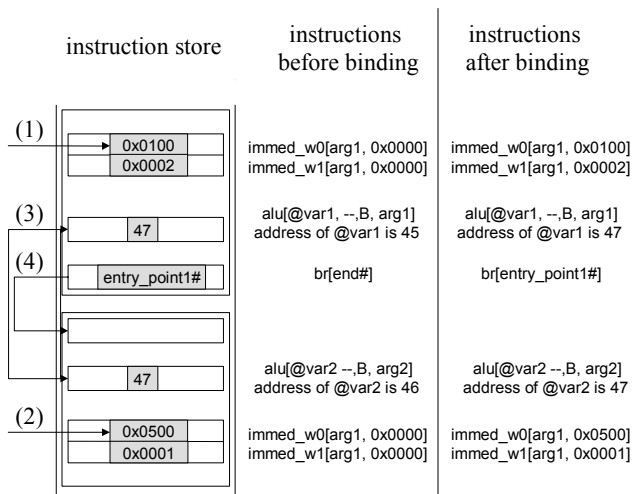


Figure 5: Dynamic Binding in NetBind

## B. Implementation

### 1) Binding System

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 6.

The NetBind binding system consists of:

- 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.
- 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*. Transport modules encapsulate component code and are accessed by the data path constructor.
- 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.

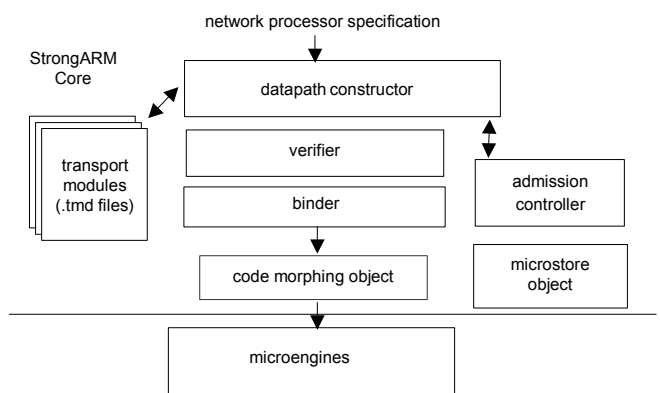
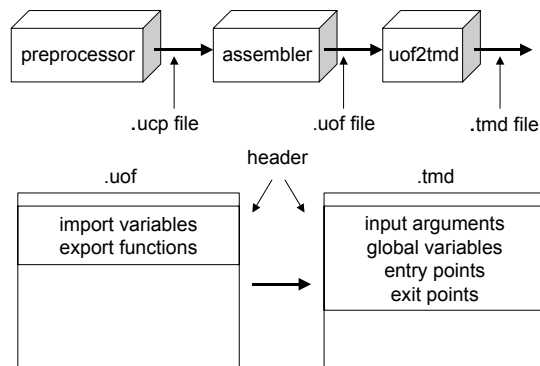


Figure 6: NetBind Binding System

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. The most difficult part in our implementation effort has been the realization of the

admission controller. In NetBind, admission control drives the assignment of system resources including registers, memory, instruction store space, headroom and hardware contexts. NetBind applies a “best fit” bin-packing algorithm to determine the most suitable microengines where data path components should be placed. In order to determine the remaining resources (i.e., leftover) 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 equally on the calculation of the leftover resources. Among different solutions that equally satisfy the resource requirements of a set of data paths, the one that results in less processor handoffs is selected.

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. 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 [18].



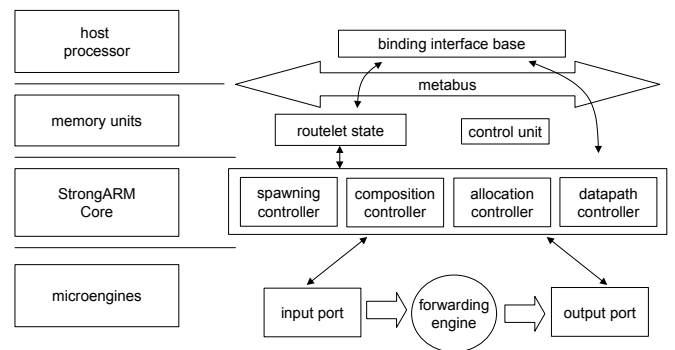
**Figure 7: Microassembler Extensions**

## 2) Microassembler Extensions

Transport modules can be developed using tools such as the Intel Developer Workbench [18]. The IXP1200 microassembler encapsulates the microcode associated with a component project into a “.uof” file. The UOF file format is an Intel proprietary format. The UOF 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 [2], (i.e., the Intel Developer Workbench,

the “transactor” simulation environment and the microassembler), we have created our own microassembler extensions on top of these Intel tools. Our microassembler extensions have been used for creating the transport modules of components.

The UOF file format is suitable for encapsulating statically compiled microcode, but does not include dynamic binding information in the header. For this reason, we introduced a new file format, the TMD (Transport MoDule) format for encapsulating microcode. Our microassembler extensions are shown in Figure 7. 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.



**Figure 8: Routelet**

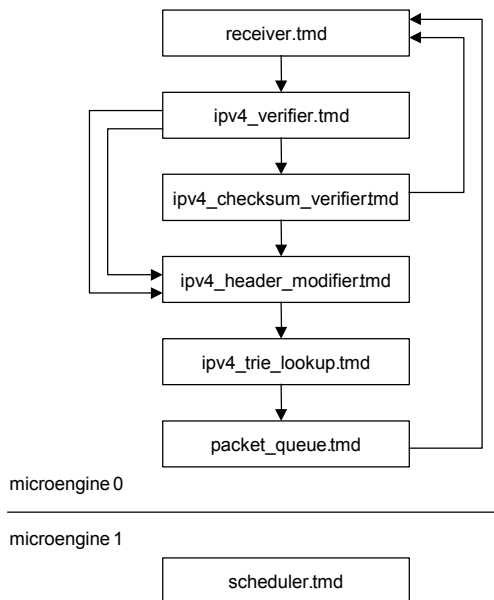
## C. Applications

### 1) Virtual Routers

We have used NetBind to create programmable virtual routers that seamlessly share the resources of the communications infrastructure. We consider the communications infrastructure as consisting of network processor-based programmable routers. A software model reflecting the structure, and building blocks, of a programmable virtual router is illustrated in Figure 8. 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 [16] 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 ports and forwarding engines, which comprise atomic transport modules. Transport modules perform simple operations on packets such as decrementing

the 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 8, 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.

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 by binding their associated software components into the network infrastructure at run time [14].



**Figure 9: An IPv4 data path**

## 2) IPv4 Data Path

The data path associated with an IPv4 virtual router implemented using NetBind is illustrated in Figure 9. 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;
- 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.

## IV. EVALUATION

### A. Experimental Environment

#### 1) Hardware 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 [17] 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.

#### 2) Software Environment

Initially NetBind was developed for an IXP1200 Ethernet evaluation board running at 166 MHz. To evaluate NetBind we ported the NetBind code to 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 complex system that can be used for programming microengines and the StrongARM Core. In what follows we provide an overview of MicroACE.

### 3) *MicroACE overview*

MicroACE is an extension of the ACE [17] 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 control plane counterpart of a microblock running in the microengines. The core component handles exception, control and management packets.

### 4) *Binding in MicroACE*

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.

### B. *Dynamic Binding Analysis*

#### 1) *Qualitative Analysis*

MicroACE and NetBind have some similarities in their design and realization but also differences. MicroACE offers much higher programming flexibility allowing 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, as discussed in Section III-A.

The main difference between NetBind and MicroACE in terms of performance comes from the choice of binding technique. 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.

#### 2) *Headroom Analysis*

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 any multi-threaded network processor architecture. We 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:

$$T = \frac{m}{C} \quad [\text{Eq. 1}]$$

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:

$$\frac{T}{n} = \frac{m}{n \cdot C} \quad [\text{Eq. 2}]$$

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  $\rho$  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:

$$H = \frac{\rho \cdot m}{n \cdot C \cdot t_c} \quad [\text{Eq. 3}]$$

The utilization factor was measured when our system was running the IPv4 data path discussed in Section III-C, and was found to be equal to 0.98. We doubled the percentage of idle time to have a worst case estimation of the utilization factor for many different types of data paths, resulting in  $\rho = 0.96$ . After substituting  $C = 100$  Mbps,  $m = 64$  bytes,  $n = 4$ , and  $t_c = 5$  ns into Eq. 3, we find that the headroom  $H$  in the IXP1200 Bridalveil system is 246 microengine cycles.

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 1: Binding Instructions in the Data Path**

module name	size	input arg.	global var.	entry points	exit points
receiver (initialization)	48	N/A	N/A	N/A	N/A
receiver	100	3	3	3	3
ipv4_verifier	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

**Table 2: Component Sizes and Symbols for the IPv4 Data Path**

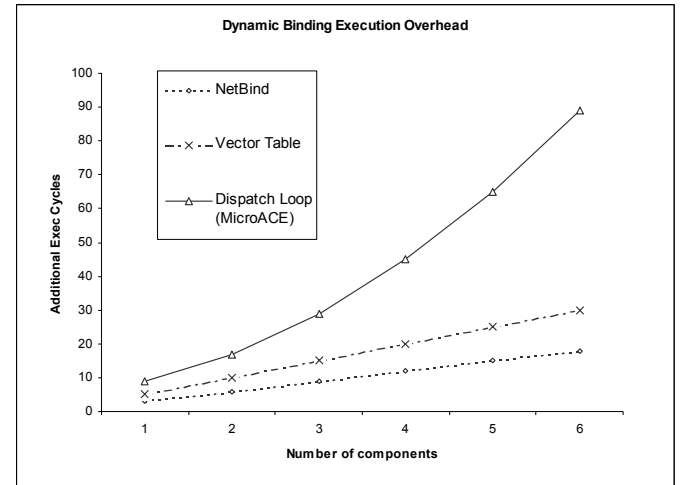
### 3) Binding Overhead Analysis

We evaluated the performance of the IPv4 data path discussed in Section III-C 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, as discussed in Section II. We used this tool in our experiments in order to compare the three binding techniques (presented in Section II) in a quantitative manner.

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 1 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 6 components (from "receiver" to "packet\_queue" in Table 2) 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 2 shows the number of instructions, input arguments, global variables, entry points, and exit points for each component of the receiver pipeline.



**Figure 10: Dynamic Binding Overhead**

We added a "worst case" generic dispatch loop to evaluate the MicroACE data path. The dispatch loop included 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 5 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 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 vector table scheme is heavily dependent on the speed of 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 5 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 10 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.

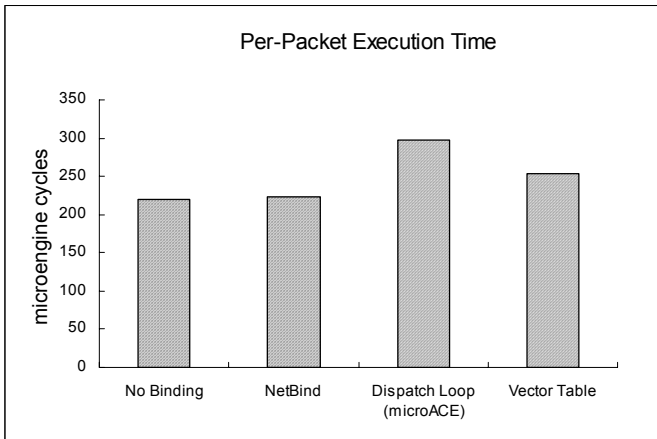


Figure 11: Per-Packet Execution Time

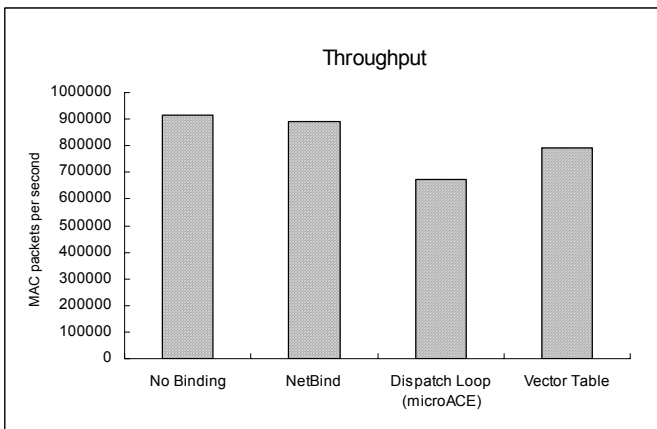


Figure 12: Packet Processing Throughput

Figures 11 and 12 show per-packet execution times and packet processing throughput for the three binding techniques, and a monolithic data path. The measurements were taken for the case where the data path is split between 6 components, and packets are forwarded at the maximum possible rate from four input ports to the same output port. 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.

#### 4) Binding Latency Analysis

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

Loading six modules from a remotely mounted NFS server takes about 60ms to complete. Since the IXP1200 network processor prevents access to its instruction stores while the hardware contexts are running, we had to stop the microengines before binding, and restart them again after the binding was complete. We are currently studying better techniques for dynamic placement without disruption to executing data paths.

## V. 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 [19] 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 [20]. The Synthesis Kernel aims for improving kernel routine performance in general-purpose processors as well.

## VI. CONCLUSION

We 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 ([comet.columbia.edu/genesis/netbind](http://comet.columbia.edu/genesis/netbind)) for experimentation.

#### ACKNOWLEDGEMENT

This research is supported by grants from the NSF CAREER Award ANI-9876299, and the Intel Research Council on "Signaling Engines for Programmable IXA Networks".

#### REFERENCES

- [1] Network Processing Forum, <http://www.npforum.org>
- [2] Intel IXP1200, <http://www.intel.com/IXA>
- [3] IBM Corporation, IBM PowerNP NP4GS3 Network Processor Datasheet, May 2001.
- [4] Intel Corporation, IXP1200 Network Processor Datasheet, Dec 2000.
- [5] Postel J., Editor, "Internet Protocol", *Request For Comments 791*, September 1981.
- [6] Campbell, A. T., Gormez, J., Kim, S., Valko, A., Wan, C., Turanyi, Z., "Design Implementation, and Evaluation of Cellular IP", *IEEE Personal Communications*, vol. 7 No. 4, pg. 42-49, Aug 2000
- [7] Intel Corporation, Intel IXA SDK ACE Programming Framework Developer's Guide, June 2001
- [8] Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M., "The Click Modular Router", *ACM Transactions on Computer Systems* 18(3), Aug 2000, pg 263-297.
- [9] Wolf T., Turner, J., "Design Issues for High-Performance Active Routers", *IEEE Journal on Selected Areas in Communications*, March 2001.
- [10] Taylor, D., Turner, J., Lockwood, J., "Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers", *IEEE Open Architectures and Network Programming*, April 2001.
- [11] Spalink, T., Karlin, S., Peterson, L., "Evaluating Network Processors in IP Forwarding", Technical Report TR-626-00, Nov 15, 2000
- [12] Spalink, T., Karlin, S., Peterson, L., Gottlieb, Y., "Building a Robust Software-Based Router Using Network Processors", In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pg. 216-229, Oct 2001
- [13] Karlin, S., Peterson, L., "VERA: An Extensive Router Architecture", In *Proceeding of the 4<sup>th</sup> International Conference on Open Architectures and Network Programming*, pg 3-14, April 2001
- [14] Kounavis, M. E., Campbell A. T., Chou, S., Modoux F., Vicente, J., and Zhang H., "The Genesis Kernel: A Programming System for Spawning Network Architectures", *IEEE Journal on Selected Areas in Communication*, Vol. 19, No 3, pg. 511-526, March 2001.
- [15] NetBind, <http://www.comet.columbia.edu/genesis/Netbind>
- [16] Genesis Project, <http://www.comet.columbia.edu/Genesis>
- [17] Intel Corporation, Intel IXA SDK ACE Programming Framework Reference, June 2001.
- [18] Intel Corporation, IXP1200 Network Processor Development Tools User's Guide, Dec 2000.
- [19] Montz, A., Mosberger, D., O'Malley, S., Peterson, L., and Proebsting, T., "Scout, A Communication Oriented Operating System", *Operating System Design and Implementation*, 1994.
- [20] Pu, C., Massalin, H., Ioannidis, J., "The Synthesis Kernel", Springer Verlag, 1988.
- [21] Campbell, A.T., Kounavis, M.E., Vicente, J., Villela, Miki, K. and De Meer, H., G., "A Survey of Programmable Networks", *ACM SIGCOMM Computer Communication Review*, Vol. 29, No. 2, pp. 7-24, April 1999.
- [22] Keshav S. "An Engineering Approach to Computer Networking", *Addison Wesley*, 1997
- [23] Bennett J. C. R., and Zhang H., "Hierarchical Packet Fair Queueing Algorithms", *IEEE/ACM Transactions on Networking*, 5(5):675-689, Oct 1997.
- [24] Stoica I., Zhang H., and Ng T. S. E., "A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service, in *Proceedings of SIGCOMM'97*, Cannes, France, 1997, pp. 249-262.