The Potential of Just-in-Time Compilation in Active Networks based on Network Processors

Andreas Kind, Roman Pletka, Burkhard Stiller

Abstract— **Byte-code representations in active networks provide architectural neutrality and code compactness; however, the resulting execution speed is typically poor due to interpretation overhead. This paper shows that the performance of capsule-based active networks can benefit from compiling active network programs into native network processor instructions at traversed routers (just-in-time compilation). A key aspect of the paper is to demonstrate that just-in-time compilers for active networks can be fast and small enough for applicability in the datapath of network processors. The approach has been implemented based on the SNAP active network framework for the PowerNP network processor.**

I. Introduction

ACTIVE networks break with the traditional networking paradigm in which information is passively forwarded from node to node. In packet-based active networks, routers perform customized computations on packets flowing through the network. The code for computations is either carried inside the packets (capsule approach) [1], [2] or is loaded out-of-band to routers according to code references inside packets (plug-in approach) [3]–[6]. Both approaches allow single packets (or flows of packets) to *actively* influence the forwarding process through the network. In this respect, active networks decouple networking services from the underlying infrastructure and provide new ways for flexible and customized service creation in packet-based networks [7].

To provide the advantages of architectural neutrality and code compactness, typical capsule-based active network systems represent programs in virtual machine code (byte-code). This introduction briefly discusses these advantages and focuses on the key drawback of the byte-code representation: execution performance.

Architectural neutrality is achieved by byte-code representation if an execution environment to emulate the corresponding virtual machine is provided wherever a byte-coded program is executed (i.e., on every router). All specific characteristics of the underlying router can be hidden by the virtual machine, so that an active network program should have the same semantics on every hop of a potentially heterogeneous network.

By representing common operations with a single byte-code, virtual instruction sets tailored to a specific application domain (e.g., active networking) yield significantly smaller programs than either non-specific virtual instruction sets or even native instruction sets [8]–[10].

Unfortunately, interpreting byte-coded active network programs poses serious performance problems [7]. Compared with traditional packet forwarding the processing effort increases and may jeopardize wire-speed forwarding at routers. A key requirement for the applicability of active networking in production networks is to limit the number of cycles spent for executing an active network program to the available cycle budget per packet. In backbone routers this may be no more than a few hundred cycles.

The reason for poor execution speeds of byte-coded active network programs is equivalent to the reason for poor execution speeds of byte-code interpretation in general. Each instruction has to be mapped (i.e., loaded, decoded, and invoked) by the interpreter. Interpretation times of byte-coded applications are thus normally more than ten times longer than execution of native machine code [11], [12]. Instruction mapping is necessary with native code too, but the processor hardware is able to perform the mapping much faster and in parallel with instruction execution.

This paper proposes to address performance problems in capsule-based active networks with just-in-time (JIT) compilation. Compared to compilation in advance (e.g., when the active packet is created), JIT compilation retains the safety properties of the byte-code language and maintains architectural neutrality in networks with heterogenous networking infrastructure. The technique has led to speed-ups with implementations of general-purpose programming languages such as Lisp, Smalltalk, Java, and Forth (See e.g., [13]). The insufficient performance of ordinary software-based routers prevents the application of JIT compilers in active networks. However, with the advent of network processors this idea becomes reasonable. The paper describes how the specific characteris-

Andreas Kind and Roman Pletka are with IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland. E-mail: {ank,rap}@zurich.ibm.com

Burkhard Stiller is professor at the Computer Engineering and Networks Laboratory (TIK), ETH Zürich, CH-8092 Zürich, Switzerland. E-mail: stiller@tik.ee.ethz.ch

tics of active network programs can actually facilitate a fast and lightweight compilation process that matches the strict processor and memory limits that exist when extending datapath functionalities using network processors.

We show that for a typical active network framework the number of processor cycles spent for compiling an individual instruction is only slightly larger than the number of cycles needed for interpretation. We measured that native execution of an individual machine instruction is, on average, over ten times faster than interpreting an equivalent byte-code from the virtual instruction set. From these results, it is clear that the performance benefit of JIT compilation increases the more often the processor executes a part of the program, or even the entire program, without recompilation. Reuse depends on the amount of recursion as well as looping in the programs and can be supported by caching of compiled code either at routers (e.g., reuse for packets of the same flow) or inside the packet. In the latter case, compiled code can be reused at each intermediate router that supports the same native instruction set. The idea of JIT compilation has been successfully tested with implementations of other languages for distributed programming [13], so that the main contribution of the paper is to show the applicability, potential benefits, and implementation approach of JIT compilation for active networks on network processors.

This paper describes the potential for JIT compilation for a general active network setup, using a dialect of the SNAP active networking language [14] running on a PowerNP network processor [15]. We show that JIT compilation is feasible in an active router because the compiler is small enough and fast enough to run in the data plane.

The paper is structured as follows: Section II positions our approach with respect to other work on active networks concerned with improving performance. Section III presents the active network framework we use for our JIT compilation approach and describes the active network language. A key feature of this language is the capability to define resource bounds while allowing loops. Section IV describes the JIT compilation approach. A quantitative comparison of interpretation versus compilation of active network programs is given in Section V. The paper draws conclusions in Section VI.

II. Related Work

The objective to improve performance of active networks is common to many active network frameworks. Whereas some related work gives flexibility priority over performance concerns, e.g., by providing high-level programming environments, others trade flexibility for better performance.

PLAN [16] is a packet language for active networks that does not require authentication and still executes safely. The language is simple in order to achieve reasonable performance. PLAN programs share state through service routines at routers and enable the creation of new protocols without encapsulation. The drawback of PLAN is, however, unrestricted resource usage. It can be shown that PLAN programs exist that execute in time exponential to packet size.

Moore *et al.* [14] balance the tradeoffs between flexibility, efficiency, and safety with *SNAP* (Safe Networking with Active Packets). SNAP, a stack-based active networking language, evolved from PLAN and is safe with respect to network resource usage (resource conservation) and evaluation isolation. The execution of a SNAP program can only consume bandwidth, CPU, and memory resources up to a limit linear to the packet's length. To achieve this goal, only forward branches are allowed, i.e., loops and recursions are prohibited. Furthermore, packets may only stay at nodes for a limited amount of time. This balance of features distinguishes SNAP from other active networking approaches that are either restricted to the control plane, have unacceptably low performance, or sacrifice safety.

The *Smart Packets* approach for active networking introduced by Schwartz *et al.* [17] focuses on network management and monitoring. To simplify management and consistency of active code, routers do not maintain state across packets. The packet-transport service is connectionless and smart packets must be self-contained. Therefore, programs have to be smaller than 1 KB. Security concerns restrict the active code to be executed within a sandbox environment. A C-like language called Sprocket is compiled into a machine-independent assembler, which in turn is assembled into byte-code for a virtual machine. Extending services dynamically is not feasible with Sprocket as it would require modifying the virtual machine itself. The security architecture consists of a maximum number of instructions to be executed, limited memory usage, and restricted access to the management information base.

A route taken by Decasper *et al.* [3] is to address the demand for high-performance active networks with a set of hardware design measures. The resulting *Active Networking Node* (ANN) is built with processing engines (CPU + FPGA) on each port of a switch backplane and is combined with a suitable operating system (NodeOS) and software infrastructure. Active networking instructions are executed in an execution environment on top of the NodeOS.

JIT compilation has been used with the *Liquid Software* mobile code approach [18]. The initial idea with compilation in this system was to start compilation while still receiving the remaining part of a program. The compiler translates Java byte-code into native instructions (e.g., P5, SPARC). The Liquid Software compiler has also been used to construct an active network node using the ANTS [19] framework. The base ANTS system is statically linked as native code into the active node. The compiler is then able to translate Java byte-code carried in capsules into native code. ANTS, based on the Liquid Software approach, showed clear performance improvements over the standard Java-based implementation.

Plezbert and Cytron [20] propose to interleave compilation, interpretation, and native execution of program units in mobile code systems. They show that this approach can outperform the speed-up achieved with standard JIT compilation. Their idea derives from the fact that the overall execution speed of a program does not necessarily benefit from compiling all program units. It makes more sense to compile only those program units for which the relation between compile time and interpretation time indicate a potential speed-up.

The work described in this paper uses SNAP as a starting point and tries to improve its performance and enhance its capabilities. In Section III a dialect of SNAP is defined that allows controlled program loops in active network programs. With SNAP, we try to base our work on an existing active networking framework, so that results will not depend on specific characteristics of a newly created active network framework.

Related work on JIT compilation in active networks mentioned above is similar to our work. However, this paper focuses only on capsule-oriented active network frameworks in the sense that each packet carries its own program, which typically is far from being, for instance, a full-fledged Java program. Moreover, the target execution hardware in our case is a network processor with a dedicated instruction set and additional specialized coprocessors designed specifically for packet processing. To our knowledge, no work has been published so far that investigates the possibilities of JIT compilation for active networks based on network processors.

III. Active Networking Framework

This section presents the active networking framework used in this paper. The framework follows the capsule approach by using active packets that carry data and code to be executed on each node along the path. The framework is based on a dialect of SNAP, but allows program loops under strict resource-bound conditions.

The general model of a router based on network processors as used in this paper is shown in Figure 1. Forwarding decisions are usually taken on the ingress side, whereas the egress network processor is mainly involved in scheduling packet transmission. The processing of an active packet can be done on the ingress as well as the egress side, so that two entry points in the active code have to be maintained in the packet. If the router has a centralized processing unit, as for example on a software-based Linux router, the egress code has to be placed directly following the ingress code segment. Such a router can then ignore the egress entry point, and execute all active code beginning at the ingress entry point. Execution will automatically fall through from ingress to egress code section.

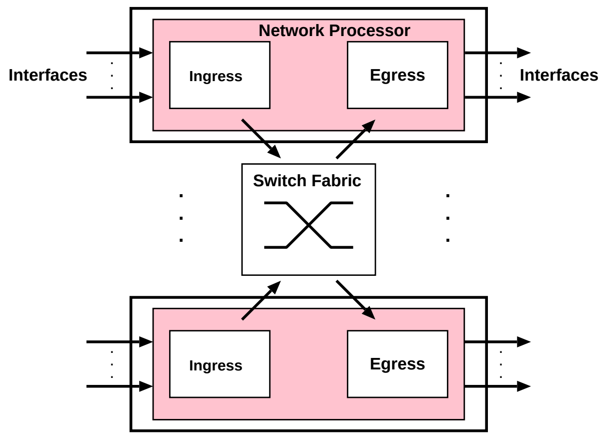

Fig. 1. Router model with network processors.

To simplify access to active packet data[1], the memory section consisting of heap and stack is located between the packet header and the active code section (see Figure 2). The memory section has a maximum size of 128 bytes.

A. Instruction Set

Unlike general-purpose processors, network processors often have hardware support for parallel packet processing, tree lookups, checksum generation, scheduling, counting, etc. In an active networking context, these hardware assists should be made available to active programs in order to accelerate the execution of active code.

As different network processor have different hardware assists, it is the task of the active networking sandbox (ANSB) to hide this complexity and to provide the functionality by other means (e.g., general-purpose processor) if not present. The absence of hardware support has a direct impact on the instruction cost as will be explained later on.

Our active networking framework uses an extension of the SNAP [14] instruction set shown in Table I. Core service instructions have been added to cover frequently used

[1]Usually, network processors break packets into separate cells [15], [21] when a packet is received.

TABLE I

INSTRUCTION CLASSES.

Instruction Class	SNAP	SNAP Dialect (Modifications and Extensions)
Stack control	`EXIT, PUSH, POP, POPI, PULL`	`SWAP`
Flow control	`PAJ, JI, BEZ, BNE`	Backward branches allowed
Heap operations	`MKTUP, NTH`	Replaced by `RCL, STO`
Relational operators	`EQ, EQI, NEQ, NEQI, GT, LT, GEQ, LEQ`	
Arithmetic operators	`ADD, ADDI, SUB, MOD, NEG, NOT, AND, OR, ORI, XOR, LSHL, RSHL`	
Operation on addresses	`SNET, BCAST`	
Core services	`GETSRC, GETDST, FORW, SEND, HERE, ISHERE, GETRB, ISNEIGH`	`GETITF, GETNQ, GETENTRY, SETENTRY, GETCS, GETLRB`
Router services	`CALL, RREQ`	

router services, such as accessing the interfaces, queuing classes, and congestion status. Other changes have been made to simplify heap operations (`STO` and `RCL`). A frequently used stack operation (`SWAP`) has been added. `GETLRB` delivers the local resource bound, as will be explained later. `GETENTRY` and `SETENTRY` allow the manipulation of the ingress and egress entry points. Heap and stack elements are 32 bits in size. Instructions are encoded with 16 bits (seven bits for the opcode and nine bits for immediate arguments).

B. *Resource Bound*

To prevent a denial-of-service attack, SNAP uses several restrictions to limit resource utilization of active packets in the network. These restrictions consist of a resource bound derived from the time-to-live (TTL) field and the fact that SNAP programs use bandwidth, CPU, and memory resources in linear proportion to the packet's length [22]. Hence, byte-code instructions must execute in constant and predictable time.

The resource bound as defined in SNAP turns out to be difficult to respect because, as will be shown later in this paper, the byte-code instructions differ significantly with regard to their execution time. In addition, the limitation that only forward branches are allowed is too restrictive for real programs (e.g., the congested hop counter example in Section III-D would not be feasible). Therefore, a new definition of the resource bound is introduced here.

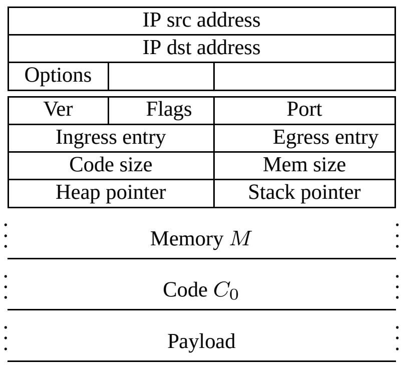

Fig. 2. Active packet including parts of the IP header. Ingress and egress entry point to the corresponding starting points in the code section C_0. Heap and stack pointer point to the current positions in the memory section M.

Consider two packets with the same packet length. One of the packets has a large code section and no payload, the other packet has a large memory and payload section. The packet with a large code section should get the same resources for executing active code as the packet with a large memory and payload section. In other words, as processing a packet p takes at most $O(|p|)$ time ($|p|$ denotes the packet length), payload and memory section can be accounted in the packet's resource budget in the same way as code sections. The idea is to use this extra budget for program loops.

We define a packet p as $p = \langle r, C_0, C, M \rangle$ using the notation borrowed from [22], where r is the network part of the resource bound, C_0 the full program code, C an arbitrary sequence of instructions within C_0, and M the memory size containing heap and stack of the packet.[2] The notation $\longrightarrow_{local}$ describes the execution of an active packet on a given node ; $\longrightarrow^{j}$ is a sequence of j reductions. $|C|$ denotes the number of dynamically executed byte-code instructions of the sequence C. The SNAP reduction for CPU safety

$$\langle r, C_0, C, M \rangle \longrightarrow^{j}_{local} \langle r, C_0, \emptyset, M' \rangle \quad j \leq |C_0|, \tag{1}$$

no longer holds in the presence of loops because $|C|$ might be larger than $|C_0|$. Further on, byte-code instructions do not execute in constant time. Therefore, we define $\hat{C}$ as the sequence of machine dependent CPU cycles corresponding to the byte-code sequence C, and denote $|\hat{C}|$ as the number of machine dependent CPU cycles thereof.

[2]For simplicity the notation of a packet does not contain all parts of a packet (e.g., packet header is not represented). Only the relevant parts are included.

In the following, we show that our extension of the SNAP resource concept does not violate safety on CPU, memory, and bandwidth usage. The new resource bound, a two dimensional vector[3], consists of a local part $n|p|$, which is linear to the packet size $|p|$, and a network part r proportional to the TTL of the packet (n is a constant of proportionality and is measured in instructions per byte). While the conditions on the network part remain unchanged, the resource bound for the local part has to be redefined. The new definition of the local resource bound is based on a maximum instruction costs w_i (derived from the real execution times) associated with each instruction i in the byte-code instruction set I.

For an active packet with data D (where D is the entire packet less the code section C_0), the local resource bound is proportional to the sum of the sizes of code section $|C_0|$ and data $|D|$. This sum has to be greater than the sum of all maximum instruction costs w_i (in native cycles for the byte-code $i \in I$ independent of any program) executed for the packet in total. Thus, CPU safety holds under the following condition

$$n|p| = n|C_0| + n|D| \geq n|C| = m|\hat{C}| = m \sum_{k=1}^{|C|} w_{C[k]} \, , \tag{2}$$

where $C[k]$ is the k-th instruction in the execution sequence. Note that $C[k] \in I \quad \forall k : 1 \leq k \leq |C|$. The factor m is determined by the average instruction cost and the maximum number of processor cycles the ANSB can spend on executing active code for a given packet.

We are now going to prove Equation (2). Let $|\hat{i}_b|$ be the number of native (single-cycle) instructions needed to interpret byte-code $i \in I$ given that the machine state is such that the interpreter will take branch b out of the set of all possible branches B_i for that opcode. During initialization time, all w_i have been chosen to satisfy the following equation,

$$\max_{b \in B_i} |\hat{i}_b| \leq w_i \qquad \forall i \in I \tag{3}$$

by taking the longest branch possible. $|\hat{C}[k|b]|$ is the number of native instructions in the sequence $\hat{C}[k]$ given that the machine state is such that the interpreter will take branch b out of the set of possible branches $B_{C[k]}$. Equation (3) holds for every instruction i defined in the byte-code language and is independent of any code sequence C:

$$\max_{b \in B_{C[k]}} |\hat{C}[k|b]| \leq w_{C[k]} \qquad \forall k : 1 \leq k \leq |C| \quad . \tag{4}$$

By induction on $\longrightarrow_{local}$ each instruction i decreases the available local resources by w_i. Therefore the sum over an arbitrary sequence of instructions is limited by the resource bound given in Equation (2):

$$m \cdot \sum_{k=1}^{|C|} \max_{b \in B_{C[k]}} \left| \hat{C}[\, k \,|\, b \,] \right| \leq m \cdot \sum_{k=1}^{|C|} w_{C[k]} \leq n|p| \tag{5}$$

$\square$

The less restrictive bound on memory usage defined in SNAP remains unchanged for the following two reasons: First, all byte-code instructions can push at most one element on the stack or add at most one element to the heap. Second, using (2), the maximum number of instructions that can be executed is bounded and the maximum growth of the stack does not depend on the presence of loops. Memory safety as well as processing isolation remain unchanged.

C. *Sandbox Environment*

The active networking framework supports services that can be requested from the ANSB. This is very similar to the plug-in approach [3] but is used here to offer customized networking services that are time-constrained and that do not allow the execution of arbitrary code segments (e.g., IP address lookup, QoS parameter lookup).

The ANSB provides safe execution of active code because memory access and resource utilization are strictly controlled. For example, full read/write access is only granted for the memory section of the packet, whereas read access is only given for the packet header. Other data structures are only accessible using core or router services. Thus active network code is not allowed to write data at any location in the packet or ANSB environment.

The ANSB has two separate tasks. The first task is to control the safe execution of active code of an incoming active packet. The second task is a background task to maintain tables. An example of such a table is the congestion status table that keeps track of the occupancy of the queues on a given output interface. Depending on the underlying hardware, the ANSB can run either in the control point or directly on a network processor. The latter has been implemented and used on a network processor for this paper.

D. *Examples*

Active code can vary from simple arithmetics without loops to complex lookup iterations. Two complementary examples for active code are given below and are later used in Section V for illustration and comparison. The first example is a loop-free active program that is able to send a new packet based on the current packet. The second example invokes core router services in a loop. The services

[3]The resource bound is defined as a vector because local and network resources are non-exchangeable for safety reasons.

are likely to be costly because of tree lookups that have to be performed.

D.1 Traceroute

The program shown below demonstrates how a simple traceroute program can be implemented. The actual hop count and the initial TTL are kept in the packet's heap. The hop count is first loaded onto the stack (`RCL`), incremented by one (`ADDI`), and then stored (`STO`). `HERE` loads the IP address of the current router onto the stack. A second packet is then created and sent back (`SEND)` to the source containing all the active router IP addresses traversed so far. The `SEND` instruction is, however, expensive because it involves creating a new packet. If the packet reaches the destination it forwards itself back to the source (`FORWTO`) and modifies the ingress and egress entry points (`SETENTRY`) to stop collecting hop information on the backward path.

```
L1   RCL  hop_count ; Load heap element onto stack
L2   ADDI 1         ; Increment hop count
L3   STO  hop_count ; Store the new value
L4   HERE           ; Push current hop ip@ onto stack
L5   GETDST         ; Destination ip@ from header
L6   ISHERE         ; Test if destination reached
L7   BNZ  L20       ; Branch to L20 if dest reached
L8   GETENTRY       ; Prepare stack for SEND.
L9   PUSH L19       ; Ingress entry offset to L19
L10  LSHL 16        ; Shift left by 16 bits
L11  ORI  L19       ; Egress entry offset to L19
L12  ADD            ; Add to actual entries
L13  RCL  hop_count ; Load stack depth of new packet
L14  RCL  init_ttl  ; Load initial TTL
L15  GETRB          ; Get actual TTL
L16  SUB            ; Exact amount of resources used
                    ; up to now that should be given
                    ; to the new packet.
L17  GETSRC         ; Get sender IP address
L18  SEND           ; Send a new packet back
L19  FORW           ; Stop execution and forward
                    ; the packet
L20  PUSH L19       ; New ingress entry at L19
L21  LSHL 16
L22  PUSH L19       ; New egress entry at L19
L23  ADD            ; Merge the entry points
L24  SETENTRY       ; Adapt entry points for
                    ; return path
L25  GETSRC
L26  FORWTO         ; And send the packet back
```

D.2 Congested Hop Counter

The congested hop counter collects the number of congested queues in active routers along the path through the network and stores this information on the stack together with the router's IP address. In contrast to the previous example, this program uses a loop (`JI`) to accumulate the data and performs tree lookups in the loop to get the number of queues on the outgoing interface (`GETNQ`) and the congestion status (`GETCS`) of these queues. `GETCS` takes the IP address of the interface and the queue index from, and pushes the congestion status onto the stack. The congestion status is a two-bit value ranging from zero (not congested) to three (heavy congestion). The queue is counted as congested if this value is equal or larger than two by doing a right shift on the congestion status (`RSHLI`). Congestion status information is collected over three hops, therefore the program needs a memory section of at least 40 bytes.

```
L1   HERE            ; Current hop IP@ onto stack
L2   PUSH  0
L3   RCL   hop_count ; Load hop count onto stack
L4   ADDI  1         ; Increment hop count
L5   PULL  0         ; Duplicate top stack value
L6   RSHLI 2         ; Divide by 4
L7   BNZ   L28       ; Stop at 4th hop
L8   STO   hop_count ; Store new hop count
L9   GETNQ           ; Get number of queues
                     ; for this interface
L10  GEQI  0x8       ; 8 is max loop counter
L11  BNZ   L14       ; Are there more than 8 queues
L12  GETNQ           ; No
L13  JI    L15
L14  PUSH  0x7       ; Yes, take the max
L15  PULL  0         ; Duplicate top stack value
L16  BEZ   L28
L17  PULL  0         ; Duplicate top stack value
L18  HERE
L19  SWAP
L20  GETCS           ; Get congestion status for
                     ; this queue
L21  RSHLI 1         ; Take the higher cs bits
L22  BEZ   L26       ; Do not count if not congested
L23  SWAP
L24  ADDI  1         ; Increment number of congested
                     ; interface
L25  SWAP
L26  SUBI  1         ; Decrement loop counter
L27  JI    L15       ; End of loop
L28  FORW
```

IV. Just-in-Time Compiler

This section describes a general approach to compiling active network programs into native instructions of target network processors.

A. Target Network Processor

The objective of just-in-time (JIT) compilation is to translate an active network program into a semantically equivalent program for a given target network processor just before its execution. Typically, the active network program is represented in a stack-oriented byte-code instruction format, and the output of the compiler is a sequence of native network processor instructions (Figure 3).

A network processor is a processor that is designed for fast *and* flexible packet processing compared to other solutions, like Application Specific Integrated Circuits or general purpose processors. It is typically based on an em-

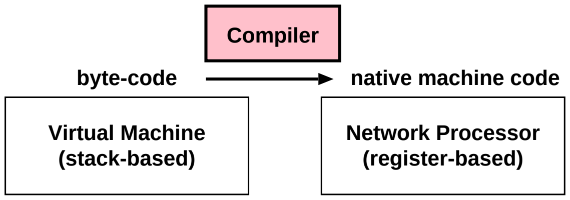
Compiler
byte-code
native machine code
Virtual Machine (stack-based)
Network Processor (register-based)

Fig. 3. Compilation

bedded processor complex for parallel packet handling including co-processors. The generic network processor architecture assumed here is inspired by the IBM PowerNP network processor but is sufficient generic to cover also other network processor architectures [21], [23]. The minimum characteristics of this architecture are as follows:

- 15 scalar word registers (`w0` to `w14`).
- An array register[4] initialized with at least the first 64 bytes of the packet being forwarded (`ARRAY_PKT`).
- A mechanism to read more data from the packet into `ARRAY_PKT`.
- A mechanism to update the packet being forwarded with data from `ARRAY_PKT`.
- A second array register of at least 64 bytes for temporary values (`ARRAY_TMP`).
- Write access to instruction memory (`INSTRMEM`).
- Load and store operations to move data between registers.
- Standard arithmetic and logical operations on scalar registers.
- Support for standard comparison.
- Standard control flow operations (e.g., branches, subroutine calls).

Instruction memory access and high memory bandwidth, which is required by the given network processor model, is currently not supported by some network processors today [24].

For performance reasons, we assume compilation cannot be performed on a control processor attached to the network processor when active packets have to be handled at wire-speed in the data path. Such a solution would simplify the implementation of the compiler because control processors are typically equipped with a standard general-purpose processor, a multitasking operating system, and convenient development tools. Therefore, the compiler itself has to be implemented in the native network processor instruction set and has to meet the demands of using only few cycles and showing a small memory footprint.

[4] Array registers represent data as a collection of bytes.

B. *Stack-Based to Register-Based Code*

As, in general, network processors are register-based, a JIT compiler for active networks must translate the, typically, *stack-oriented* operation of byte-code instructions to *register-oriented* operations of native network processor instructions. This requirement does not necessarily increase the complexity of the compiler. As the stack inside an active packet cannot grow very large, all stack positions can be mapped directly onto a subset of the available network processor registers. This mapping from relative stack positions to absolute registers can, however, be determined only during compile-time if it is possible to associate a fixed stack depth with each source instruction. In other words, an instruction in a source program must not be executed at different stack levels (e.g., as it would be the case with true recursion).

Some sample code fragments illustrate this case. The first instruction (`GETCS`) in the following code fragment could potentially be executed several times at different stack locations. It is therefore impossible to associate a fixed register with the load operation.

```
GETCS  ; Load congestion status
DUP    ; Duplicate top of stack
BEZ -2 ; Branch if zero
```

However, in the next code fragment the branch continues at the same stack level and the compiler can translate the memory load operation into a native load register operation.

```
RCL 0  ; Load congestion status
BEZ -1 ; Branch if zero
```

Admitting only this kind of recursion (also known as tail-recursion) in combination with a maximum stack size limited by the number of available native registers, allows simplifying the JIT compiler while still retaining the possibility of handling `while` and `for` loops. Complex register-allocation schemes, including register flushing into memory, that would complicate the compiler are not necessary. Active packets that do not follow this requirement are either treated as being standard non-active packets or handled by the interpreter. The latter case requires that both, interpreter and compiler, are installed on the network processor.

C. *One-Pass Compilation*

The source program representation inside active packets is linearized and in binary format, so that the compilation process does not include costly compilation phases, such as tokenizing, syntax/semantic analysis, etc. In fact, it can be performed in one pass:

1. Initialize byte-code vector pointer.
2. Load byte-code and inlined argument.
3. Store stack level of current source instruction.
4. Jump to instruction code.
5. Check stack level and inlined argument ranges.
6. Construct native instruction code.
7. Write native instruction code into instruction memory.
8. Increment byte-code vector pointer.
9. Go to step 2, unless end of byte-code vector reached.

This operation is very similar to an interpreter loop, except that native instructions are not executed but written to instruction memory. The compiler assembles one or more four-byte words that implement the source instruction in the register-based stack operation.

After the last instruction of the active network program has been translated and stored, registers are initialized according to the stack values in the packet, and execution continues at the entry point of the native program located in instruction memory. When the program has finished, the stack inside the packet is updated according to the stack values, and normal forwarding in the network processor continues.

D. Compiling the Examples

To illustrate the compiler, the two examples of Section III are discussed here. The output is given in a simple assembler language for the sake of human readability. The following listing shows the traceroute example compiled into assembler:

```
L1   ldr  w5, ARRAY_TMP[0]
L2   add  w5, #1
L3   str  ARRAY_TMP[0], w5
L4   ldr  w5, w2
L5   ldr  w6, ARRAY_PKT[IPv4Header.dstAddr]
L6   cmp  w2, w6
L7   jeq  L20
L8   ldr  w6, ARRAY_PKT[ANHeader.entryPoint]
L9   ldr  w7, L19
L10  sll  w7, #16
L11  or   w7, L19
L12  add  w6, w7
L13  ldr  w7, ARRAY_TMP[0]
L14  ldr  w8, ARRAY_TMP[1]
L15  ldr  w9, ARRAY_PKT[IPv4Header.ttl]
L16  sub  w8, w9
L17  ldr  w9, ARRAY_PKT[IPv4Header.srcAddr]
L18  call anSend
L19  ret
L20  ldr  w6, L19
L21  sll  w6, #16
L22  ldr  w7, L19
L23  add  w6, w7
L24  str  ARRAY_PKT[ANHeader.entryPoint], w7
L25  ldr  w6, ARRAY_PKT[IPv4Header.srcAddr]
L26  j    anFwdTo
```

Most active network instructions can be directly translated into a single assembler instruction. Line labels indicate the source code line number from Section III-D. Complex instructions are provided by the sandbox environment (e.g., SEND, FORWTO, and GETCS). For some instructions a subroutine call is performed (e.g., at label L18). The top of stack starts in register w5. Register w0 to w4 hold special values provided by the sandbox (e.g., w2 contains the local IP address). ARRAY_TMP holds the active network memory, i.e., heap and initial stack. The first 64 bytes of the packet and the active packet memory can be accessed through ARRAY_PKT.

The active code example to count the number of traversed congested hops contains a loop through the existing queues at the outgoing router interface. The compiled assembler code looks as follows:

```
L1    ldr  w5, w2
L2    ldr  w6, #0
L3    ldr  w7, ARRAY_TMP[0]
L4    add  w7, #1
L5    ldr  w8, w7
L6    slr  w8, #2
L7    cmp  w8, #0
      jne  L28
L8    str  ARRAY_TMP[0], w7
L9    call anGetNQ
L10   cmp  w7, #0x8
L11   jnz  L14
L12   call anGetNQ
L13   j    L15
L14   ldr  w7, #7
L15   ldr  w8, w7
L16   cmp  w8, #0
      jeq  L28
L17   ldr  w8, w7
L18   ldr  w9, w2
L19   ldr  w10, w9
      ldr  w9, w8
      ldr  w8, w10
L20   call anGetCS
L21   slr  w8, #1
L22   cmp  w8, #0
      jeq  L26
L23   ldr  w8, w7
      ldr  w7, w6
      ldr  w6, w8
L24   add  w7, #1
L25   ldr  w8, w7
      ldr  w7, w6
      ldr  w6, w8
L26   sub  w7, #1
L27   j    L15
L28   ret
```

Special subroutine calls are required for GETNQ (getting number of queues) and GETCS (getting queue congestion status). It can be seen that not all source instructions can be compiled into a single native instruction. For example, the SWAP operation has to be represented by three native load instructions (see labels L23 and L25).

V. Interpretation *vs.* Compilation

As explained in the preceding section, JIT compilation for active networks can be performed in only one pass through the source byte-code vector. This simple operation implies that a compiler need not take significantly more cycles for compiling a program than interpretation would take for interpreting all byte-code instructions in the program exactly once. This section tries to support this assumption and shows that with loops and native code caching, JIT compilation operated in the data-path of a network processor is feasible and can lead to performance advantages.

The approach described in the Section IV has been implemented for the IBM PowerNP 4GS3 [15] network processor. The PowerNP network processor has an embedded processor complex that consists of 16 *Pico Processors*, multiple specialized co-processors, and a PowerPC microprocessor. *Picocode* is the native Pico Processor instruction set. The picocode instruction set is specifically designed for packet processing and forwarding. With the support of co-processors (e.g., for table lookup or checksum computation) up to 32 packets can be processed in parallel (i.e., aggregate 2128 MIPs). The PowerNP reference platform used for testing is capable of processing 40 Fast Ethernet or 4 GBit Ethernet ports in wire-speed. The total PowerNP picocode instruction memory is 128 KB. Owing to the simplicity of the compiler, it actually is small enough (12 KB) to fit into the native picocode instruction memory, and leaves enough space for regular IP forwarding functionality.

A. Comparing Single Instructions

The cost of compiling active programs is the key factor that determines whether JIT compilation is feasible for active networks at all. A performance benefit is achieved if interpretation time T_I exceeds the sum of compilation time T_C and native execution time T_E:

$$T_I > T_C + T_E \qquad (6)$$

Because the compiler performs a fixed amount of cycles for each instruction, the compilation time increases linearly with the program size, whereas interpretation and native execution times depend on control operations and the given resource bound.

In order to identify the principal compilation and execution costs that allows the instruction costs w_i to be determined, the number of cycles spent for compiling, interpreting, and executing has been measured for each instruction (see Figure 4). With 1139 cycles, the cost of executing or interpreting the SEND operation is significantly higher

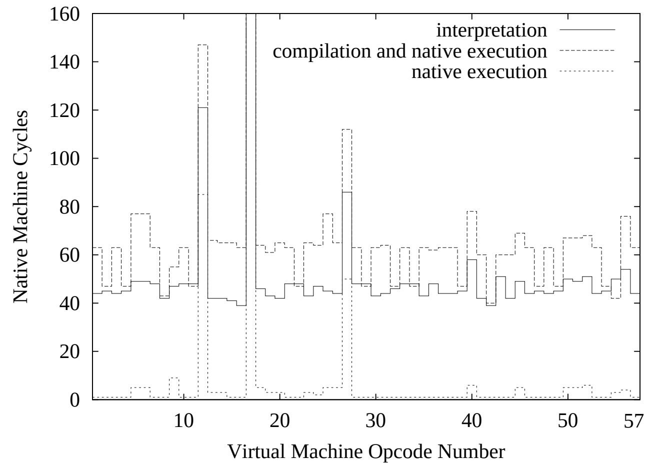

Fig. 4. Costs of interpretation, compilation, and native execution of individual source instructions.

than the average execution/interpretation cost of all other instructions. This is due to the fact that network processors are not optimized for such specific active networking tasks. Multicast, which is in some ways similar to the SEND instruction, can be treated in network processors by simply creating additional packet headers that are linked to the original payload. However, the SEND instruction not only builds a new header, but also reuses parts of the code, stack, and heap. Therefore, the new packet has to be created from scratch at high cost. Without the SEND operation, 4.4 cycles for native execution, 47.7 cycles for interpretation, and 57.4 cycles for compilation are spent on average. Thus, by looking only at the average cost of individual instructions, interpretation is slightly more expensive than compilation. However, native execution is about ten times faster as interpretation.

To illustrate the different costs shown in Figure 4, the implementation of the ADD operation is presented in assembler code below. With interpretation two parameters are loaded from memory, stack underflow is checked, and the result of the native add is stored to the top-of-stack position in memory:

```
ldr  w14, ARRAY_PKT[w1]  ; Get top of stack value
add  w1, #4              ; Adjust top of stack
cmp  w1, w0              ; Stack underflow?
jge  stack_underflow
ldr  w13, ARRAY_PKT[w1]  ; Get top of stack value
add  w14, w13            ; Perform addition
str  ARRAY_PKT[w1], w14  ; Put result on stack
```

With compilation, first the current stack depth has to be stored for checking tail recursion. Then stack underflow is tested, and the native operation is constructed. Finally, the operation is written to instruction memory:

```
str  ARRAY_TMP[w1], w3  ; Store stack depth
cmp  w3, #1             ; Stack underflow?
```

```
bl   stack_underflow
ldr  w14, #0xC4700      ; Construct native instr
or   w14, w1             ;   C47<reg1><reg2>000
sll  w14, #8
sub  w1, #1
or   w14, w1
sll  w14, #12
str  INSTRMEM[w2], w30  ; Write native instr
add  w2, #0x10          ; Inc instr mem addr
```

The native execution basically performs only the following statement:

```
add  w5, w6
```

For interpretation and compilation, a constant overhead for mapping the source instruction is actually part of the total cost. In Figure 4, this overhead of about 36 cycles has already been included.

B. *Execution Cost of Examples*

So far we have looked at costs of interpretation, compilation, and native execution of individual source instructions. In Figure 5 we now compare cycle cost for interpretation with cycle cost for compilation and native execution for the example programs mentioned above. We measured cycle costs using the PowerNP picocode profiler tool.

When an active packet is interpreted, the ANSB first has to check the consistency of the packet, load the memory into a temporary array register, and fetch the first instruction block. This is done during initialization (Init). Then the ANSB starts the main interpretation loop. At each iteration the current instruction is decoded, the program counter as well as the resource bound are updated, and the corresponding instruction code is located (ANSB). The instruction code is then executed (Instr). Finally stack and heap are written back and the active networking header is updated (Clean) before any further packet processing occurs.

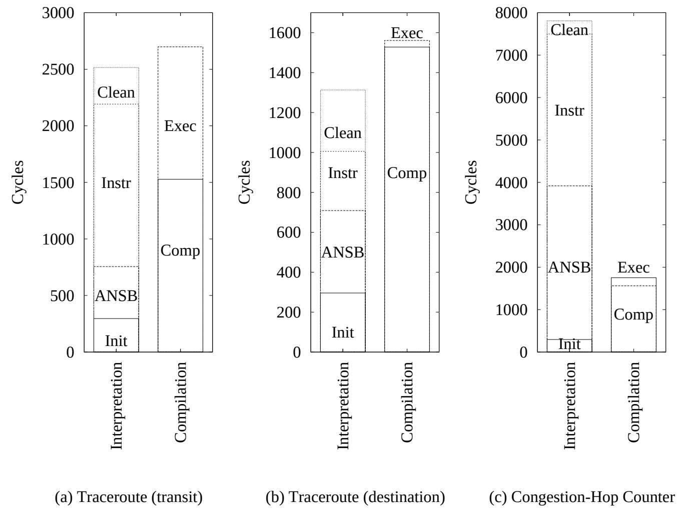

Fig. 5. Comparing cycle cost for interpretation with cycle cost for compilation and native execution.

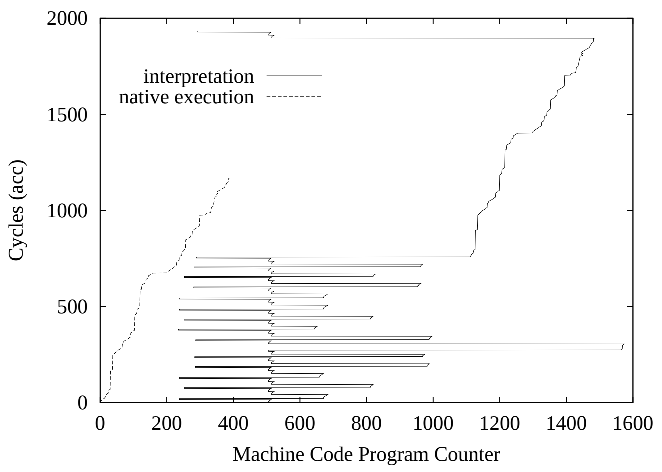

Fig. 6. Accumulated cycle costs for interpretation and native execution of the traceroute program at a transit router. The horizontal structure in the case of interpretation is due to the branch to interpret each instruction. The main loop of the interpreter is near the machine code program counter 500.

This total interpretation cost is compared with the sum of the cycles needed for compilation (Comp) and native execution (Exec). Figure 5 shows that the JIT approach does not lead to a performance improvement for the traceroute example (a, b). However, it does so for the congestion hop counter example (c). This demonstrates that the JIT compilation approach described so far may already lead to speed-ups as soon as parts of the program are executed more than once (e.g., because of looping in (c)). In this case the overhead for instruction mapping and memory-based stack operations increasingly dominates the total cost of interpretation. The identified relative costs of compiling, interpreting, and natively executing source instructions as well as the congestion hop counter example, reveal that a single repeated execution of large parts of a program can already pay back the investment for compilation. The high cycle cost of the transit traceroute example—even in the case of compilation—is, as in Figure 6, due to the `SEND` operation.

C. *Caching*

Relative to interpretation, the compilation overhead can be reduced further if the compiled code is cached either at nodes (i.e., basically not removed from instruction memory) or inside the packet. In the first case, packets of the same flow could reuse native code with different initial memory entries. In the second case, a packet can reuse native code if another router on the way to the destination supports the same native instruction set. In a homogeneous

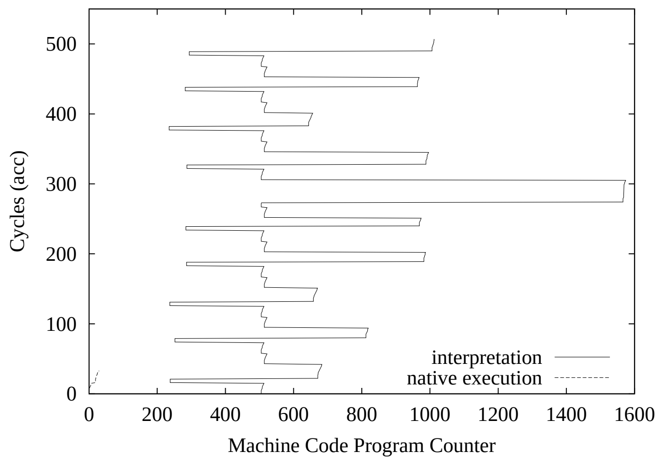

Fig. 7. Accumulated cycle costs for interpretation and native execution of the traceroute program at a destination router.

network environment, byte-code vectors in packets could even be replaced by the compiled native code at the first active network node in order to minimize packet overhead.

Thus, we revisit the examples introduced above and compare interpretation (native code execution of interpreter interpreting the active code) directly with native execution of the compiled version without considering the additional cost of compilation. From measurements of the exact costs for executing the example programs in the interpretation and compilation approach, Figures 6 to 8 have been produced. The accumulated number of cycles is plotted over the program counter. The figures clearly show the amount of branching during interpretation. For interpreting each byte-code instruction, the program counter jumps to the entry point of the instruction code. Native execution however shows loops only if the source code itself contains loops, as for instance with the congested hop counter example (see Figure 8).

The predominance of the `SEND` operation in the traceroute example when executed at a transit router is visible in Figure 6. There is no benefit with native execution with this operation because it is implemented as a core router service. Only 29 cycles have been executed with native execution before the `SEND` is invoked. Without the `SEND` operation, the compiled program is therefore more than 26 times faster. With the `SEND` operation the speed-up is about 1.7.

If traceroute is executed at the destination, the `SEND` operation is not invoked. In this case a speed-up of over 15 times is achieved. In Figure 7 the line for native execution ends at 33 cycles.

For the congested-hop counter example, a speed-up of nearly eight times is achieved. The execution of the congested-hop counter example shown in Figure 8 is also

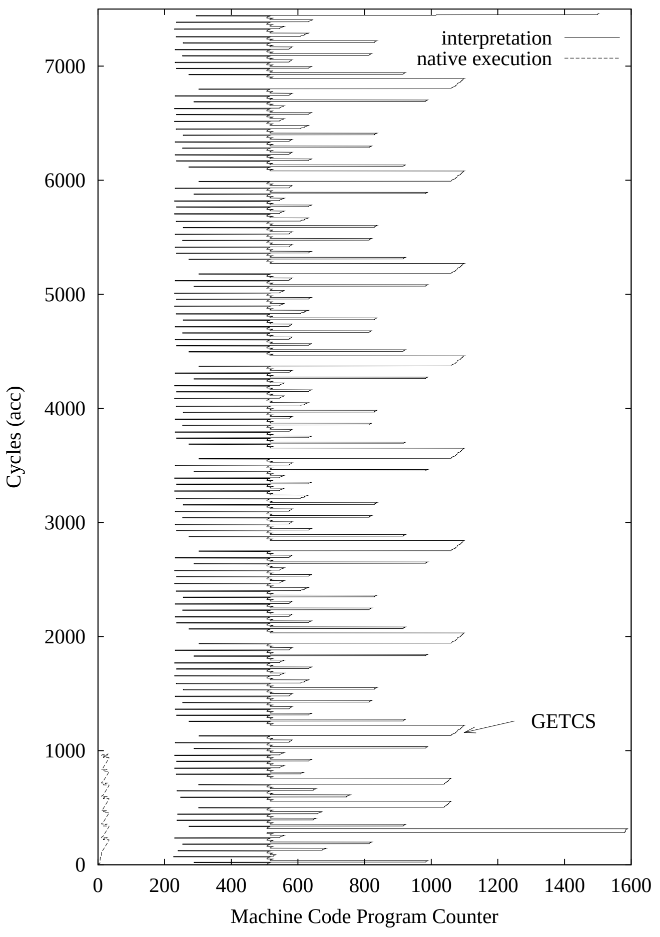

Fig. 8. Accumulated cycle costs for interpretation and native execution of the congested-hop counter program (using eight queues).

dominated by core router service functions. The tree lookup operations `GETNQ` and `GETCS` are responsible for about 80% of the executed cycles with native execution. Without the service functions, the speed-up would even rise to over 30 times.

The inequation given earlier in this section can be adapted to also address the case of native code caching:

$$aT_I > T_C + aT_E \ , \tag{7}$$

where a is the number of times a native program can be executed without recompilation. By re-arranging the inequation and assuming $T_E \ll T_I$, we obtain $aT_I > T_C$. As we have found that compilation is only slightly more expensive than interpretation, a performance benefit can in general be expected already with $a \geq 2$.

VI. Conclusions and Future Work

The approach in this paper is different from previous work on improving active network performance. First, an existing active network language is used. Second, active network nodes are based on network processors. And third, active network programs are compiled before execution into native network processor code at an active network node. We have shown that this JIT compilation approach can lead to a significant performance improvement if the source code contains loops or if the approach is combined with native code caching. In these cases the overhead for instruction mapping and memory-based stack operations increasingly dominates the total cost of interpretation.

The results have been obtained with an implementation of the SNAP language on a PowerNP network processor. Although we tried to gain experience towards JIT compilation for active networks on network processors in general, not all of the assumptions apply to the currently existing network processors. Critical for the generic applicability of the described approach are write access to the instruction memory and the instruction memory size that needs to be big enough to hold the compiler and the JIT-compiled active code.

The mechanisms for code caching either at nodes or inside packets have not yet been integrated into the current implementation. Further work is also planned on static analysis of active network programs. Information from static program analysis could help to determine optimum initial resource bounds so that it can be assured that benign packets reach their destinations. This information could also be used to determine the range of a packet meaning the set of final destinations reached by the packet or its descendents, for a known resource bound.

Acknowledgments

We would like to thank the anonymous reviewers and in particular Scott Karlin for their valuable comments and help while preparing the final version of this paper.

References

[1] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge, "Smart Packets: Applying active networks to network management," *ACM Transactions on Computer Systems*, vol. 18, no. 1, pp. 67–88, Feb. 2000.

[2] D. J. Wetherall and D. L. Tennenhouse, "The ACTIVE IP option," in *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[3] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router Plugins: A software architecture for next-generation routers," *IEEE/ ACM Transactions on Networking*, vol. 8, no. 1, pp. 2–15, Feb. 2000.

[4] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M. Smith, "Active Bridging," in *Proceedings of the ACM SIGCOMM Conference*, New York, Sept. 1997, vol. 27 of *Computer Communication Review*, pp. 101–114, ACM Press.

[5] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura, "An architecture for active networking," in *Proceedings of INFOCOM '97*, Apr. 1997.

[6] S. Rooney, J. E. van der Merwe, S. A. Crosby, and I. M. Leslie, "The Tempest: A framework for safe, resource-assured programmable networks," *IEEE Communications Magazine*, vol. 36, no. 10, pp. 42–53, Oct. 1998.

[7] D. J. Wetherall, "Active network vision and reality: Lessons from a capsule-based system," *Operating Systems Review*, vol. 34, no. 5, pp. 64–79, Dec. 1999.

[8] L. P. Deutsch, "A LISP machine with very compact programs," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Aug. 1973, p. 697.

[9] E. C. R. Hehner, "Computer design to minimize memory requirements," *IEEE Computer*, vol. 9, no. 8, pp. 65–70, Aug. 1976.

[10] J. Ernst, W. Evans, Ch. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression," in *SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997, pp. 358–365.

[11] B. Krupczak, M. H. Ammar, and K. L. Calvert, "Implementing protocols in Java: The price of portability," in *Proceedings of INFOCOM '98*, Mar. 1998, pp. 765–773.

[12] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy, "The structure and performance of interpreters," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996, pp. 150–159.

[13] M. Cierniak and W. Li, "Just-in-time optimizations for high-performance Java programs," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1063–1073, 1997.

[14] J. T. Moore, M. Hicks, and S. Nettles, "Practical programmable packets," in *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)*, Apr. 2001.

[15] "IBM PowerNP NP4GS3 Network Processor Datasheet," http://www.ibm.com/chips/techlib/ techlib.nsf/products/IBM_PowerNP_NP4GS3, Feb. 2002.

[16] M. W. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles, "PLAN: A packet language for active networks," in *International Conference on Functional Programming*, 1998, pp. 86–93.

[17] B. Schwartz, W. Zhou, A. Jackson, W. Strayer, D. Rockwell, and C. Partridge, "Smart packets for active networks," *ACM Computer Communications Review*, Jan. 1998.

[18] J. J. Hartman, P. A. Bigot, P. Bridges, B. Montz, R. Piltz, O. Spatscheck, T. A. Proebsting, L. L. Peterson, and A. Bavier, "Joust: A platform for Liquid Software," *IEEE Computer*, vol. 32, no. 4, pp. 50–56, Apr. 1999.

[19] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," *Proceedings of IEEE OPENARCH'98*, Apr. 1998.

[20] M. P. Plezbert and R. K. Cytron, "Is *Just in Time = Better Late than Never*?," in *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 15–17 Jan. 1997, pp. 120–131.

[21] Intel Corporation, *IXP1200 Network Processor Datasheet*, September 2000.

[22] J. T. Moore, "Safe and efficient active packets," Tech. Rep. MS-CIS-99-24, University of Pennsylvania, Oct. 1999.

[23] Vitesse Semiconductor Corporation, Longmont, Colorado, *IQ2000 Network Processor Product Brief*, 2000.

[24] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, "Building a Robust Software-Based Router Using Network Processors," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001, pp. 216–229.