A Router Primitive Approach for Building Diverse Network Services

Joel Sommers
Colgate University
jsommers@colgate.edu

Paul Barford
University of Wisconsin–Madison
pb@cs.wisc.edu

Ben Liblit
University of Wisconsin–Madison
liblit@cs.wisc.edu

***Abstract*—The vantage points of routers along end-to-end paths in the Internet have long made them a compelling target for additional functionality. In this paper, we describe a new method for router data plane programmability that is simple, general, flexible and safe and enables complex network services to be built and deployed. Our method targets network processor (NP)- and FPGA-based routers, and is based on two central ideas: (1) *primitive functions* for routers that are designed for specific target service domains, and (2) a *primitive-aware programming language* that is expressive, easy to use and easy to analyze. To demonstrate our method, we describe a set of primitive functions in the context of three diverse domains — network measurement, real-time applications and traffic engineering. Next, we describe Morpheme, our primitive-aware programming language, and show how it can be statically analyzed to safeguard router processor and memory resources. We demonstrate the efficacy of our method by implementing the primitive functions in the Click modular router [1] and a prototype Morpheme compiler. Through a series of microbenchmark experiments, we substantiate the capabilities of our prototype implementation.**

I. Introduction

A spectacular array of Internet applications and services has been developed over the past several decades. These applications have been enabled in no small measure by the exponential increase in bandwidths connecting end hosts to the rest of the Internet. However, *functionality* within the network from the user perspective has remained generally limited to simple packet forwarding. We argue that in order to foster and facilitate future applications and services, *i.e.*, "the next big thing," it is critical to consider new functional capabilities *within the network*.

Routers are appealing targets for new functionality since they dictate how traffic is forwarded between end hosts and because of their potential view of numerous, diverse traffic flows [2]. For example, application developers often have to go to great lengths to ensure quality of service, and network service providers struggle to adapt to traffic dynamics and outages in order to satisfy contractual commitments in service level agreements. Better support for QoS-sensitive applications and for traffic measurement and management could be more easily facilitated from within the network. The key question, however, is how new capabilities can be offered on routers in a simple, flexible, and safe fashion.

In this paper, we describe a new method for router programmability that is based on exposing *primitive functions* in network processor (NP)- and FPGA-based routers that can be harnessed by network operators, researchers, and other interested parties through a purpose-built high-level programming language. We define primitives that cover a broad cross-section of data plane operations, such as packet inspection and modification, and packet forwarding and queuing. We examine potential uses of the primitives through the specific and diverse domains of network measurement, real-time applications, and traffic engineering.

As an illustration to further motivate our approach, the simple example program below uses a set of primitives to inspect packet header elements (the **`field`** primitive) and to either direct packets to a lower priority queue or along a non-standard forwarding path (the **`forward`** primitive) or to modify packet header elements. The program would be compiled, checked, and deployed on a target set of routers by a party with privileges to do so. (We describe these primitives and the other language constructs in Sections IV and V.)

```
; low-priority queue for bittorrent
if field ip proto == tcp:
  if field tcp port range 6881 6999:
    forward next-hop queue 1 tail

; modify TOS bits for icmp pkts
elif field ip proto == icmp:
  field ip tos |= 0x08

; non-default forwarding for dhcp
; to an intermediate host
elif field ip proto == udp:
  if field udp dport 68:
    forward intermediate-host.example.net
```

The primary goal of our work is to simplify and facilitate the development of future Internet applications and services. We argue that either specialized or generalized programming capability *without regard to the functional requirements of specific domains* makes service and application implementation very difficult. If support is too general, programs become larger and more complex, and difficult to debug and maintain. On the other hand, if it is too specific, then certain applications may not be well served. We address this tension by identifying specific domains of interest, including gaming, content delivery, measurement and traffic engineering. We abstract the functional requirements of these domains into simple and general primitive operations that can be built into routers and accessed using a high-level programming language to create

978-1-4673-0298-2/12/$31.00

complex services and applications in a straightforward fashion. The key requirements of the primitives are that they be simple, lightweight, and efficient, thus making them more likely to be implemented and deployed. In this work, we concentrate on data plane-oriented primitives.

The primitives we identify are utilized through a new *primitive-aware programming language* called Morpheme. Our objectives in the design and development of Morpheme are to create a language that is efficient, easy to use, and easy to analyze. We accomplish this through an implementation that directly enables use of the primitive functions and that has a simple set of capabilities beyond the invocation of primitive functions. Our approach facilitates important static analyses and efficient compilation for target router platforms.

The fundamental contribution that we make in this paper is to create a method for exposing capabilities of the underlying hardware in a general, simple, and flexible way that is easily accessed by high level programming constructs. We make these capabilities available in a framework that ensures that resource constraints can be enforced, which was one of the primary limitations of prior methods for in-network programmability.

To demonstrate the efficacy of our approach, we define a set of primitive functions for router data plane operations. While our primitives broadly cover common router data plane operations, we focus on three target domains: real-time applications (*e.g.,* gaming and VoIP), network measurement applications (*e.g.,* packet loss measurement), and traffic engineering and VPNs. To assess our approach, we implemented a prototype Morpheme compiler, which generates an intermediate representation in C++ which can be compiled into executables that could run on or be adapted for any number of existing NP or FPGA platforms. We implemented a runtime environment to support Morpheme programs in the Click modular router [1].

We demonstrate the capabilities of our prototypes through a set of example programs focused on real-time and measurement applications and network virtualization. The examples highlight how router primitives and Morpheme can be used to easily implement applications that would be complex or impossible to implement on end hosts. We also describe how static analysis of Morpheme programs can be used to assess resource consumption. Finally, we run a series of microbenchmark experiments on Morpheme programs in a controlled laboratory environment. The results of these experiments demonstrate the correctness and utility of the primitives and their implementations, and highlight the strength and potential of our approach.

II. Related Work

Our work is informed by prior research on *active networks*. These studies investigated the idea of enabling customized computation on packets as they flow through network devices such as switches and routers [3], [4]. A key objective of active networks was to enable new applications and new capabilities in existing domains such as network management; our objectives are similar. A conceptual similarity is the notion of "primitives" in network devices that can manipulate packets [5], [6]. However, the context for this idea was that "capsule packets" carrying program fragments would be executed by network routers to dynamically extend router and network functionality. Rule-based forwarding [7] also shares some similarities with this earlier work. In contrast, in our approach packets do not carry programs, and only built-in primitives can be accessed by Morpheme programs.

There have been a number of studies on programming environments for network devices. These studies largely fall into two categories: software development kit (SDK)-based approaches [8]–[10], and entirely new programming languages [11]–[13]. Related to SDK-based access are studies that focus on program-based access to configuration functions. Notable among these types of systems is Openflow [14]. While Openflow capabilities can be used to address important issues such as traffic management, an external controller system must be involved for *programmatic* packet processing. We note that Morpheme could be used in ways similar to the example given in Section I to implement any Openflow-like capability.

The notion of including specialized primitive functions in routers has been an important focus of vendors for many years. Indeed, router technology has evolved to the point where vendors are building devices that are programmable [8], [10]. Many of these functions focus on improving the manageability or performance of these devices as opposed to application support. However, the programming paradigms for today's networked systems include either something very close to the hardware (*e.g.*, VHDL [9]) or something very general purpose (*e.g.*, Java [10]). We posit that neither of these hits the mark in terms of providing a natural way to implement network applications and services.

Lastly, we note that this work extends an earlier workshop paper [15] in which we proposed a set of primitives for Internet measurement. We expand and generalize that work by defining a broader set of primitives, defining and implementing the Morpheme language and compiler, a run-time environment in Click, and conducting a set of experiments with these tools to demonstrate the feasibility of our approach.

Table I provides a summary comparison between Morpheme and current programmable and configurable systems. Morpheme is targeted toward deploying network applications and services (*i.e.*, programs, not bits of configuration) on routers that can be easily written and for which resource usage can be statically analyzed. Furthermore, Morpheme is designed to support multiple simultaneous users, and its primitives are designed in the context of specific networking applications.

III. Morpheme Overview and Framework

In this section we discuss ways in which Morpheme could be used, and describe four components of the Morpheme framework for enabling programmable functionality on routers.

A. Overview

New router-based capabilities enabled by Morpheme are potentially useful to multiple constituents. A game application developer could exploit Morpheme capabilities to improve the gaming experience by reducing delay for critical state-carrying

TABLE I
COMPARISON OF CURRENT PROGRAMMABLE AND CONFIGURABLE NETWORKING TECHNOLOGIES.

	Morpheme	Openflow [14]	NetFPGA [9]	Click [1]	JUNOS SDK [8]	Cisco SDK [10]
Programmable?	Yes	Only in controller	Yes	Yes	Yes	Yes
Ease of use	High	High	Low	Medium	Low	Low
Program static analysis	Yes	No	No	No	No	No
Multiple users?	Yes	With FlowVisor [16]	No	No	Unknown	Unknown
Application-focussed	Yes	No	No	No	Yes	Yes
High performance	Unknown	Yes	Yes	No	Unknown	Unknown

packets. Network operators might use Morpheme to monitor and shape traffic demands in order to satisfy service-level agreements. Network providers might delegate access to their customers so that the customers can manage and measure their own traffic streams. There are different security, flexibility, and safety concerns in each of these settings, but our approach could be successfully adapted to each of these contexts.

As an example use case, consider a network researcher who wishes to achieve a network measurement goal on a set of routers in an operational network or research testbed. The researcher might write programs in the Morpheme language to actively collect measurement data from the set of routers (see Section IV for examples of specific programs the researcher might create). The researcher would obtain credentials in order to load the programs on target routers. Once loaded, the programs would execute until stopped by the researcher. Although addressing design tools and tools for automatic resource consumption analysis of Morpheme programs are beyond the scope of the present work, the researcher would use the Morpheme compiler to perform analysis of the programs prior to loading and running them to ensure that they satisfy the resource constraints of the target environment.

B. Framework

(1) Primitive Functions: We specify a set of primitive functions that extend the basic capability of a router, including primitives to cover a broad array of data plane-operations: packet inspection, modification (including dropping a packet), generation, control flow and scheduling, and utility-like functions such as random number generation. Our process for defining primitives focuses on generalizing the functional requirements of specific application and service domains, resulting in capabilities that are both effective and attractive for developers, as well as generally applicable and useful. Our objective is to keep the primitives conceptually simple, and thereby keep implementation feasible. We recognize that this may not always be the case and we plan to investigate the complexities of primitive implementation in future work.

(2) The Morpheme Programming Language: The Morpheme programming language coordinates and gives access to the primitive functions. Our objectives with Morpheme are to develop a language that is expressive, easy to use, and easy to analyze. The language syntax we define relates directly to the primitive functions, and we include constructs for simple but important programmatic capabilities such as loops, conditionals, and mathematical expressions, along with network-specific functions such as generating, inspecting, and modifying packets. Although our current design does not support user-defined functions or data structures such as tables and associative arrays (hashes), we intend to include support for these in the future. A key objective is to make the language restrictive so that programs can be evaluated effectively and will run on systems with limited resources.

(3) Runtime Environment: The initial execution environment for Morpheme programs is envisioned to be simple and relatively easy to implement. Each Morpheme program is allocated a dedicated network processor thread on a target router, with no mechanism for spawning additional threads. This is motivated by the increasing ubiquity of NP-based network systems such as the CRS-1 [17], which support 160 simultaneous execution threads. While this program-per-thread approach is neither scalable nor efficient from an NP utilization perspective, it eliminates the complex scheduling, resource management, and security measures that a shared environment would require. The only other assumption that we make for this paper is that programs have access to all traffic flowing through a node. In a live setting this will almost certainly not be the case for security, privacy, and scalability reasons.

(4) Router Access Control: Our router access control model follows directly from the program-per-thread execution strategy. Morpheme programs can only be executed on routers by users who have proper authorization. We assume that user credentials are established offline. A properly credentialed user will be granted access to some set of routers in a network. The user will have the opportunity to load, start, and stop programs whenever they wish. We envision these control operations to happen in an out-of-band process, and for them to be managed and facilitated with purpose-built tools.

IV. ROUTER PRIMITIVES

The starting point for our work is the idea of specifying simple router-based functions that cover a broad set of data plane-oriented applications and services. In this paper we focus on three target application domains: real-time applications network measurement (both passive and active) and traffic engineering/network virtualization. The selection of these domains was made to highlight the flexibility of our approach. We argue that the primitives are widely applicable in data plane-oriented applications, although we cannot claim completeness of our primitive set. In future work, we plan to apply a

framework such as [18] so that we can reason formally about the scope and capabilities of our primitives.

There are essentially two types of primitives in our design: *events* and *actions*. The basic idea is that an action, or set of actions, can be associated with the occurrence of an event. Invoking an event or action is specified in terms of Morpheme statements. In this section, we focus on the range of primitives in our present design. The next section describes Morpheme from a programming language perspective.

Events trigger the execution of Morpheme code blocks which contain action primitives. Events may be associated with a *timer expiry* or *packet receipt*. Primitives exist for setting timers that will trigger the execution of Morpheme actions. The **after** and **periodic** primitives can be used to set one-off and recurring timers, respectively.

Besides timers, an action or sequence of actions can be associated with the arrival of a packet that matches certain criteria. In this work, we assume that all packets pass through the Morpheme runtime environment. While this assumption is convenient for our initial implementation, it should not be considered necessary. More generally, one could consider a mechanism external to Morpheme (*e.g.*, an access control list) that could perform the necessary function of selecting packets for Morpheme processing.

Action primitives form the core functionality of Morpheme. Here we focus on data plane-oriented primitives, motivated by and generalized from the specific domains of real-time applications, network measurement, and traffic engineering.

Target Domain 1: Real-time Applications: The key requirement for real-time applications is for predictable performance. This typically translates into ensuring low delay and loss in the face of uncertain competing traffic demands.

Traffic classification and marking. The **field** primitive is provided to enable access to arbitrary elements in a packet's headers or payload. A user of the **field** primitive is required to specify the header name (or **payload**) and the size of the data to access and its offset in the specified header, or field name mnemonic. The **field** primitive can also be used to modify a packet's headers or contents. Thus, **field** can be used for classification of a packet as well as marking it. For example, if we wanted to set the IP TOS bits to 0x10 ("low delay") for all UDP traffic, we could use the following code:

```
if field ip proto == udp:
    field ip tos |= 0x10
```

Expedited forwarding. We define a **forward**...**queue** primitive to specify a numeric output queue to which a packet should be directed and how the packet should be queued (at the head or tail of the queue). For example, the following would forward all UDP packets to the head of the default output queue (defined as queue 0):

```
if field ip proto == udp:
    forward next-hop queue 0 head
```

Explicit discard of packets. For low-priority flows or for flows that are excessive consumers of bandwidth, it may be desirable to explicitly drop packets. A simple **drop** primitive is provided for that purpose.

Timestamp manipulation. Monitoring bandwidth consumption of flows is useful for ensuring superior performance for real-time traffic (and for possibly degrading the performance of low priority traffic). In order to evaluate bandwidth consumption over time, the ability to request the current time is necessary. A **now** primitive is provided for that purpose. It yields the current time as a floating point value.

With the above primitives, along with other basic Morpheme features described below, a basic rate-limiting application can be created. The following Morpheme program attempts to cap all TCP traffic to 500 kb/s by periodically computing an exponentially weighted moving average of TCP bandwidth consumption. The moving average is recomputed every ¼ second, at minimum. If the rate exceeds 500 kb/s, all traffic during an interval is dropped.

```
; simple rate limit program.
ratecap = 500000.0
rateewma = 0.0
lastts = now
bytes = 0
alpha = 0.1
mininterval = 0.25
if field ip proto == tcp:
  bytes += field ip len
  ts = now
  ; check if time to recompute rate
  if mininterval < (ts - lastts):
    ratenow = bytes / (ts - lastts)
    rateewma = ratenow*alpha+rateewma*(1-alpha)
    bytes = 0
    lastts = ts
  if rateewma > ratecap:
    drop
```

With other Morpheme constructs, the example could be extended to vary traffic rate limits depending on time of day or other conditions. The example also highlights how an external entity could apply in-network traffic shaping to different classes of their own traffic to preserve QoS for their high-priority flows. Neither of these applications are possible with current generation routers.

Target Domain 2: Network Measurement: For network measurement, the key requirements are to be able to create and send probes for active measurement, to annotate packets with passive measurement data, and to have a minimal impact on other data plane traffic if desired. We define the following set of primitives for use with network measurement tasks.

Create and send a packet. The **probe** action is used to construct and send a new packet. Parameters to this primitive can be used to specify the header and payload contents:

```
probe 10.10.2.1 udp dport 3000 payload {42/4B}
```

The above example creates and sends a UDP packet with a particular destination address and port, and with a payload consisting of the number 42 stored as a 4-byte quantity.

Using the **probe** primitive along with a timer, the following Morpheme program could be used to implement the Badabing packet loss measurement algorithm which sends geometrically

distributed probe pairs, with each probe consisting of three rapidly emitted packets [19]:

```
seq = 0
interval = 0.005
slot = 0
next = geom-rv 0.3
; wait 'next' time intervals before sending next
periodic next * interval:
  repeat i in 3:
    probe 10.0.1.1 udp dport 3000 \
      payload {slot/4B seq/4B i/4B}
  if slot % 2 == 0:
    ; send next probe at next time slot
    next = 1
  else:
    ; otherwise, wait for some geometrically
    ; distributed number of time slots
    next = geom-rv 0.3
  slot += next
  seq += 1
```

We also note that random number generation primitives enable packet sampling, and that the **field** primitive in conjunction with standard mathematical operators can be used to perform hashing for the purpose of load balancing or collection of flow-based traffic statistics. In the latter case, the notion of flow can be flexibly defined within a program.

Annotate a measurement probe. Another measurement primitive enables annotation of packets with passive measurement data as they are forwarded through a router. For example, high-resolution timestamps could be added to a packet both on ingress and egress from a router. Interface addresses could also be added to a packet. More generally, any passive measurement data that is readily available (*e.g.*, SNMP MIB variables) could be added to a packet as it is forwarded through a router. (Note that modifying packet contents may require recomputing header checksums.) We define a set of primitives for each of these functions:

```
input-timestamp
output-timestamp
input-address
output-address
input-mib 1.3.6.1.2.1.31.1.1.1.6.
output-mib 1.3.6.1.2.1.31.1.1.1.6.
```

In the example above, packets are annotated with timestamps, interface addresses, and the `ifHCInOctets` counter (from the Interfaces Group MIB [20]) on both ingress and egress.

Conditional packet forwarding. The **forward...when** action can be used to specify that a probe should be held until a condition involving the output queue becomes true. This statement implies that additional buffer space and processing is needed at router egress. Packets that are processed with **forward...when** are held until the specified condition becomes true. Because buffer space is finite, packets held awaiting a **when** condition may be eventually dropped. We discuss this issue further in Section VI.

Interestingly, using **forward...when** could be used to implement a record-route feature that has minimal impact on other data plane traffic:

```
input-address
output-address
forward next-hop when outputqueue < 0.1
```

Note that the destination specified in the **forward** statement can be a specific host rather than the default **next-hop**. Using this feature could enable a user to a packet to a specific intermediate host that is not on the standard forwarding path, thereby enabling active measurement of arbitrary paths.

Target Domain 3: Traffic Engineering and VPNs: The third application domain for which we define an initial set of primitives is traffic engineering and virtual private networks, or more generally, network virtualization. The motivation for this capability is similar to OpenFlow (*i.e.*, to support sophisticated routing and traffic management), but our approach is more general. These applications typically rely on an additional packet header to tag or tunnel traffic, *e.g.*, through an MPLS [21] shim header that includes a label or an additional IP header for GRE. Although we have defined an initial set of primitives for these applications, we have not yet implemented them in our prototype.

We introduce a primitive for packet header insertion and removal (**header** `insert` and **header** `remove`) for these applications. This primitive is required for adding and removing shim headers and to encapsulate and decapsulate an existing IP packet within another. In addition, the **field** primitive describe above can be used for modifying existing shim headers. A combination of the **header** and **field** primitives can be used to implement push-, pop-, and swap-like functionalities for tagged network virtualization techniques, *e.g.*, MPLS. Given these primitives, Morpheme can be used to dynamically assign virtual paths to flows depending on traffic load, time of day, or any other expressible criteria.

V. The Morpheme Language

Morpheme's router primitives provide the end effectors that interact with the network environment. The Morpheme language provides the control logic that guides these interactions. Here we describe how the primitives developed above are embedded within a more fully-functional language to allow creation of complex router programs.

A. Language Design

As a language, Morpheme must be flexible, able to support a wide variety of router-based applications now and into the future. Yet it must also be restrictive enough to allow strong static analysis, and support efficient compilation to run on resource-constrained devices such as NPs and FPGAs. Morpheme's syntax and semantics must be familiar and accessible to a broad range of users: it should make easy things easy, and hard things possible.

1) Concrete Syntax: Morpheme's syntax is inspired by Python. Line breaks separate statements, and indentation marks statement blocks, as in:

```
timestamp = now
if timestamp > previous + 10:
  late += 1
  retry = (late < 5)
else:
  previous = timestamp
```

Beyond assignments, several additional simple statements expose the router primitives discussed in Section IV.

Compound statements include conditionals as seen above, a **repeat** statement for bounded looping, and **periodic** and **after** statements for multi- or single-use timer-triggered execution of statement blocks:

```
repeat <id> in <expr>:
  <statements>

periodic <expr>:
  <statements>

after <expr>:
  <statements>
```

Note that there are no **goto** statements, no **while** loops, and no generalized **for** loops as in C/C++. Furthermore, at present there are no defined functions and therefore no recursion. We likewise omit Cilk-style spawn/sync or any other form of generalized parallelism [22]. The only looping constructs are **repeat** and **periodic**. The former is iteration-bound, while the latter repeats forever but always with a timed delay between iterations. Therefore, it is impossible for a Morpheme program to consume unlimited CPU resources without either completing execution or at least relinquishing the CPU while waiting for a timer to expire. Likewise no Morpheme program can send an unlimited number of packets or consume unlimited network bandwidth before either completing execution or at least sleeping on a periodic timer. This realizes part of a fundamental design goal: that every Morpheme program is bounded in its consumption (or rate of consumption) of system resources.

Expressions follow the style of Python or C/C++, with all of the expected mathematical, logical, and bit-wise operators. A single new expression syntax, "**field** <section> <size> <offset>," accesses <size> bytes at the given <offset> within the given <section> (payload, icmp, udp, etc.) of the packet currently being processed. This field syntax may appear on either the left or right sides of assignments, thereby allowing packet contents to be read, written, or even updated in place. For example,

```
field ip 1 1 |= 0x3
```

forces the ECN CE bits [23] on while leaving the current packet otherwise unchanged.

2) Data Model: Like many scripting languages, Morpheme allows new variables to be introduced and used at any point, without explicitly declaring them first. Unlike most scripting languages, though, Morpheme is intended for compilation to high-performance NP- or FPGA-based routers, with no underlying virtual machine. It is therefore important to statically identify the number, sizes, and types of all variables used by a Morpheme program. Furthermore, every Morpheme program must require no more than some limited amount of data memory, also identifiable statically. This realizes the other part of our goal of bounded (or rate-bounded) resource demands in all Morpheme code.

To achieve this, we adopt a restrictive data model with only a few primitive types and no dynamic allocation. Each Morpheme program variable holds either a Boolean value, a signed integer, or a signed floating-point number. Morpheme currently offers no structs, pointers, or other derived types, though we intend to add support for tables, associative arrays (hashes) and other types in the future. Given our current limited palette of three data types, it is straightforward to infer the types of all program variables automatically.

As can be seen from the examples given above, Morpheme is an imperative language, not a functional one. This is consistent with its scripting-inspired syntax and should be familiar to a broad range of programmers. An imperative style is also a natural fit for our target application domains: it easily expresses in-place modification of packet fields or incremental updates to counters, accumulators, and other measurement data. Purely functional programming necessarily puts additional pressure on data storage, often managed by a garbage collector. It is critical that Morpheme resource demands be small and predictable, supported by a very modest run-time environment. Adding a real-time garbage collector would be undesirable. Thus we opt for the simplicity and predictability of a small, sequential, imperative language.

B. Compiler Implementation

We have implemented a reference compiler for Morpheme. Our implementation consists of about two thousand lines of Java and ANTLR code to create the lexer/parser, type checker, code generator, and other intermediate analysis passes.

1) Lexing, Parsing, and Front End: The entire Morpheme lexer and parser consist of just 152 lines of ANTLR code defining the language grammar, token structure, and mapping of this syntax into the compiler's abstract syntax tree intermediate form. This reflects our desire to keep the language structure simple and accessible. After parsing, Morpheme programs undergo four checking/analysis phases:

Type inference identifies every named variable used in the program, and initially assumes that each has a fictitious "unused" type. We then trace the flow of values from one location to another, and widen each variable's type as needed to accommodate the values it receives.

Type inference makes repeated passes over the Morpheme program until reaching a fixed-point after which no variables' types change. At this point, each variable has the most restrictive type able to hold all values that may flow into it at run time. The type coercion lattice is of finite height (unknown $\prec$ Boolean $\prec$ integer $\prec$ floating-point) and type inference is monotonic, moving types up this lattice but never down.

Type checking validates the types discovered by type inference against the required types of arguments to language primitives. For example, a **probe** port must be integral, while **if** statement's branch condition must be Boolean.

Useless variable analysis identifies and reports variables whose types are still unknown after type inference. This can

happen if a variable is never initialized at all, or if it is only initialized with copies of other uninitialized variables.

Induction variable analysis ensures that **repeat** loops eventually terminate. Each **repeat** <id> **in** <expr> loop steps some named *induction variable* <id> across values from 0 though <expr> − 1 with <expr> being evaluated just once, when the loop begins. If a Morpheme program could directly change an active induction variable (*e.g.*, reset it to 0), then one could create infinite, non-timer-triggered loops, which we clearly wish to avoid. Therefore, within the loop body, the induction variable must neither be assigned to directly nor used as the index for any nested **repeat** loop.

2) Back End and Code Generation: Depending on the target platform, the information provided by the front end may suffice to drive code generation directly. Our reference implementation atop Click, however, requires additional transformations to align certain Morpheme language features with Click.

Callback extraction bridges the gap between Morpheme's timer-delayed compound statements and Click's run-time model of timer-triggered callback functions. We identify each **periodic** or **after** statement and replace it with a compiler-internal statement that registers a one-shot timer and calls some named callback function when the timer expires. The block of code that originally appeared within each **periodic** or **after** statement is extracted and stored separately; the code generator builds a suitable callback function containing these statements. For **periodic** blocks, the callback function additionally installs its own timer to give the desired periodic behavior.

Code generation emits C++ source code as Morpheme's "assembly language." This offers us some measure of portability without having to deal with low-level details such as register allocation or instruction selection for all supported target platforms. A single Morpheme program is converted into a single C++ class definition. Each Morpheme variable is a field in this class, using the statically-declared type selected during type inference. A specially-designated main method represents the Morpheme program's entry point, while additional methods encapsulate **periodic** and **after** blocks so that they may be used as callback functions.

C. Resource Consumption Analysis

Routers provide critical network infrastructure, and have only limited memory, speed, and other resources. Morpheme programs must behave appropriately in this sensitive environment. The restrictions designed into the Morpheme language let us classify Morpheme programs according to their resource-consumption behavior, and perform various static analyses to bound that consumption. Here we discuss these program categories and the analyses that each allows.

1) Program Categories: **Statically mortal** programs must always complete execution after a finite length of time, and this finite limit can be determined completely statically. Programs in this category contain no **periodic** statements. Furthermore, if a program contains any **repeat** or **after** statements, then the iteration count or timer duration (respectively) must be a constant expression known at compile time.

For a statically mortal program, several analyses can compute exact upper limits on resources consumed, including

- maximum execution time, given platform-specific time requirements for primitives such as **probe**, **drop**, etc.;
- maximum number of probe packets or bytes that can be sent across the network;
- total storage required for all program state, since Morpheme offers neither recursion nor heap allocation
- total storage required for all program instructions, since Morpheme offers no dynamic code generation

Dynamically mortal programs must always complete execution after a finite length of time, but this time bound may not be known at compile time. Programs in this category contain no **periodic** statements, but may contain arbitrary **repeat** or **after** statements, including those with iteration counts or timer durations computed at run time. The following program which reads its loop count from a packet is dynamically (but not statically) mortal:

```
repeat attempt in field payload 4 12:
  probe ...
```

A dynamically mortal program is subject to similar analyses as a statically mortal program, but with resource limits given as formulae rather than as compile-time constant values. Thus, we can determine maximum execution time as a function of outside inputs and internally-computed values. Analysis of maximum probe packet counts and bytes sent similarly changes from a single hard value to a function describing resources used as a function of other dynamic behaviors. We retain the ability to compute the exact total storage required for data and instructions, as Morpheme is incapable of expressing programs with unlimited storage appetites.

Statically rate-limited programs are immortal, but can never demand system attention at more than some fixed rate. A program in this category may contain **periodic** statements, but the timer delay argument to each such statement must be a constant expression known at compile time.

Programs in this category allow analysis similar to that of statically mortal programs, but with maxima expressed in terms of resource consumption *rates* rather than absolute counts. As before, we retain the ability to compute the exact total storage required for data and instructions, as this is always limited and statically known in all Morpheme programs.

Dynamically rate-limited programs are immortal, and the frequency with which they may demand system attention is not statically limited or known at compile time. This is the most general category, and includes Morpheme programs whose **periodic** statements compute timer delays at run time, possibly based on outside inputs from the network. Resource consumption analysis for programs in this group is both formula-based (as for dynamically mortal code) and described in terms of consumption rate over time (as for statically rate-limited code). Data and instruction storage requirements remain finite and statically determined.

2) Implications: Table II summarizes the Morpheme program categories, from most to least restrictive, along with the

TABLE II
SUMMARY OF RESOURCE-CONSUMPTION PROGRAM CATEGORIES AND ANALYSES EACH SUPPORTS.

	Restrictions		Possible Analyses				
Program Category	**periodic**	**repeat** & **after**	Time	Packets	Bytes	Data	Code
statically mortal	forbidden	constant	exact	exact	exact	exact	exact
dynamically mortal	forbidden	unrestricted	formula	formula	formula	exact	exact
statically rate-limited	constant	unrestricted	rate	rate	rate	exact	exact
dynamically rate-limited	unrestricted	unrestricted	rate formula	rate formula	rate formula	exact	exact

types of analyses that can be performed. Observe that even the least-restrictive group always provides exact limits on code and data size. Thus, we can verify that some router is at least capable of hosting a given Morpheme program at the time of deployment, enabling some form of admission control.

More generally, these program categories can help form the access control policies mentioned in Section III. The compiler can easily identify a given program's category. A network administrator might limit novice users to injecting only statically mortal code, for which one can prove hard limits on compute time and bandwidth consumption. Most experienced users might be allowed to move down the table, loosening static restrictions and gaining greater dynamic flexibility.

VI. PROTOTYPE IMPLEMENTATION

We implemented a prototype runtime environment for Morpheme in the Click modular router [1]. The functionality of the Morpheme runtime is split across two elements within our Click implementation. The key element is called `MetaMorphic` and implements the bulk of the functionality required to support the various Morpheme primitives and language features. Timer handling (to support **`after`** and **`periodic`**) is handled here, as is probe creation and emission, packet access and modification, and ingress passive data annotation (*e.g.*, input timestamps, input MIB data). Most importantly, the compiled C++ code emitted from the Morpheme compiler is loaded and invoked by `MetaMorphic`.

All functions that are logically related to router egress are *preprocessed* in the `MetaMorphic` element. In essence, metadata are added to a packet to indicate that it needs a certain type of egress processing by the Morpheme runtime. For example, metadata are added to indicate any passive data that need to be written to the packet on egress. The metadata are added so that a back-end Morpheme element (`MorphQueue`) can perform any remaining run-time tasks prior to packet egress. We note that any passive data to be added to a packet in `MorphQueue` are written just before the packet is passed on to be sent to the network. Thus, timestamps and other data are applied as close as possible to the packet leaving the router.

Besides application of egress passive measurement data, processing of **`forward...when`** and **`forward...queue`** statements are handled in the `MorphQueue` element. Metadata are added to a packet in `MetaMorphic` to indicate the type of back-end processing that needs to take place. In the case of *when* processing, a reference to a Boolean function generated by the Morpheme compiler is passed along in order to evaluate the condition expressed in the original Morpheme program. A fixed-size circular queue is used to store packets waiting for a *when* condition to become true ("packet purgatory"). As the packets are added and removed from the queue, the Boolean functions associated with packets in purgatory are executed. If the function returns true, the packet is released and forwarded. If the queue is filled and a new packet arrives for *when* processing, the packet at the head of the queue is dropped to create space for the new arrival. To a user, it will appear that the packet is lost, but no indication is sent to the user as to its fate. This issue brings up a larger question of how to handle exceptional conditions in the Morpheme runtime — should a user be immediately notified using, *e.g.*, a mechanism similar to an SNMP trap? If not, should some other method be used to propagate exceptions? In future work we hope to gain more insight into these issues.

Finally, we note that a restriction with our current implementation is that it only supports Click running in user-mode, which limits performance. We intend to make the necessary changes for a kernel-mode implementation in the future.

VII. EVALUATION

We now describe a series of microbenchmark experiments to evaluate our prototype implementation of Morpheme and the Morpheme runtime in Click. Through these experiments, our goal is to demonstrate the capabilities of Morpheme in simple and easily analyzable scenarios. To that end, our microbenchmarks are not intended to demonstrate "new" networking functionalities, but rather to understand the performance and utility of Morpheme in well-studied application contexts.

A. Testbed

For our experiment testbed, we used two nodes to generate traffic that flowed through another node running a Morpheme-enabled version of Click. Traffic was then forwarded to a single sink node. All links were 1 Gb/s. On either side of the Click node we installed Ethernet taps that diverted copies of all packets received by or sent from the Click host to a separate host. We used this additional calibration node as a way to better understand delays or inaccuracies introduced by Morpheme. Each host in our testbed ran Linux 2.6.31 kernels and each had multiple Intel Gigabit Ethernet interfaces, a quad-core Intel Xeon processor running at 2.4 GHz, and 2GB of RAM. Due to restrictions with running Click in user mode, we needed a way to artificially induce queuing within Click. To accomplish that, we introduced a `BandwidthShaper` element configured to

shape traffic to 2 Mb/s. The result is that packets could build up in the back-end `MorphQueue` element, which was configured to hold a maximum of 100 packets.

Unless otherwise stated below, we used a simple background traffic scenario for all experiments. Using Harpoon [24], we created a set of 5 long-lived TCP flows from each source node to the sink node. In addition to this traffic, we sent UDP probes from one of the source nodes to the sink node. These probes were sent at a rate of one per second and were used to collect information from Morpheme. As the probes were forwarded through the Click node, Morpheme was configured to annotate them with various passively-collected data, *e.g.*, timestamps. In our experiments below, we use the passive measurement data collected in these "drive-by" probes in order to understand the behavior and performance of Morpheme. Importantly, we verified that the data collected in the probes was accurate and sufficiently precise using the calibration node in our testbed. On this node, we collected packet traces that were used for the verification process. For all experiments, our comparisons revealed that the probe-collected data was accurate, thus we examine the probe data directly in the experiments described below. Finally, we note that all microbenchmark experiments were run for 3 minutes each.

B. Experiments

a) Baseline experiment: For our initial experiments we used the following simple Morpheme program:

```
input-timestamp
output-timestamp
```

Additional Morpheme code (not shown) specifies that these timestamps are only applied to the UDP probes that pass through the router. (This is true for all experiments described unless otherwise specified.)

We ran the above program in a scenario with no background traffic, and in one with two long-lived TCP sources. In the experiment with no background traffic, the average time taken for a packet to pass through Click, as measured using the Morpheme-enabled timestamps, was about 2 milliseconds. This relatively high value is due to running Click in user mode. Figure 1 shows results for the scenario with the two long-lived TCP sources. We see that the timestamp differences reveal the well-known sawtooth behavior of TCP in simple network settings.

b) Probe emission: Next, we assess the ability of the Morpheme runtime to accurately emit probes. We used the program below to emit probes every second with embedded sequence numbers and timestamps.

```
periodic 1.0:
    pnum += 1
    ts = now
    probe 10.10.200.1 udp dport 4000 \
        payload {pnum/4B ts/8B}
```

We experimented with this Morpheme program both in the absence and presence of background traffic. In each case, the Morpheme runtime maintained the packet emission schedule as specified, which we verified from our calibration host. In the experiment with background traffic some probes were lost at the output queue of the router, indicating that Morpheme could be used to actively measure loss *on the same router* from which probes are sent.

c) High-priority forwarding: In this experiment, we ran the following program with 10 background traffic sources and the UDP probes:

```
input-timestamp
output-timestamp
output-mib queue-length
forward next-hop queue 0 head
```

Our goal was to evaluate the effect of high-priority forwarding primitives in Morpheme. In this experiment, the UDP probes are inserted at the head of the output queue. All other TCP traffic is processed normally (*i.e.*, added to the tail of the output queue). Figure 2 shows a CDF of internal router delays for the UDP probes. From the figure, we see that probes see low forwarding delays despite the router being heavily congested. The maximum of 10 milliseconds is related to the default time slice for the user-mode Click process. When examining the output queue length at the time that each UDP probe is sent to the egress link (via information stamped by Morpheme into each UDP probe), we found that, as expected, it was generally close to full. Because of the Morpheme runtime processing, the probes essentially bypass this congestion, resulting in significantly better service.

d) Rate limiting: We used the rate limit example program shown in Section IV in our final benchmark experiment. That program implements a basic exponentially-weighted moving average calculation of the background TCP traffic. This average is computed periodically and is used to test whether the rate is above a 500 Kb/s limit. If so, then all TCP traffic during the next interval (before the next EWMA calculation) is dropped. Clearly, this kind of rate limitation is rather blunt, but serves to illustrate the capabilities of Morpheme. Note that for UDP traffic, this rate limit does not apply. Also in this experiment, we added Morpheme code to include the computed EWMA rate in the UDP probe as it passes through the router (using a **`field`** expression) as well as timestamps on input and output. Thus, we could easily collect measurements from Morpheme itself about how the program ran.

Figure 3 plots the computed EWMA value from the Morpheme program that is stamped into probes as they pass through the router. Observe that the measured rate hovers around 500 Kb/s, which is the target maximum rate in the Morpheme program. It is expected that the rate will rise above this level due to recomputing the EWMA at discrete time intervals. Still, Morpheme effectively limits the bandwidth consumption of all TCP traffic.

We believe that the microbenchmark experiments described in this section highlight the capabilities, ease of use, and potential of Morpheme. In future work, we intend to investigate more complex examples and scenarios.

VIII. Summary and Conclusions

The increasing use of network processors in routers and growing capabilities of devices such as NetFPGAs suggest

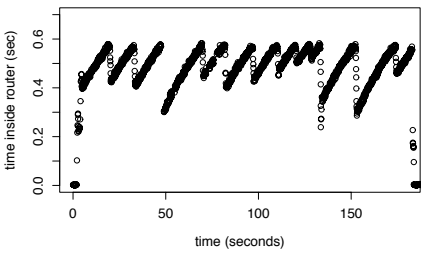

Fig. 1. Baseline Morpheme program with two background TCP sources.

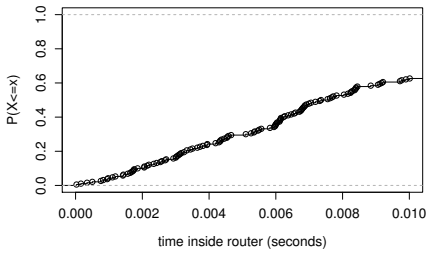

Fig. 2. CDF of probe forwarding times through Click (egress-ingress timestamp).

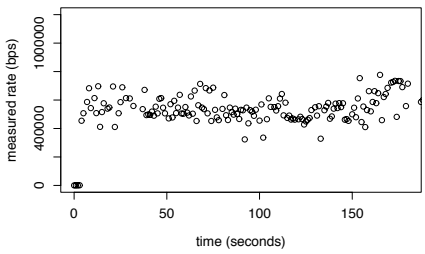

Fig. 3. Results from rate limit benchmark experiment.

intriguing possibilities for new router-supported applications and services. The goal of our work is to develop a simple, yet powerful programming environment for routers that facilitates development of network applications and services. Our approach is based on specifying general primitive functions that are accessed and composed into more complex functions through a new high-level programming language, Morpheme. To demonstrate our approach, we specify a set of data plane-oriented primitive functions along with the primitive-aware Morpheme language, which includes additional capabilities that facilitate the composition of larger programs. We built a compiler which accepts programs written in the Morpheme language and produces C++ code, which can be used as input to target platforms. We demonstrate how the Morpheme language facilitates resource analysis, which is critical to effective use on resource constrained devices. Finally, we implemented the primitive set in Click and conducted experiments that highlight the utility of the primitive functions. The results of the experiments illustrate the correctness of our implementation and the simplicity of functions implemented in Morpheme.

In future work, we plan to expand the primitive set to include control plane operations, and to consider our primitive functions in a reasoning framework so that we can formally assess their characteristics. We also plan to examine hardware implementation issues toward the goal of being able to run Morpheme programs on live systems such as NP-and FPGA-based devices and scale to high-speed links.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-0716460, CNS-0831427 and CNS-0905186, NSF CAREER awards CNS-1054985 and CCF-0953478, DoE contract DE-SC0002153, and LLNL contract B580360. Any opinions, findings, conclusions or other recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the NSF, the DoE, or LLNL.

REFERENCES

[1] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, August 2000.

[2] J. Saltzer, D. Reed, and D. Clark, "End-to-end Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, November 1984.

[3] A. Campbell, H. D. Meer, M. Kounavis, K. Miki, J. Vicente, and D. Villela, "A survey of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 2, 1999.

[4] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Mindin, "A Survey of Active Network Research," *IEEE Communications Magazine*, January 1997.

[5] D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *ACM SIGCOMM Computer Communications Review*, vol. 26, no. 2, April 1996.

[6] M. Hicks, P. Kakkar, J. Moore, C. Gunter, and S. Nettles, "PLAN: A Packet Language for Active Networks," in *Proceedings of ACM SIGPLAN ICFP*, September 1998.

[7] L. Popa, N. Egi, S. Ratnasamy, and I. Stoica, "Building extensible networks with rule-based forwarding," in *USENIX OSDI*, 2010.

[8] J. Kelly, W. Araujo, and K. Banerjee, "Rapid Service Creation using the JUNOS SDK," in *Proceedings of ACM SIGCOMM PRESTO Workshop*, August 2009.

[9] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: Reusable Router Architecture for Experimental Research," in *Proceedings of ACM SIGCOMM PRESTO Workshop*, August 2008.

[10] B. Davie and J. Medved, "A Programmable Router for Service Provider Innovation," in *Proceedings of ACM SIGCOMM PRESTO Workshop*, August 2009.

[11] A. Voellmy and P. Hudak, "Nettle: Taking the sting out of programming network routers," in *PADL*, January 2011.

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The *nes*C Language: A Holistic Approach to Networked Embedded Systems," in *Proceedings of ACM SIGPLAN PLDI*, June 2003.

[13] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN ICFP*, September 2011.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, 2008.

[15] J. Sommers, P. Barford, and M. Crovella, "Router Primitives for Programmable Active Measurement," in *Proceedings of ACM SIGCOMM PRESTO Workshop*, August 2009.

[16] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?", in *Proceedings of OSDI*, October 2010.

[17] "Cisco CRS-1 Multishelf System," http://www.cisco.com/en/US/products/ps5842/.

[18] M. Karsten, S. Keshav, and S. Prasad, "An Axiomatic Basis for Communication," in *Proceedings of the Fifth Workshop on Hot Topics in Networks*, Irvine, CA, November 2006.

[19] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Improving Accuracy in End-to-end Packet Loss Measurement," in *Proceedings of ACM SIGCOMM*, August 2005.

[20] K. McCloghrie and F. Kastenholz, "The Interfaces Group MIB," IETF RFC 2863, 2000.

[21] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," IETF RFC 3031, January 2001.

[22] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of ACM SIGPLAN PPOPP*, 1995.

[23] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," IETF RFC 3168, September 2001.

[24] J. Sommers and P. Barford, "Self-configuring network traffic generation," in *Proceedings of ACM SIGCOMM IMC*, 2004.