HYBRID RESOURCE CONTROL FOR FAST-PATH ACTIVE EXTENSIONS

by

Parveen K Patel

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

December 2003

Copyright © Parveen K Patel 2003

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Parveen K Patel

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

____________________ ____

Chair: Frank J. Lepreau

____________________ ____

Wilson C. Hsieh

____________________ ____

John D. Regehr

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of __________ Parveen K Patel __________ in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

____________________ ____________________________________

Date

Frank J. Lepreau
Chair: Supervisory Committee

Approved for the Major Department

____

Christopher R. Johnson
Chair/Director

Approved for the Graduate Council

____

David S. Chapman
Dean of The Graduate School

ABSTRACT

The ability of active networks technology to allow customized router computation critically depends on having resource control techniques that prevent buggy, malicious, or greedy code from affecting the integrity or availability of the router's resources. It is hard to choose between static and dynamic checking for resource control. Dynamic checking has the advantage of basing its decisions on precise real-time information about what the extension is doing but causes runtime overhead and asynchronous termination. Static checking, on the other hand, has the advantage of avoiding asynchronous termination and runtime overhead, but is overly conservative. This thesis presents a hybrid solution: static checking is used to reject extremely resource-greedy code from the kernel fast path, while dynamic checking is used to enforce overall resource control. The hybrid solution uses a restricted programming model that guarantees termination. It leverages the termination guarantee to reduce the overhead of runtime checks and to avoid asynchronous termination.

This thesis also presents a design and initial implementation of the key parts of the hybrid resource control technique in a router toolkit called RBClick. RBClick is an extension of the Click modular router toolkit, customized for active networking in Janos, an active network operating system. Untrusted extension code is written in a resource-bounded version of Cyclone, a type-safe version of C. RBClick would allow users to download new router extensions directly into the Janos kernel. The thesis shows, by presenting an analysis of existing and new extensions, that hybrid resource control can be successfully applied to many classes of extensions. Further, as compared to the dynamic resource control in Janos, the hybrid solution can improve the performance of router extensions by up to a factor of two.

To my parents.

CONTENTS

ABSTRACT . . . . **iv**

LIST OF FIGURES . . . . **viii**

LIST OF TABLES . . . . **ix**

ACKNOWLEDGMENTS . . . . **x**

CHAPTERS

1. INTRODUCTION **1**

2. BACKGROUND **5**

2.1 Software Systems 5
2.1.1 Click 5
2.1.2 Cyclone 6
2.1.3 Janos 8
2.2 Overhead of Resource Controls in Janos 8

3. HYBRID RESOURCE CONTROL **13**

3.1 Overview 13
3.2 Static Analysis 15
3.3 Runtime Accounting 19
3.4 The Acceptable Limits 19
3.5 Choosing an Inflation Factor 20
3.6 Poll Points 21

4. DESIGN OF RBCLICK **24**

4.1 Overview 24
4.1.1 RBCyclone Elements 24
4.1.2 Deploying an Extension 25
4.2 RBCyclone 26
4.2.1 Why Cyclone? 26
4.2.2 Namespace Control 27
4.2.3 Restricted Programming Constructs 27
4.2.4 Memory Management 28
4.3 Static Analysis 29
4.3.1 Code Analysis of an Untrusted Element 29
4.3.2 Static Analysis of an Extension Configuration 31

5. EVALUATION **32**
5.1 Implementation 32
5.1.1 RBClick 33
5.1.2 CAT 33
5.1.3 RBCyclone 33
5.2 Feasibility 34
5.2.1 Criteria and Experimental Parameters 34
5.2.2 Elements of the IP Router 35
5.2.3 RBCyclone Elements 38
5.2.4 Analysis of Router Configurations 38
5.2.5 Summary 39
5.3 Performance 39
5.3.1 Experimental Setup 40
5.3.2 IP Router 40
5.3.3 Overhead of Cyclone Interface 42
5.3.4 DES 43
5.3.5 ActiveDoom 43
5.3.6 ECN 44

6. RELATED WORK **45**

7. CONCLUSIONS AND FUTURE DIRECTIONS **48**

APPENDICES

A. ANALYSIS OF CLICK EXTENSIONS **50**

REFERENCES **52**

LIST OF FIGURES

2.1 An example Click configuration with four elements. 6

2.2 DARPA active network node architecture. 8

2.3 Software architecture of Janos and the corresponding DARPA active network node architecture. 9

2.4 The three execution levels in Janos. 9

2.5 Hand-off of packets between the system thread and user-level threads at the Moab user-kernel resource control boundary. 12

3.1 High-level algorithm of the Hybrid Resource Control technique. 14

3.2 Results of a memory latency benchmark on an Pentium III 850 MHz machine with 16KB L1 and 256 KB L2 cache. 18

3.3 A system with two nonpreemptively scheduled extensions. 20

3.4 Code executed at a poll point. 22

4.1 An example RBClick extension. 25

4.2 RBClick and Janos. The control plane for RBClick router configurations is implemented via the Bees EE on Moab. 26

5.1 The code for the top-level push function of the ARPQuerier element. . 37

5.2 Experimental configurations of the IP router experiment. 41

5.3 Performance of IP forwarding in three configurations. 41

5.4 Performance of IP forwarding in two configurations. 43

5.5 IP forwarding with DES 44

LIST OF TABLES

2.1 IP−− forwarding rates inside and outside Moab. 11
3.1 Memory access latencies for a Pentium III 850 MHz machine. 16
5.1 Memory latency table used in experiments. 35
5.2 Parameters used by CAT. 35
5.3 CPU estimation of the elements of standard IP graph by CAT. 36
5.4 CPU estimation of RBCyclone elements by CAT. 38
5.5 CPU estimation of router configurations by CAT. 39
5.6 ActiveDoom 44
5.7 IP router with ECN 44
A.1 Classification of Click elements 51

ACKNOWLEDGMENTS

This thesis culminates three years of my life as a graduate student. I achieved this goal with help from a number of people who I would now like to thank. First, I would like to thank my advisor, Jay Lepreau, for teaching me how to do systems research and getting me started on this work. It was Jay's attention to every word I said or wrote that made me take myself seriously. Jay gave me the opportunity to work on STP [25, 24], a project separate from this thesis, which I really enjoyed working on in the past year. I thank him for sending me to conferences and teaching me how to give good presentations.

The work presented in this thesis has benefited from discussions with Alastair Reid, John Regehr, Wilson Hsieh, and Mike Hibler. Alastair's quick responses to my emails pointed me in the right direction and prevented me from ever getting stuck. John and Wilson provided me timely feedback on thesis drafts and helped me improve it. Mike answered all my kernel programming questions. It was reassuring to know that a kind kernel expert was sitting just a few cubicles away. I thank Eddie Kohler from MIT for writing Click, which formed a basis for my work. I learned a great deal about programming by looking at Eddie's code. I thank Leigh Stoller for his initial work on Click in Moab, which helped me get a handle on things quickly. I thank Tushar Mohan for letting me use LBNL's memory benchmark tool, which I used to substantiate my reasoning in Section 3.2. I am grateful to DARPA (grant F33615-00-C-1696) and NSF (grant ANI-0082493) for funding this work.

I thank all my collegues and friends (it would be unfair to list only a few here), who helped me take my mind off work and recharge my batteries. Last but not the least, I thank Divya for being a best friend, an understanding companion, and an encouraging voice.

CHAPTER 1

INTRODUCTION

Active network technology allows users of a network to customize the computation performed at routers using mobile code. Users run mobile code either by including it in each data packet or by installing it ahead-of-time. This flexibility of dynamically adding packet processing code to routers comes with risks. User-supplied code can be buggy or malicious, and by consuming excessive resources can harm the active router, other active services on the router, or the network itself. Therefore, it is important to limit the resources available to active code.

This thesis addresses the issue of resource control of active extensions in *discrete* or *control-plane* active networking [31]. Active extensions are loaded into the active router via a separate control channel and then invoked by examining the headers of data packets. More specifically, we are interested in developing an architecture that allows a rich set of active extensions to be installed in the kernel of an active node operating system. We call these extensions *fast-path* extensions because they extend the functionality of an OS's routing fast-path. For example, an extension implementing application-level gateway functionality could be installed into the kernel to process packets at high speed, provided the protection and resource control challenges were met.

Most existing systems take one of two approaches to control the resources consumed by active code: *sandboxing* or *static analysis*. In sandboxing, active code is run in a resource-limited, preemptive environment and runtime checks are performed to monitor its resource usage. On detection of misbehavior, the active code is asynchronously terminated. Asynchronous termination can affect

the integrity of data structures shared by multiple extensions. The costs to build such a preemptive environment are high, as shown in this thesis.

Static analysis, on the other hand, avoids runtime checks and asynchronous termination by statically verifying that active code does not consume excessive resources. However, static analysis is often conservative and overestimates applications' resource requirements. Difficulties in modeling complex features of modern computers, such as cache hierarchy, and multiple instruction issue also add to the degree of overestimation. In our measurements, ignoring the effects of caching alone can cause static analysis of x86 assembly code to be overestimated by up to a factor of 50. Therefore, it is hard to choose between the two; sandboxing incurs runtime overhead and causes asynchronous termination, while static analysis is very conservative.

This thesis proposes *hybrid resource control*, a resource control mechanism that uses a combination of static analysis and runtime accounting. Static analysis techniques, combined with realistic assumptions about a program's cache behavior, are used to statically predict resource upper bounds of active code. These resource bounds are then used to control admission to a nonpreemptive and lightly protected kernel execution environment, prevent the extensions from running too long, and reduce the overhead of runtime checks. At the same time, precise runtime accounting is used to overcome the restrictions due to the pessimism of static analysis and admit more extensions than otherwise allowed. The resource control technique is termed "hybrid" because of the combination of dynamic and static methods.

We have developed a prototype of the key parts of the hybrid resource control technique in an environment for active extensions, called *RBClick*, "Resource Bounded Click." RBClick is an extension of the Click modular router toolkit [19], implemented in Janos, an active network operating system [32] that was originally designed to achieve resource control only through sandboxing. Active extensions in RBClick are Click graphs involving both trusted and untrusted elements. Trusted elements are taken from a base version of Click, while untrusted elements are written in RBCyclone, a resource-bounded variant of Cyclone [16] that is a type-safe

version of C. RBClick estimates resource bounds on active extensions and acceptable extensions are then loaded into the Janos kernel.

This thesis evaluates hybrid resource control along two dimensions: feasibility and performance. To assess the former, we present an analysis of a Click release and find that acceptably flexible versions of all of its elements can be written with the resource bounding restrictions of RBCyclone. Further, these elements can be successfully analyzed using our prototype static analysis tool. To estimate the performance improvement from reducing the overhead of runtime checks, we compare hybrid resource control with dynamic resource control in Janos. In Janos, RBClick extensions can benefit by up to a factor of two in IP forwarding rate by using hybrid resource control instead of the sandboxing techniques currently being used.

The contributions of this thesis are:

- The hybrid resource control technique that uses a combination and static analysis and runtime checking to efficiently control the resources of untrusted mobile code.

- The design of RBClick, an extensible modular router toolkit based on Click, that would allow users to download untrusted extensions in the Janos kernel. RBClick uses hybrid resource control to control the CPU resource of active extensions.

- An evaluation of the feasibility and potential performance benefits of hybrid resource control using a prototype implementation of RBClick.

The rest of this thesis is organized as follows. Chapter 2 provides some background on three existing technologies that are leveraged by this thesis: Click, Cyclone, and Janos. Chapter 2 also details the the overhead of resource controls in Janos and motivates RBClick. Chapter 3 discusses the mechanisms used by hybrid resource control and lays out their rationale. Chapter 4 details the design of RBClick and its companion tools, RBCyclone and CAT. Chapter 5 evaluates the feasibility and potential performance benefits of hybrid resource control using a prototype

implementation of RBClick, RBCyclone and CAT. Chapter 6 relates hybrid resource control and its implementation with previous work, and finally, Chapter 7 concludes this thesis and discusses some future directions for this work.

CHAPTER 2

BACKGROUND

2.1 Software Systems

This section presents some background on three existing technologies from networking, languages, and operating systems domains that come together in this thesis: Click [22], Cyclone [16], and Janos [32]. Readers familiar with these technologies can skip this section.

2.1.1 Click

Click [19, 22, 18] is a router toolkit for developing modular, extensible, and yet low overhead routers. Routers in Click are built from modular software components called *elements*, written in C++. A Click router configuration is a set of elements connected in a directed graph, which is specified in the Click configuration language. The edges of a Click graph indicate possible directions for packet flow. By convention, Click elements perform small, general functions, such as decrement time-to-live (ttl) field in an IP packet or classify a packet based on its contents, and not complicated, specialized functions, such as "IP routing". Figure 2.1 shows an example Click graph. The graph shows four elements: a `FromDevice` element that reads packets from the network, a `Classifier` element that filters packets based on some user-specified criteria, a `Sink` element that destroys all packets, and a `ToDevice` element that sends packets on the network.

By convention, a Click element has a very well-defined and limited interface. This limited interface constrains how elements interact with each other—an element makes only packet send and receive calls on its neighboring elements. These standard practices, combined with the domain-specific Click specification language, enable automatic analysis and editing of Click routers. For example, Click software

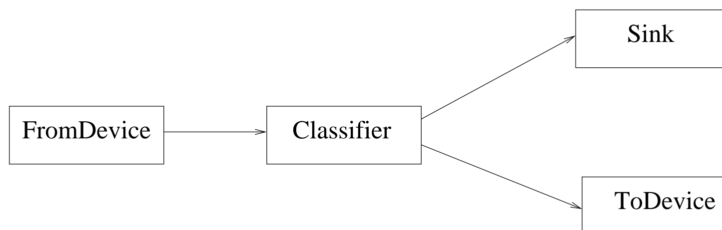

Figure 2.1. An example Click configuration with four elements.

comes with tools that reorganize elements in a Click configuration for optimizations and tools that enforces high-level Click invariants. In Chapter 4, we discuss how we use automatic analysis of Click configurations to enforce resource-boundedness on routers.

Click ships with many router extensions. We worked with Click version 1.2.1, which contains about 234 elements that can be combined in many different ways to form new routers. Many of these extensions are contributed by users of Click in different universities and organizations. Therefore, these extensions, to some extent, represent typical router extensions that users care about and provide us with a useful base to work with.

2.1.2 Cyclone

Cyclone [16] is a type-safe language that is syntactically similar to C. In addition to its type-safety, Cyclone has the following features that make it more suitable for interfacing with C-based environments than other high-level safe languages, such as Java.

1. **Compatibility with C**. Cyclone and C have the same calling conventions and data representation for most data-types. Therefore, it is easy and efficient to transfer data between C and Cyclone programs—most data structures can be safely shared between C and Cyclone by passing pointers instead of copying data. However, some data structures still need marshaling because Cyclone has a different (fatter) representation of pointers that support pointer arithmetic.

2. **Region-based memory management**. Cyclone provides support for region-based, manual memory management, which makes it easy to provide memory safety without incurring the cost of garbage collection (GC) [13]. (By memory safety, we refer to avoidance of nontype-safe and unauthorized memory accesses.) In region-based memory management, each object lives in a region and, except a designated heap region that may be garbage collected, all objects in a region are deallocated simultaneously.

   Cyclone has three types of regions: *heap region*, *stack region*, and *dynamic region*. There is only one global heap region. All static objects are automatically created in the heap and users can explicitly allocate objects on the heap with `malloc` and `new` primitives. However, to preserve memory safety, Cyclone does not provide a `free` primitive. The objects allocated on the heap can only be reclaimed at program termination or using an optional garbage collector. Stack regions correspond to C's local declaration blocks. A stack region's lifetime begins and ends when the control enters and leaves the corresponding declaration block. Dynamic regions, in contrast to heap and stack regions have completely dynamic lifetimes. A dynamic region can be created and destroyed anywhere in the program controlling the lifetime of objects allocated in these regions. In that sense, dynamic regions are similar to memory areas managed by `malloc` and `free` primitives in the C language. These three types of regions provide sufficient flexibility and control over memory management in a Cyclone program.

3. **Lightweight runtime**. Cyclone language has a much smaller runtime compared to other high-level languages, such as Java. It is easily portable to a kernel, as it does not require a full C library. In addition, the Cyclone runtime does not depend on any heavy-weight mechanisms; for example, the garbage collector is optional. We were able to port the Cyclone runtime, excluding the support for garbage collection, to the Janos [32] kernel without much problems.

2.1.3 Janos

The active networking community has agreed on a three-layer general architecture of an active node [28, 26, 1], as shown in Figure 2.2. At the lowest layer is a node operating system, or NodeOS, that manages the resources available on a node and provides primitives for resource management. At the middle-layer, one or more execution environments (EE) provide a programming environment and runtime for active applications. At the topmost layer are the active applications themselves.

Janos [32] is an active network operating system that implements both the NodeOS and EE layers of an active node. Figure 2.3 shows a block diagram of the software layers in Janos and their correspondence with the DARPA active network node architecture. Janos supports Java-based active applications on top of the Bees execution environment, a successor of the ANTS EE [36], which runs on top of the resource controlling JVM, called JanosVM. Together, Bees and JanosVM form the EE layer of an active node and run on top of Moab, the NodeOS in Janos. Moab is an active-node operating system based on the OSKit [10] and it implements the active networks community standard, the NodeOS API specification [1].

2.2 Overhead of Resource Controls in Janos

In this section, we examine the performance of packet forwarders at different execution levels in Janos and point out the resource control overheads in Moab that motivate our work. Figure 2.4 illustrates these execution levels. At the topmost

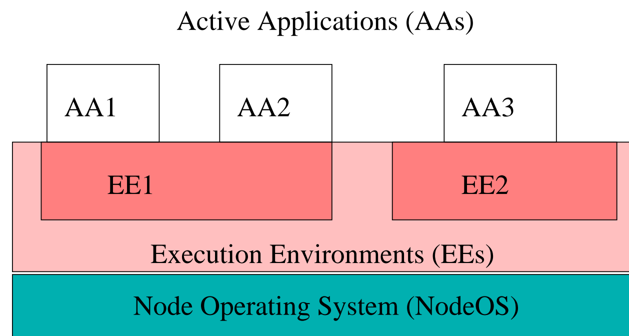

Figure 2.2. DARPA active network node architecture.

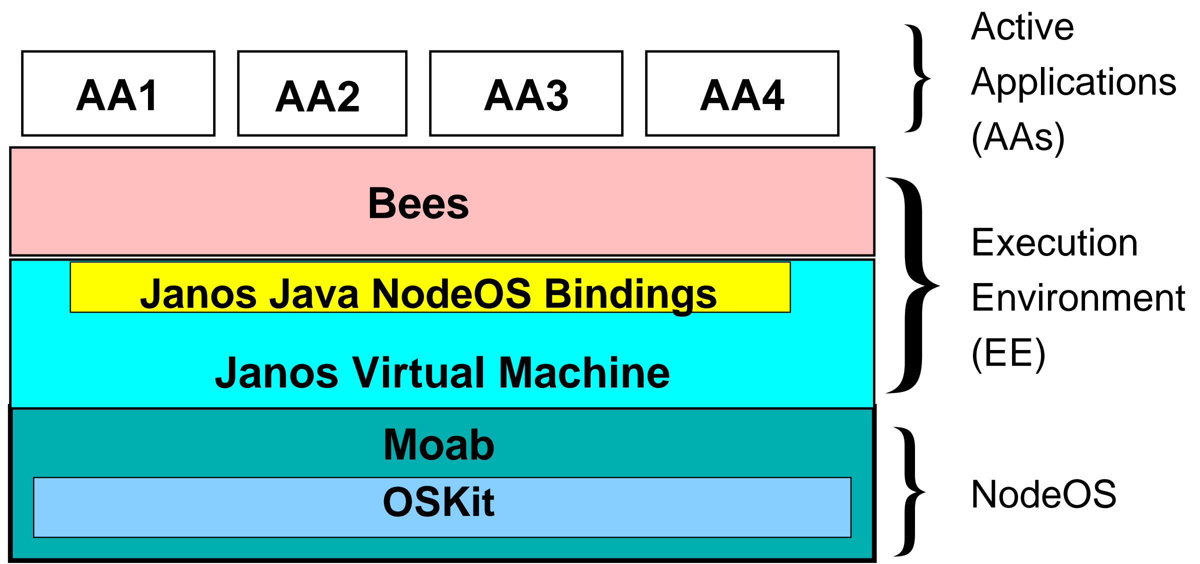

Figure 2.3. Software architecture of Janos and the corresponding DARPA active network node architecture.

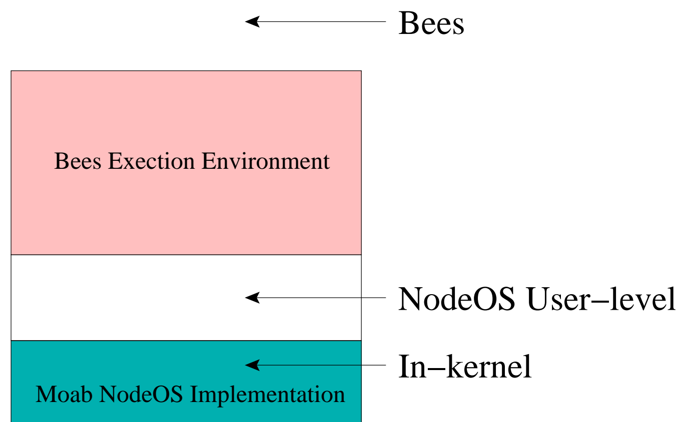

Figure 2.4. The three execution levels in Janos.

level, the Bees execution environment runs untrusted active applications written in Java. At the next lower level, Moab runs C-based trusted applications or memory-safe untrusted applications written to the NodeOS API. At the lowest level, trusted packet forwarders run directly inside the Moab kernel.

The performance of Bees-based active applications was compared with C-based user-level applications by the authors of Bees [29]. The authors implemented a

simple UDP packet relay program at various execution levels in Bees and at Linux user-level. The UDP relay program was a null forwarder—it received and sent packets as fast as it could without looking at or modifying them in any way. The Bees-based packet forwarder performed five times slower than the C-based packet forwarder.

There are many reasons for this performance gap between Bees and C. First, the cost of boundary crossing between C and Java is high. Packets need to be marshaled into proper Java objects before they can be processed by Bees-based applications. Second, the Bees environment incurs costs for resource control and security mechanisms that provide support for general-purpose active programs. The active applications in Bees run twice as slow as normal Java programs in the JanosVM. The JNodeOS layer that implements the NodeOS API specification and the Bees layer that implements the security checks in Java (see Section 2.1.3) are the major causes for this performance slowdown. Third, the Kaffe JVM, on which the JanosVM is based, is not as optimized as commercial JVMs [3]. Finally, Java programs are inherently slower than C programs due to the cost of interpretation and type-safety checks. All these costs add up and slow down Java-based active applications by a factor of 5 as compared to their C-based counterparts [29].

The huge slowdown of Bees programs, as compared to C programs, seems overkill for active applications that add little computation to vanilla IP forwarding. Moab provides resource control support at a much lower level than Bees. However, it does not provide any type of memory safety, and therefore, only active applications written in safe language can be run directly on top of Moab. Cyclone fits this requirement and is reportedly efficient to interface with C [16]. To test its performance, we ported Cyclone to Moab and found that a Cyclone implementation of the UDP relay program performed as well as a C program on Moab. (For a general performance comparison of C vs. Cyclone programs, refer to [16].)

Using Cyclone on Moab, we can offset a significant amount of overhead due to the many layers that constitute the Bees EE. However, further experiments with a complex program show that even Moab's native resource control mechanisms

are quite expensive compared to the computation time of vanilla IP forwarding. To estimate the cost of Moab's native resource control mechanisms, we executed a minimal IP router program, called IP−−, on both sides of Moab's user-kernel boundary.

The IP−− forwarding rates inside and outside Moab are shown in Table 2.1. Note that Moab does not implement any type of memory safety. Therefore, the performance difference between user-level and in-kernel executions is entirely due to resource control checks. Table 2.1 shows that Moab's resource control mechanisms cause about 42% performance slowdown. On further investigation, we found that 40% of this total slowdown is due to preemptive scheduling mechanisms in Moab. Moab controls the CPU consumption of active applications by running each application in its own preemptive threads. This mechanism results in a thread context switch whenever an extension is scheduled or descheduled. Figure 2.5 illustrates these context switches. The remaining 60% of the overhead in IP−− is due to checks performed at the Moab user-kernel boundary. Moab tries to avoid packet drops due to long running extensions by polling the network interfaces at the entry point of its system calls. Note that the overhead due to system calls depends on the number of system calls made by an extension.

Simple calculations show that Moab's preemptive scheduling-based resource control mechanism is an unnecessary overhead for well-behaved extensions. The Moab kernel runs at a clock granularity of 0.01 sec. The processing time for a single packet in IP−− forwarding program is about 9 μsec. This implies that for most extensions the thread running the packet processing function finishes execution before it can be preempted. Thus, if the system knows that the upper bound on execution time of a packet processing function is much lower than the timer

Table 2.1. IP−− forwarding rates inside and outside Moab.

Environment	Forwarding rate (Kpps)
Moab user-level	67
Moab in-kernel	115

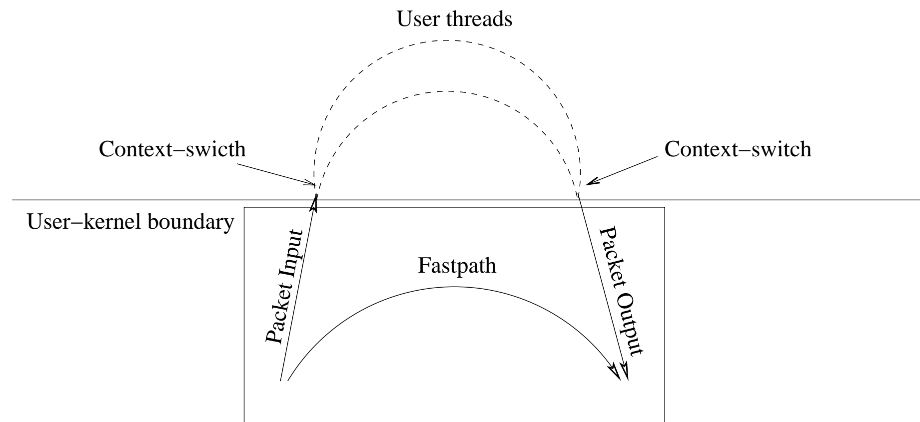

Figure 2.5. Hand-off of packets between the system thread and user-level threads at the Moab user-kernel resource control boundary.

granularity, it can execute that function in the system thread itself and save two context switches per packet. A similar reasoning implies that polling at the system call boundary is also unnecessary overhead and it can be avoided if upper bounds on execution time of packet processing functions are known in advance.

This thesis aims to lower the overhead due to resource control of fast active extensions by predicting the execution time of extensions prior to loading them; and performing minimal resource control checks during their execution. Although we use Janos as our motivating example and later as a platform for evaluation, the general principles of our solution should also apply to other untrusted mobile code based systems.

Dynamically controlling the consumption of two other important active node resources—outgoing network bandwidth and memory—does not require expensive runtime checks. Outgoing network bandwidth is controlled by a low-overhead, proportional-share link scheduler. Controlling the memory only requires one check during memory allocation and one additional instruction during memory deallocation. Since the cost of these additional checks is not significant, we do not change the way these resources are controlled in Moab and focus on the CPU.

CHAPTER 3

HYBRID RESOURCE CONTROL

Hybrid resource control is a technique for efficiently controlling the resources consumed by untrusted active extensions. In this chapter, we discuss the general mechanisms involved in implementing it. Hybrid resource control is designed for fast-path extensions that operate on almost every packet of an end-to-end flow, such as a network-address translator (NAT) or an application-level gateway (ALG).

3.1 Overview

Hybrid resource control employs two key mechanisms: *static analysis* and *runtime accounting*. In this thesis, static analysis is used to infer *soft* upper bounds on resources required to process a single packet using an extension. Extensions that are estimated to consume resources within some *acceptable limits* are admitted and allowed to run nonpreemptively. However, static resource analysis often overestimates resource requirements—sometimes by an order of magnitude. Therefore, completely relying on soft upper bounds for admission can be very constraining, i.e., the system will reject extensions that would actually consume resources within the acceptable limits.

To overcome the limitations due to static analysis, the system conditionally admits all extensions whose estimated resource bounds fall within *inflated limits*. The inflated limits are calculated by multiplying the acceptable limits by an *inflation factor*. The system then ensures that the actual resource consumption of each admitted extension is within the acceptable limits by performing inexpensive runtime accounting to detect extensions that consume excessive resources. Thus, by using a combination of static and dynamic checking, hybrid resource control

enforces a soft upper limit on resource consumption without relying on expensive mechanisms such as preemptive scheduling.

Figure 3.1 illustrates the above discussed high-level algorithm of hybrid resource control. First, the system obtains the source code of an extension to be loaded. The source code is analyzed and its resource upper bounds are estimated. Next, the resource upper bounds are checked against the inflated resource limits. If the extension passes this check, it is loaded into the system, or else it is rejected. The

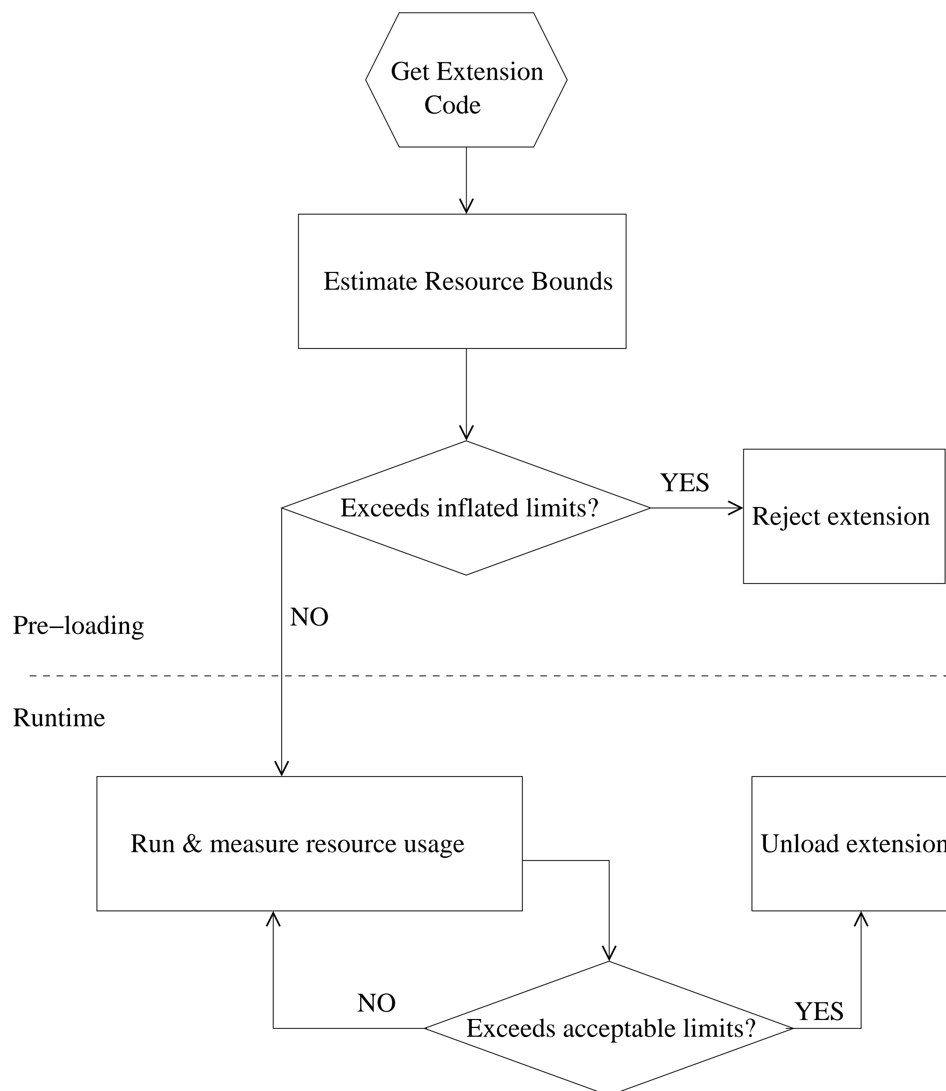

Figure 3.1. High-level algorithm of the Hybrid Resource Control technique.

resource usage of all running extensions is continuously measured and compared against the acceptable resource limits. If an extension consistently violates the acceptable limits, it is unloaded as soon as it becomes idle.

The next two sections discuss static analysis and runtime accounting in more detail. The following two sections discuss rules of thumb to choose the acceptable limits and an inflation factor for a router. Finally, the last section of this chapter describes *poll points*, a mechanism that provides flexibility in using hybrid resource control.

3.2 Static Analysis

As discussed in the previous chapter, 40% of the cost of resource control is due to preemptive scheduling. The goal of hybrid resource control is to prevent this cost by running extensions in a nonpreemptive environment while still enforcing an upper bound on their resource consumption. To achieve this, hybrid resource control uses static analysis to automatically estimate upper bounds on resources required to process a packet using an extension.

The halting problem prevents us from successfully analyzing all programs written in a general-purpose programming language. To determine the potential of using static analysis for typical fast-path extensions, we manually studied the source code of extensions that ship with Click. The results are presented in Appendix A. This study found that all loops in the code of Click extensions can be bound to statically known quantities, such as the maximum size of an Ethernet packet, or the maximum size of a particular protocol header. Further, the total amount of resources consumed by these extensions is also bounded. These results lead us to conclude that these extensions can be easily programmed in a restricted programming language that allows only resource-bounded programs, such as PLAN [15] and SNAP [21][1].

[1] Note that we cannot directly use PLAN or SNAP to write Click-like router extensions because these languages were primarily designed to replace packet headers and hence lack support for global or persistent memory.

Programs written in a resource-bounded language can be successfully analyzed using static code analysis techniques. For example, techniques similar to those used for worst-case execution time (WCET) analysis can be used to estimate upper bounds on execution time. Hybrid resource control leverages on this property of resource bounded languages and requires that an untrusted extension be written in a resource-bounded language. As shown by our study of Click elements, this requirement is not too constraining for typical router extensions.

Static analysis, even when performed on low-level assembly code, is very conservative due to two reasons. First, complex features of modern hardware, such as multiple pipelines, caching, branch prediction, etc., are extremely hard to model accurately and are therefore ignored by conservative static analysis. Second, static analysis cannot know the input data and hence the exact codepath taken at runtime. Therefore, it always assumes the worst-case codepath and estimates much more than the average resource usage of a program.

In our measurements of a Click extension on an Intel Pentium III 850 MHz machine, ignoring the effects of caching alone causes static analysis to be conservative by up to an additional factor of 50. This large factor can be easily understood by looking at the memory access latencies of this machine, as shown in Table 3.1. The table was obtained using the HBench benchmarking tool [7] and it shows the memory read times for all elements of the memory hierarchy. Memory write time for the main memory is about 200 cycles or 33% more than the memory read time. If static analysis assumes all memory accesses to miss the cache hierarchy, it ends up with up to a factor of 75 overestimation for each memory instruction.

Table 3.1. Memory access latencies for a Pentium III 850 MHz machine.

Memory Element	Size	Read Latency
L1 Cache	16KB	2.4 ns (2.0 cycles)
L2 Cache	256KB	7.1 ns (6.0 cycles)
DRAM	512MB	174.3 ns (148.7 cycles)

Static analysis in hybrid resource control avoids a significant amount of overestimation by making realistic assumptions about the cache behavior of an extension. A Click router that only accesses protocol headers of a packet, incurs only five level-2 data and instruction cache misses [18]. These cache misses include the two cache misses to access the DMA descriptors at the input and output device. Even for extensions that access the complete payload of a packet, the number of cache misses is very small due to large cache line sizes (eight machine words in our setup) and compiler-driven prefetching. However, since the active code can perform arbitrary memory accesses, hybrid resource control cannot assume Click-like cache behavior from all extensions. Instead, it takes advantage of the fact that the worst-case memory access time of an extension can be limited by limiting the size of its working set.

To illustrate that the above is true for arbitrary code, let us look at Figure 3.2. The figure shows the results from a memory latency benchmark, called `mbench`, that makes a large number of strided accesses in a given working set. The graph plots the working set size of the program on the x-axis, versus the average number of cycles required for each write on the y-axis. Both axes are drawn in log scale. The benchmark was executed on an Intel Pentium III 850 MHz machine that had 16KB of L1 and 256 KB of L2 cache and L1 and L2 cache line sizes of 4 and 8 machine words respectively. Experiments with stride values of 8 and higher generate a worst-case memory access pattern for access latency because each successive memory access falls on a different L2 cache line. The results clearly show that if the working set of a program is restricted to the size of the L2 cache, even the worst-case memory access pattern will get an average memory latency of less than 10 cycles.

Hybrid resource control limits the size of the working set of an extension by limiting the amount of memory available to it. It then assumes that even in the worst-case scenario the active code's average memory access time will be less than that shown by `mbench` for the corresponding working set size. This reduces the amount of overestimation by static analysis.

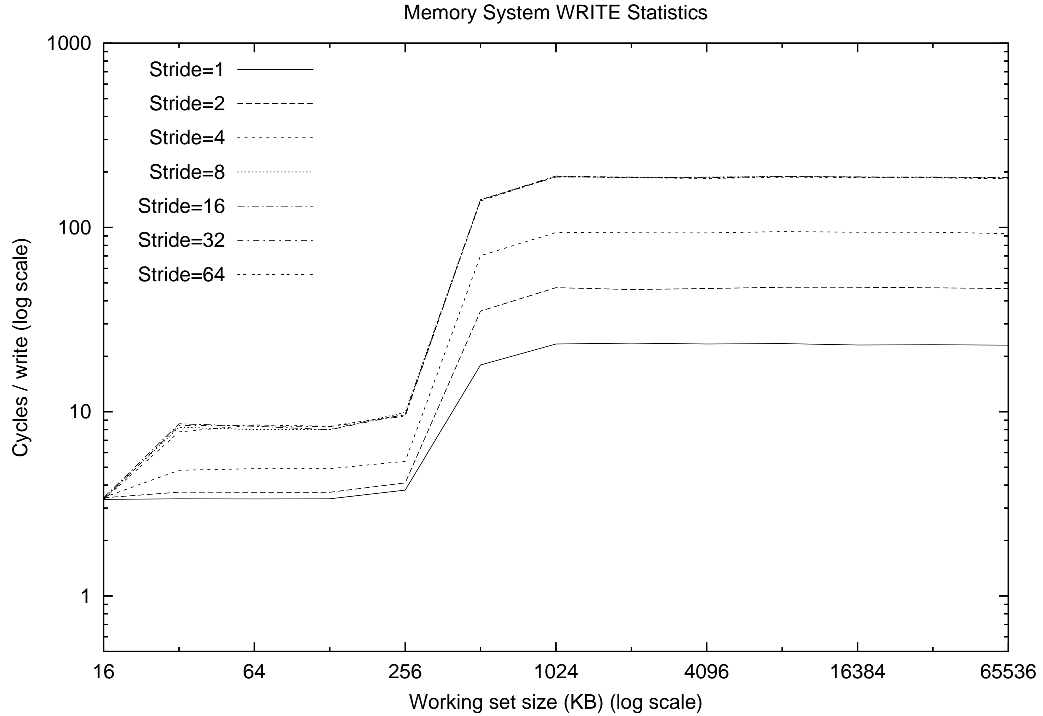

Figure 3.2. Results of a memory latency benchmark on an Pentium III 850 MHz machine with 16KB L1 and 256 KB L2 cache.

Note that it is possible for a program, whose working set is bound by the size of the L2 cache to have worse memory access time than that estimated by `mbench`. For example, this might happen if a program accesses each cache line only once. Hence all (or most) accesses would be from the main memory. However, due to the limited size of the working set, the damage due to such a memory access pattern would be limited. In other words, the execution time limits calculated by our static analysis might be violated. Therefore, we call them *soft* upper limits (as opposed to more conservative *hard* upper limits that can never be violated).

Despite making realistic assumptions about cache access, static analysis can result in a very high percentage of "false-positives." Since static analysis assumes the worst-case codepath, the system will overestimate resource bounds and reject extensions that are actually well-behaved. Hybrid resource control counters this by accepting extensions whose estimated resource bounds fall within "inflated-limits." The inflated-limits are calculated by multiplying the acceptable-limits with a pre-

determined constant, called "the inflation factor." However, the system ensures that the actual resource consumption of each admitted extension is within the acceptable limits by performing runtime resource accounting. Later in this chapter, we discuss how to choose a suitable inflation factor.

3.3 Runtime Accounting

The goal of runtime accounting is to ensure that each admitted extension's resource consumption is within the acceptable limits of the host environment. To achieve this goal, resource accounting measures the resources consumed by an extension for processing each packet. It then detects extensions that consistently violate the acceptable limits and unloads them. However, the extensions cannot be terminated asynchronously because of nonpreemptive scheduling. The system unloads a faulty extension when it is idle. (Note that all extensions terminate eventually because they are written in a resource-bounded language.)

Runtime accounting checks impose minimal overhead on the normal execution of extensions. For example, to measure the CPU time taken by an extension, we need to sample the system timestamp only before and after each invocation of the extension. In our measurements on a PIII 850 MHz machine, each reading of the timestamp costs only about 40 nanoseconds.

3.4 The Acceptable Limits

The acceptable limits for CPU usage on a node can be computed based on the maximum packet arrival rate supported on that node. For example, consider the simple extensible router shown in Figure 3.3. The router has two nonpreemptively scheduled extensions connected to 10Mbps full-duplex Ethernet ports on both input and output ends. A 10Mbps full-duplex Ethernet link has a minimum packet interarrival gap of 67.2 μsec (based on the fact that 10Mbps Ethernet supports 14,880 minimum-size packets per second [17]). Therefore, to prevent packet drops at the router input queue, the processing time for a minimum-size packet should not exceed 67.2 μsec. Similarly, processing of a maximum sized packet (1500 bytes)

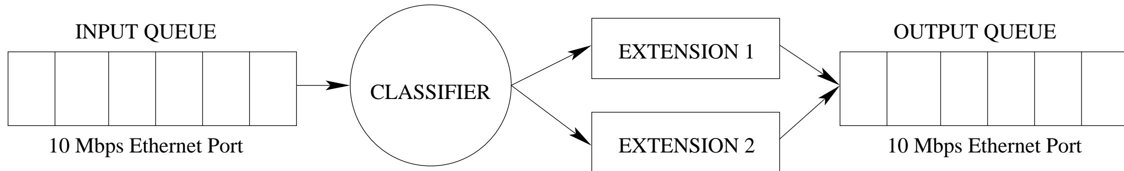

Figure 3.3. A system with two nonpreemptively scheduled extensions.

should not exceed 1.2 milliseconds (813 packets per second). These minimum packet inter-arrival times can be directly used as the acceptable CPU limits for this simple router. The calculation of acceptable limits for routers with multiple input/output queues can be performed by extending the above logic.

3.5 Choosing an Inflation Factor

The goal of hybrid resource control technique is to overcome the inflexibility of conservative static analysis while minimizing the cost of runtime checks. It achieves this goal by making realistic assumptions about the cache behavior of extensions and by inflating the acceptable limits. An inflation factor of N implies that extensions whose estimated resource bounds are N times greater than the acceptable limits are admitted into the system. While this decreases the false-positive rate of the system, unfortunately, it also increases the false-negative rate, i.e., some malicious or resource greedy extensions may be admitted into the system. For instance, an extension that always takes the worst-case codepath so that its actual resource usage is higher than the acceptable limits may be wrongly admitted. Hybrid resource control allows the users (e.g., node administrators) to limit the damage due to such false-negatives by choosing a suitable inflation-factor.

A suitable inflation factor for a router can be determined based on its tolerance to packet drops and the size of the queue at its input port(s). For example, consider a router that runs only two extensions—one trusted and one untrusted—and whose input device queue has a capacity of 64 packets. Clearly, to avoid packet drops, the untrusted extension's execution time should never exceed the arrival time of 64

packets, assuming that the extension starts execution on an empty queue. Therefore for this example, an inflation factor of 64 can tolerate one misbehavior for every 64 input packets, on average, without dropping any packets. Generalizing this rule of thumb, a system with an inflation factor of N will need M∗N slots in its device input queue to fully tolerate M back-to-back misbehaviors.

Note that hybrid resource control uses soft upper bounds, which may be violated in rare cases by malicious extensions. Therefore, even if a system chooses its inflation-factor as described above, packets may still be dropped at its input queue. To completely prevent packet drops, a system must calculate hard upper bounds using static analysis. However, hard upper bounds using static analysis are often overly conservative, which increases the false-positive rate of the system but provides increased protection against malicious extensions.

In practice, routers provide only best-effort services and dropping packets is not catastrophic. Second, a wrongly admitted extension is allowed to misbehave only once before runtime accounting detects it. Therefore, rarely dropping packets in favor of a lower false-positive ratio might be the right trade-off. So, a system may choose to use soft upper bounds and an inflation factor that tolerates very few misbehaviors.

3.6 Poll Points

The flexibility of the hybrid technique depends on the tightness of resource bounds: the rejection rate of legitimate code increases with the amount of pessimism in the estimation of resource bounds. Due to its inherent limitations, static analysis cannot be tight for all programs. Therefore, it may be desirable to admit code whose actual resource consumption is within acceptable limits but whose estimated resource bounds are greater than the inflated limits.

Hybrid resource control may admit such extensions by using internal buffer queues that are much larger than the device input queues. It inserts *poll points* in the extension code and divides the extension code into segments such that the estimated CPU of each segment is within the inflated limit. If the size of the

internal buffer queue is N times larger than the device input queue, the system can admit an extension that can be divided into N "admissible" segments by inserting N-1 poll points in its code. Of course, the system would still measure the execution time of the extension at runtime, and would unload it if the sum of the execution times of all its segments exceeds the acceptable limit.

Figure 3.4 shows the code that executes at a poll point. The system reads the clock, and if the extension is taking too long to execute, the extension is flagged to be in violation of the acceptable limit. Further, the network interfaces are polled and any packets in their input queues are stored in internal buffers to avoid packet drops. Conceptually, the poll points can be inserted anywhere in the code, for example, between two basic-blocks or at function entry/exit points. In the common case, when the extension is within the acceptable limit, the processing overhead of a poll point is very low. In our experiments on a Pentium III 850 MHz machine, this overhead is less than 100 nsec. The poll points let us trade off constraints

```
new_timestamp = poll_timestamp();
if (new_timestamp - begin_timestamp > acceptable_limit) {

    /*
     * Extension taking too long to execute
     */

     current_extension->violations++;

    /*
     * poll network interfaces
     * and store input packets in internal queues
     */
    poll_interfaces();

}
```

Figure 3.4. Code executed at a poll point.

due to pessimism of resource estimates with the overhead of runtime checks and internal buffer space.

CHAPTER 4

DESIGN OF RBCLICK

In this chapter, we present the design of Resource Bounded Click (RBClick), an active network environment for best-effort active extensions that implements hybrid resource control. To discuss RBClick in terms of established terminology, we use active networking terminology from the NodeOS specification [1] and Click terminology as defined in [19].

4.1 Overview

RBClick is an extension of the Click modular router toolkit [19] customized to support untrusted packet processors on a NodeOS. The RBClick toolkit contains a *trusted base* of Click elements. Untrusted users can add more functionality to it by supplying new elements written in a resource-bounded version of Cyclone, called RBCyclone (see Section 4.2). An RBClick-based active extension is simply a graph specified in the Click language that interconnects trusted and untrusted elements. However, RBClick configurations are resource bounded and therefore cannot have unbounded loops in them. Instead, users are required to bound loops in RBClick configurations using a special `Loop` element that enforces constant loop bounds.

Figure 4.1 shows an example active extension in RBClick. The elements with filled boxes are untrusted elements supplied by the user while all other elements are taken from the trusted base.

4.1.1 RBCyclone Elements

RBClick exports a Cyclone interface that provides equivalent functionality as the `Element` class and support libraries in Click. The interface includes essential services, such as functions to invoke timers, manipulate packets, and communicate

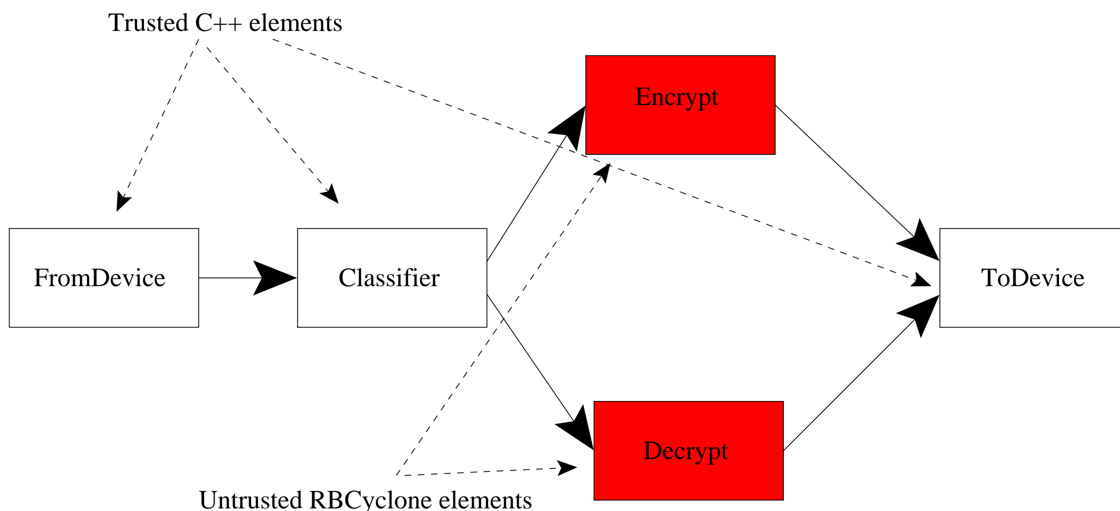

Figure 4.1. An example RBClick extension.

with other elements in the configuration. C++ and RBCyclone elements interact with each other using stub functions. These stub functions perform data marshaling and also prevent any Cyclone language exceptions from leaking into the trusted C++ code.

4.1.2 Deploying an Extension

Before an extension can be deployed a code analysis tool performs code analysis on all the untrusted elements in its configuration. The code analysis tool outputs the resource bounds of individual elements. The resource bounds for trusted elements are measured using a benchmark workload. The resource bounds for trusted and untrusted elements are fed to a graph analysis tool that calculates the overall resource bounds for a configuration. The configuration checking tool may also insert poll points in the configuration, as explained later in this chapter.

Figure 4.2 shows the interaction between Janos and RBClick. The RBClick environment runs as a trusted environment on Moab. It schedules and performs runtime accounting for all running extensions. Note that RBClick provides only

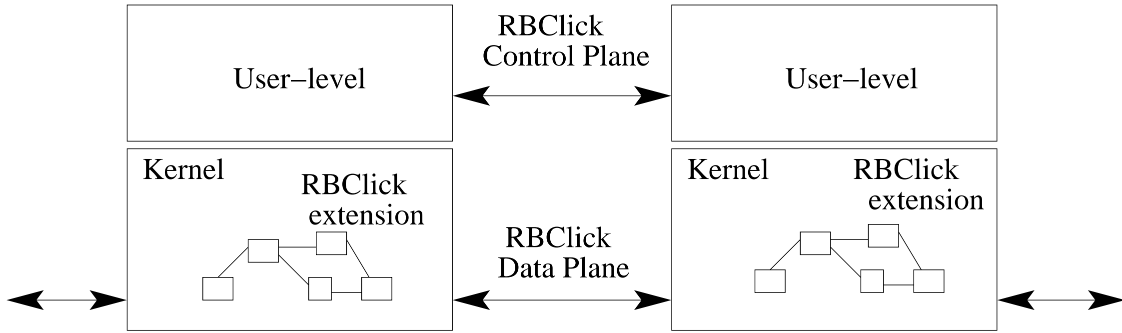

Figure 4.2. RBClick and Janos. The control plane for RBClick router configurations is implemented via the Bees EE on Moab.

datapath support, the active extensions are downloaded and installed using the control plane through the Bees EE.

In the following sections, we describe two key components involved in implementing hybrid resource control technique in RBClick: RBCyclone and code analysis of RBClick extension configurations.

4.2 RBCyclone

4.2.1 Why Cyclone?

Using active extensions technology with RBClick, users should be able to supply their own elements and extend the functionality available on a node. However, user-supplied elements cannot be trusted and must be executed in a memory and resource-safe environment.

Hardware-based virtual memory techniques can ensure complete memory safety from untrusted code, however, they seem too heavyweight for use with fast-path active extensions. Therefore, we decided to use relatively lightweight language-based technology. A comparison of software- and hardware-based memory safety techniques is shown in [30].

Interfacing Click to a high-level type-safe language, such as, Java and OCaml, seems to involve prohibitive performance costs. Overheads of interpretation, marshaling, and garbage collection are the dominant costs [2]. Therefore, we decided

to use a type-safe but C-like language, Cyclone [16]. The compatibility of Cyclone with C/C++ makes it easier and efficient to interface with Click.

RBCyclone is a resource-bounded subset of Cyclone that leverages the limited domain of active extensions. In the rest of this section, we discuss the domain-specific restrictions imposed by RBCyclone to ensure safety and resource boundedness.

4.2.2 Namespace Control

RBCyclone elements have access to only two external namespaces: RBClick_NS and CYC_NS. Untrusted elements cannot directly allocate or resize a Click packet. Instead, RBClick_NS provides restricted means to perform such operations. It also includes wrapper functions used by Click elements to make calls into RBCyclone elements, for example, to push or pull packets and invoke timer handlers. These wrapper functions handle any Cyclone language exceptions that leak through from RBCyclone code. The CYC_NS namespace provides a restricted standard Cyclone library, a Cyclone equivalent of the standard C-library. However, certain unsafe system calls which can be used to violate resource bounds, such as, *signal*, *malloc*, *new* and *exit* have been overridden to simply return error messages.

4.2.3 Restricted Programming Constructs

To limit the CPU cycles executed by untrusted code, we remove the following constructs from Cyclone: *goto*, *while*, normal *for* loops, *recursion*, and *function pointers*. The only iteration construct available in RBCyclone is a specialized *for* construct with the following syntax:

```
for (CONST) (... ; ... ; ...)  { /* Body of the for loop */ } /*
CONST is a compile time constant */
```

The above type of `for` loop is executed at most CONST times. CONST can be a symbolic constant whose value is substituted at the deployment node. Values from node-specific symbolic constants such as, PACKET_LEN_MAX, MTU, and DATA_LEN_MAX are maintained by the trusted RBCyclone compiler.

4.2.4 Memory Management

RBCyclone runtime manages memory using Cyclone's region-based memory management [13] and completely eliminates the need for garbage collection. As discussed in Section 2.1.2, Cyclone has three kinds of regions: a heap region that lives forever and is garbage collected, stack regions that correspond to local declarations, and dynamic regions that have programmer-controlled dynamic lifetimes, similar to memory areas allocated with `new` and `free`. RBCyclone has no explicit heap region and hence no garbage collection. Instead, based on our survey of networking code, we have defined the following four fixed regions in RBCyclone: `‘A`, `‘B`, `‘C`, and `‘D`. These regions have nested lifetimes: `‘A` < `‘B` < `‘C` < `‘D`. A reference in a region with longer lifetime cannot point to data in a region with shorter lifetime. For example, a reference allocated in `‘B` cannot point to data in `‘A`. The rationale for these regions is discussed below.

4.2.4.1 Region for per-packet memory (`‘A`)

This region has lifetime equal to the duration of processing a single packet. This memory region is available to all the elements of a domain while it does packet processing. Hence it can be used to share transient state among code elements. In Click such state is shared by allocating extra fields in the `Packet` object. Because it has the shortest lifetime, none of the other three regions can hold references pointing into this region.

4.2.4.2 Region for per-domain packet cache (`‘B`)

In RBClick, the memory area holding a packet is freed as soon as its processing is finished and hence a reference to it cannot be stored in persistent memory. Region `‘B` provides a fixed size array to cache packets without having to copy them to persistent memory. Packets can be stored and retrieved using *put* and *get* operations. This region is specifically designed for efficient storage of packets by packet caching applications, such as aggregated multicast [38]. This region is

allocated at domain creation time and its size is specified in an extension's RBClick configuration.

4.2.4.3 Region for per-domain memory (`'C`)

Region `'C` is used by a domain to keep a state that persists between packets. For example, an element could maintain flow state in this region. This region is allocated when a domain is created and destroyed when the domain is terminated. If an extension needs to recycle persistent memory, it can create and destroy dynamic regions in region `'C` as needed. However, region `'C` cannot be shared with other extensions.

4.2.4.4 Region for shared memory (`'D`)

This region is designated for memory that is shared between multiple domains, for example, memory for routing tables. The memory in this region can be obtained using special names for memory areas in this region. A standard filesystem-like interface can be used to access memory in region `'D`.

4.3 Static Analysis

Static analysis predicts resource upper bounds on active code. In RBClick, static analysis of an active extension is done in two stages: 1) code analysis of individual untrusted elements and 2) graph analysis of its RBClick extension configuration. As Galiter et al. noted [11], the resources consumed by a program, especially CPU time, depend significantly on local conditions on a node. Factors such as traffic patterns, execution time of system calls, machine speed, and other node-specific constants affect the resource bounds. Therefore, in RBClick, static analysis is done either at the deployment node or at a trusted site that knows the values of node-specific constants.

4.3.1 Code Analysis of an Untrusted Element

Designing a sophisticated code analysis tool to estimate resource bounds was not one of the goals of this thesis. RBClick's purpose is to demonstrate what advantages

can be gained given the values of resource bounds. However, to show the feasibility of such a tool, we have designed and partially implemented (see Section 5.1) a simple code analysis tool (CAT) that analyzes untrusted code. Currently, CAT works as follows.

An RBCyclone preprocessor validates RBCyclone code and generates valid Cyclone code. This Cyclone code is then compiled into C code by the Cyclone compiler. A loop annotation tool then analyzes the C code and generates annotations around the bodies of loops. The loop annotations also include the value of static upper bound on each loop. Similar annotations are also generated for those functions in the CYC_NS and RBClick_NS namespaces whose execution time depends on the value of its parameters, such as `memcpy`. This annotated C code is then compiled into assembly code using the `gcc` compiler. The annotated assembly code is then used by CAT's assembly level code analyzer to predict resource upper bounds on untrusted code.

CAT uses simplistic models to predict CPU usage. The basic idea behind CAT is to traverse all execution paths and collect instruction statistics along these paths. These instruction statistics are then used to generate bounds for CPU usage in the following manner. Note that RBCyclone manages the memory resource by dynamically limiting the size of memory regions.

CAT classifies all instructions as either register-only, memory reference, or function call operations. Each register-only instruction is assigned a fixed cost, typically 1 cycle. Memory references are assigned a fixed cost derived from the memory access latency benchmark for the target machine and the working set size of the extension, as explained in Section 3.2. Function call instructions are assigned a cost equal to the CPU resource bound of the called function. Function calls are either calls into untrusted code itself or library calls. For an active node, a benchmark is used to compute the cost of all library calls. A similar benchmark-based method to predict node specific CPU time for system calls is also used by V. Galtier et al. in [11].

4.3.2 Static Analysis of an Extension Configuration

CAT estimates the complete CPU usage of an extension by analyzing its RBClick extension configuration. RBClick configurations cannot have unbounded loops in them. For example, in Click [22], the IP forwarding router has unbounded loops in it and hence is not a valid RBClick extension configuration. However, bounded loops are allowed. A loop can be bounded by inserting a special `Loop` element at the loop join point. A `Loop` element takes a constant as its configuration parameter, which determines the maximum number of times a loop is traversed during the processing of a single packet.

With all configuration loops bounded, an RBClick configuration can be represented as a directed graph, where each <element, port> pair represents a node and each connection between ports represents an edge. Each edge gets its direction from the direction of packet flow (push vs. pull) that is assigned to it. Traversal of this directed graph with knowledge of static upper bounds on each element is used to find an upper bound for an active extension.

As mentioned in the Section 3.6, poll points may be used to reduce the false-positive ratio of hybrid resource control. In RBClick, this can be achieved by inserting a special `Poll` element at appropriate places in a configuration. The `Poll` element works as a poll point at function entry points in the packet processing code of an extension. The results from CAT are used to find the segments of a graph whose resource usage is within the inflated limits. These segments are then separated by inserting `Poll` elements between them.

CHAPTER 5

EVALUATION

Our goal is to evaluate the feasibility and the potential performance benefits of our approach. To this end, we have implemented a prototype of the design of RBClick discussed in Chapter 4. In this chapter, we first evaluate the feasibility of our approach by estimating execution times of existing Click elements and router configurations using CAT. We will see that for most elements, overestimation using CAT stays within a reasonable factor of 23 but for some elements it exceeds 100. Part of this excessive overestimation can be overcome using a sophisticated analysis tool, but part of it is inherent in static analysis. We will also evaluate the performance benefits obtainable with hybrid resource control in RBClick by comparing its performance with the native resource control scheme in Moab. We will show that, as compared to the dynamic resource control technique in Moab, hybrid resource control improves the performance of IP routing by up to a factor of two and benefits all the configurations we experimented with. These performance benefits directly correspond to the removal of context switch and system call overheads at the Moab user-kernel boundary that were discussed in Chapter 2.

5.1 Implementation

We have prototyped key parts of the hybrid resource control scheme in RBClick and its companion tools, RBCyclone, and CAT. Our system still lacks a number of features necessary for practical deployment, for example, Moab lacks dynamic code linking and loading and CAT lacks general control-flow analysis capabilities. The goal of this prototype is to demonstrate the feasibility of hybrid resource control and show potential performance benefits obtainable by using it. Our prototype achieves these goals despite being unready for practical deployment.

5.1.1 RBClick

RBClick prototype provides fast-path extensibility in Moab in Janos. Currently, RBClick instantiates each RBClick extension configuration in a special "lightweight" NodeOS domain [1]. A NodeOS domain is a resource container, similar to a process in a traditional UNIX-style operating system. All RBClick extension domains are managed using the hybrid resource control technique. Domains that run under hybrid resource control do not incur any resource control checks at the Moab user-kernel boundary.

5.1.2 CAT

We have implemented a rudimentary version of CAT. It consists of a set of code analysis tools that work on Cyclone and C source code and x86 assembly code to analyze an RBCyclone element and a graph analysis tool that works on the Click language to analyze a complete RBClick graph. For best results, these tools need to be integrated into an RBCyclone compiler, but for the initial prototype, we have implemented them as a set of simple tools written in `lex`, `yacc`, `C`, and `perl`. CAT works as explained in Section 4.3.

5.1.3 RBCyclone

Our RBCyclone prototype does not implement the four memory regions outlined in Section 4.2. Implementing these regions is conceptually simple and requires modifications to the type-checking system in the Cyclone compiler. As of this writing, we have not done these modifications. However, addition of these regions to the type-checking system in RBCyclone probably only affects static type-checking, and hence will have no effect on the performance numbers reported in this thesis. Another practical limitation of our RBCyclone prototype is incomplete fault-isolation from untrusted code. The default Cyclone compiler, as of version 0.5, does not prevent stack overflow and arithmetic errors like divide-by-zero. These errors could cause the Moab kernel to crash. Fortunately, it would be easy to build code analysis tool

to prevent these errors for the RBCyclone language because it cannot have recursion or unbounded loops in it. The OKE project [6, 14] has already built such a tool.

5.2 Feasibility

A key part of our approach is automatic estimation of soft upper bounds on the resource usage of extensions. In this section, we evaluate the effectiveness of our prototype implementation of CAT in doing that.

5.2.1 Criteria and Experimental Parameters

As discussed in Section 3.5, the inflation factor for a router can be derived from the size of its input port queue(s). It is common for current general-purpose operating systems to have an input port queue with 64 or more slots. For example, the network driver for the Intel EtherExpress Pro/100 network card in the Linux kernel version 2.4.18-27 uses 64 buffers each for device receive and transmit queues. This implies that the system can tolerate one complete misbehavior with inflation factor of 64, or two with inflation factor of 32, and so on. Since Moab is a dedicated router operating system, we use 128 receive and 32 transmit buffers in its network port queues. Hence, it can tolerate two misbehaviors with an inflation factor of 64.

To evaluate the feasibility of static analysis, we assume that our analysis is effective if it estimates resource bounds within a factor of 64 of the actual worst-input average case values. So an inflation factor of 64 can offset the effects of pessimism of static analysis, and result in an extremely low false-positive ratio.

As explained in Section 4.3, CAT analyzes code elements by classifying x86 assembly instructions into memory, register only, and function call instructions. It uses the average cost of a memory access from the `mbench` memory latency benchmark for the target platform based on the working set size of the extension. It then calculates the CPU usage of each function by adding up the cost of all register and memory instructions, system calls, and other called functions.

We conducted all our experiments in the Netbed network testbed facility [37]. All the machines used for the results presented in this chapter were Pentium III

850 MHz. All measurements were taken for max-sized Ethernet packets—1500 bytes. The results from `mbench` memory latency benchmark for this machine are shown in Table 5.1. Table 5.2 summarizes all the parameters used by CAT in these experiments.

5.2.2 Elements of the IP Router

We analyzed all 13 elements involved in the standard IP router configuration in Click [19]. We manually annotated all the loops with their static bounds in the C++ code of these elements, and performed analysis on them using CAT. The results are shown in Table 5.3. As we see from the table, except for the bottom two elements ARPQuerier and IPGWOption, all elements are estimated within a factor of 23.

Table 5.1. Memory latency table used in experiments.

Working Set Size (KB)	Write Latency (Cycles)
16	2.3
32	10
64	10
128	10
256	10
512	80
1024	200
2048	200
4096	200
8192	200

Table 5.2. Parameters used by CAT.

Parameter	Value
Working Set Size	64 KB
Memory Latency	10 cycles
Input port queue size	128 packets
Inflation factor	64

Table 5.3. CPU estimation of the elements of standard IP graph by CAT.

Element Name	Estimated time (CPU Cycles)	Measured Time (CPU Cycles)	Ratio
Paint	54	46	1.17
Strip	488	350	1.40
GetIPAddress	586	146	4.01
IPFragmenter	313	68	4.61
FixIPSrc	398	71	5.60
DecIPTTL	1088	139	7.83
DropBroadcasts	919	107	8.59
PaintTee	1360	118	11.52
CheckIPHeader	6075	342	17.77
LookupIPRoute	2919	141	20.70
Classifier	7276	319	22.81
ARPQuerier	38499	360	106.00
IPGWOptions	69653	350	199.00

The ARPQuerier element takes an IP packet and sends it to its next hop after making any ARP requests, if needed. The IPGWOptions element processes any IP options present in packets that pass through it.

Part of the overestimation that is most visible in the ARPQuerier element is due to the fact that CAT does not understand program control flow other than loops. Therefore, CAT always adds up the cost of even mutually exclusive code paths instead of taking only the worse of the two. For example, CAT always adds up the cost of the "then" part and the "else" part of a standard "**if** (condition) {then-body} **else** {else-body}" statement. (Making CAT recognize these mutually exclusive codepaths is straightforward but requires changes to the `gcc` compiler. As CAT's primary use is on RBCyclone elements, we have not done these changes in gcc.)

The ARPQuerier element has an if-then-else statement right at the top-level function in its call-graph, as shown in Figure 5.1. Each of the functions `handle_ip` and `handle_response` accounts for half of the estimated CPU cycles. CAT just adds up the cost of both parts of the if-then-else statement shown the figure. A

```
void
ARPQuerier::push(int port, Packet *p)
{
  if (port == 0)
    handle_ip(p);
  else {
    handle_response(p);
    p->kill();
  }
}
```

Figure 5.1. The code for the top-level push function of the ARPQuerier element.

smarter code analysis will report at most half as many cycles as reported by CAT and estimate ARPQuerier within a factor of 53.

Another part of the overestimation is due to an inherent limitation of static analysis—it always assumes the worst code path. The ARPQuerier element caches ARP responses from the network. Therefore, in the common-case the ARP request is satisfied from the cache and only the shortest code path is taken. Similarly, excessive overestimation for the IPGWOptions element results from the fact that in our experiments no packets contained any IP options. Therefore, the measured time does not include the cost of IP options processing but only includes the shortest code path in the element. CAT estimates the shortest code path within a factor of two.

In general, for elements with a large difference between the common case path and the worst-case path, static analysis will always report much higher resource usage than what will be observed in practice. One way to counter this limitation is to require the programmers to annotate the common-case code path. The code maybe admitted on the basis of estimations of the common-case code path, and the system could insert poll points in the worst-case code path and runtime accounting could ensure that only the common-case code path is taken on an average.

Another option is to split such elements in such a way that the worst-case code is isolated in one element and the common-case in another. Users can then include

only the common-case element in the nonpreemptive processing path and defer the execution of the worst-case code segment. As a matter of fact, most commercial routers defer IP options processing to a slow path. Such a split of the IPGWOptions element would bring down the ratio of estimated time to observed time to within a factor of two.

5.2.3 RBCyclone Elements

To further evaluate the feasibility of our approach, we wrote three elements in RBCyclone—two highly CPU intensive, DES and ActiveDoom, and one simple element that operates only on the IP header of a packet, ECN. DES is an implementation of the common Data Encryption Standard algorithm triple-DES. ActiveDoom is an in-network packet aggregation program for the game of Doom . The ECN element sets the congestion signal bits [27] in the IP header of all input packets sent to it by a queue. It also updates the IP checksum. The results for these elements are given in Table 5.4. As seen from the table, CAT estimates the CPU requirements of these elements reasonably well, within a factor of 22. The CPU estimate of the smaller ECN element is within a factor of two.

5.2.4 Analysis of Router Configurations

We analyzed four configurations using CAT. Since the IP router is part of the trusted computing base, for all elements of the IP router configuration we take the measured value of CPU usage from Table 5.3, as opposed to the values estimated by CAT. Of course, CPU usage for untrusted RBCyclone elements is estimated using CAT. CAT's estimates for the configurations are presented in Table 5.5. The table

Table 5.4. CPU estimation of RBCyclone elements by CAT.

RBCyclone Element	Estimated time (CPU Cycles)	Measured Time (CPU Cycles)	Ratio
DES	12654225	556110	20
ActiveDoom	728075	32584	22
ECN	3189	1964	1.6

Table 5.5. CPU estimation of router configurations by CAT.

Router Configuration	Estimated time (Cycles)	Measured Time (Cycles)	Ratio Ratio
IP Router	14765	9257	1.6
IP Router with ECN	17954	11221	1.6
IP Router with DES	12668990	570430	22
ActiveDoom	728075	32584	22

demonstrates that using our tools and a conservative 90% cache hit-rate, we are able to estimate the running time of these configurations within a factor of 22.

5.2.5 Summary

These results show that despite being rudimentary and experimental, CAT's overestimations, with the exception of IPGWOptions and ARPQuerier elements, can be accommodated within an inflation-factor of 25. These results clearly demonstrate the feasibility of our approach for many classes of best-effort router services. Our approach works particularly well for new configurations that combine a relatively small amount of new functionality with a number of trusted Click elements, as demonstrated by the ECN configuration.

We also note that our techniques do not guarantee that the extensions will never consume more CPU than estimated. For example, an extension may consume more than the estimated CPU if its cache behavior is worse than that predicted by the memory latency benchmark. We rely on runtime accounting to quickly detect these cases and unload the faulty extensions.

5.3 Performance

In this section, we evaluate the potential performance benefits of the hybrid resource control by measuring its performance with the dynamic resource control scheme in Moab.

5.3.1 Experimental Setup

All experiments were performed on the cluster portion of the Netbed network testbed [37]. All machines were 850MHz Intel Pentium IIIs with 512 MB of SDRAM and five Intel EtherExpress Pro/100+ PCI Ethernet cards. All experiments that used IP forwarding (IP router and DES) used a simple three-node setup where an active node running the corresponding router code interposes between a sender and a receiver. ActiveDoom experiments involved two ActiveDoom servers directly connected to each other and each server in turn connected to 5 FakeDoom clients. FakeDoom is a program that generates traffic similar to that generated by the real Doom program. Doom clients refresh game state at a rate of 30 times per second. All experiments run the same code under two resource control schemes: Moab's dynamic resource control and RBClick's hybrid resource control.

5.3.2 IP Router

To estimate the improvement in forwarding rate obtainable with the hybrid resource control scheme, we ran IP forwarding code in three configurations illustrated in Figure 5.2. *In-kernel IP−−* is a hardwired implementation of a minimal IP router in the Moab kernel. *Dynamic IP* a Click configuration that implements nearly all standards-compliant IP forwarding functionality [4], see [19] for details of how it works. It runs at user-level under Moab's dynamic resource control. *Hybrid IP* is an RBClick configuration which implements the same functionality as *Dynamic IP*. (This configuration contains extra `Loop` elements.) It runs under hybrid resource control .

As we see from Figure 5.3, IP forwarding under the control of the hybrid resource control technique (*Hybrid IP*) performs much better than with purely dynamic resource control (*Dynamic IP*). This improvement is due to the avoidance of context switching and reduced runtime checks at the user-kernel boundary. Note that *Dynamic IP* and *Hybrid IP* both do not include any MMU-imposed overhead. Therefore, the performance improvement in *Hybrid IP* over *Dynamic IP* is entirely

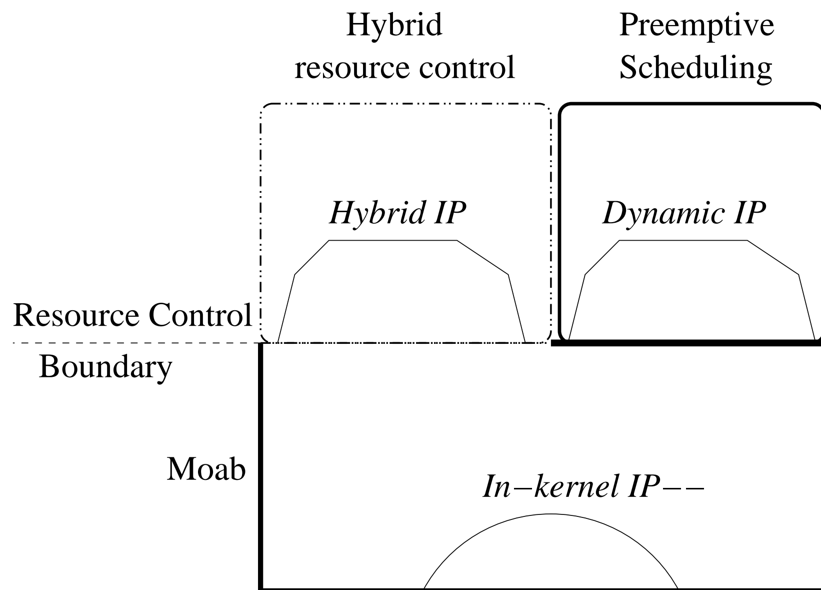

Figure 5.2. Experimental configurations of the IP router experiment.

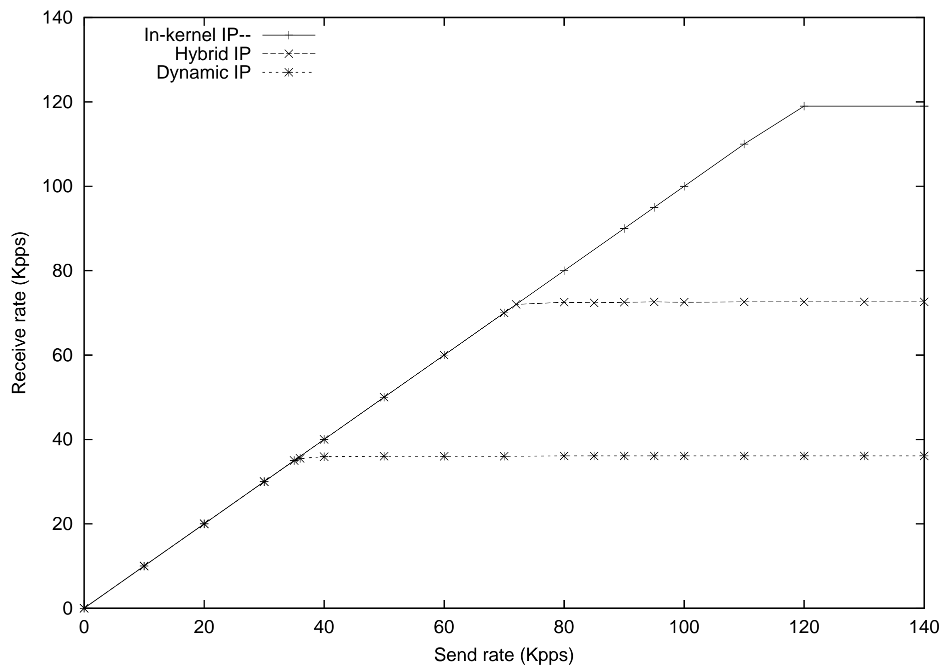

Figure 5.3. Performance of IP forwarding in three configurations.

due to the reduction in overhead of dynamic resource checks at the user-kernel boundary.

The performance difference between *Hybrid IP* and *In-kernel IP−−* results from two factors. First, the in-kernel version does not implement the full functionality of an IP router. Therefore, it performs fewer checks during IP forwarding. Second, the cost of creating a C++ object is high because of an untuned implementation of the memory management library in the OSKit. In our measurements, each object allocation takes about 450 CPU cycles and each deallocation takes 350 cycles. Therefore, the cost of creating and deleting a Click `Packet` object from a Moab packet buffer is also high. Creating a Click `Packet` does not involve data copying, but it does involve creating a new C++ object and assigning correct buffer pointers in it. Similar costs occur for all C++ objects. The in-kernel IP−− implementation does not involve any memory allocation (and deallocation) on the heap.

5.3.3 Overhead of Cyclone Interface

Figure 5.4 shows the performance difference between an IP router entirely in C++ (*C++ IP*) and an IP router in which one `Null` C++ element has been replaced by an equivalent RBCyclone element (*Cyc IP*). `Null` elements immediately push out the packets they receive. This configuration measures the overhead of boundary-crossing from trusted elements in C++ to untrusted elements in RBCyclone.

As we see from the graph, the performance difference between *C++ IP* and *Cyc IP* configurations is not significant. The boundary-crossing overhead from a C++ element to an RBCyclone element and back is less than 0.6 μs on the machines used in these experiments. Note that this overhead includes marshaling a packet to send it to RBCyclone code and unmarshaling it after receiving it back. The relatively low overhead of marshaling, unmarshaling, and boundary-crossing suggests Cyclone is an efficient type-safe extension language for the C++-based Click.

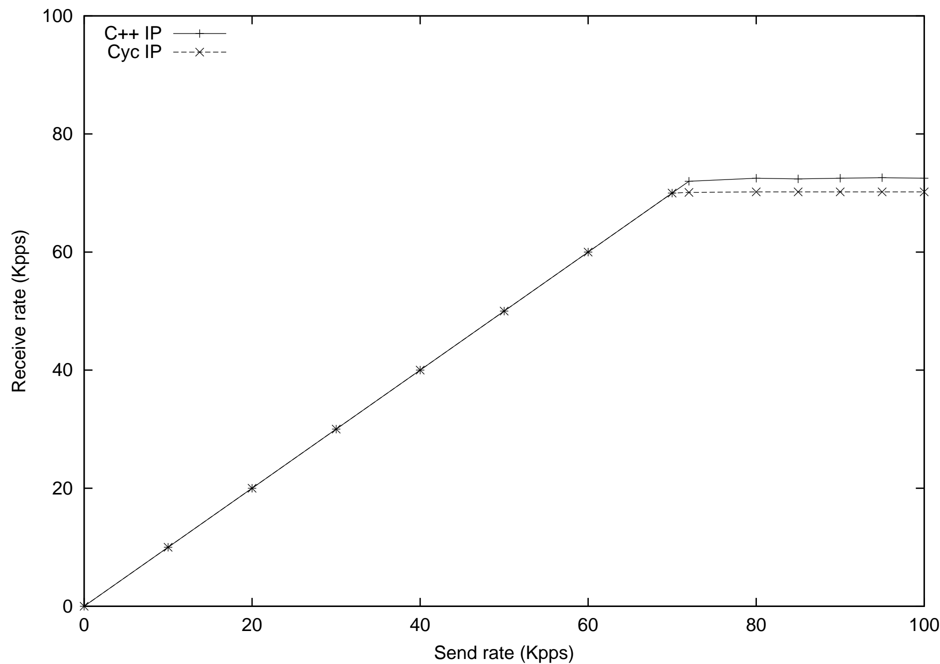

Figure 5.4. Performance of IP forwarding in two configurations.

5.3.4 DES

Figure 5.5 shows the performance of DES under Moab's native resource control scheme and under hybrid resource control using RBClick. The performance benefits due to hybrid resource control are more visible for smaller payload sizes. For bigger payloads, the cost of DES processing becomes significant, and the improvements due to hybrid resource control do not have much effect on packet forwarding rate.

5.3.5 ActiveDoom

Table 5.6 shows the performance of an ActiveDoom node under Moab's dynamic resource control scheme (Dynamic ActiveDoom) and under hybrid resource control scheme (Hybrid ActiveDoom). The results show that the benefits due to hybrid resource control are marginal because of the excessive processing cost of ActiveDoom.

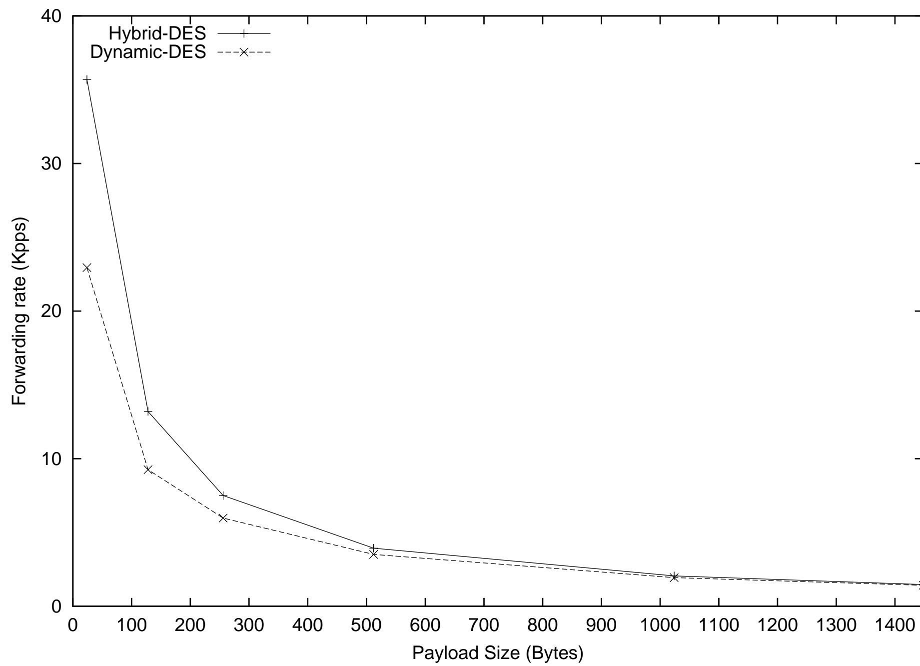

Figure 5.5. IP forwarding with DES

Table 5.6. ActiveDoom

Dynamic ActiveDoom (Cycles)	Hybrid ActiveDoom (Cycles)
36273	32584

5.3.6 ECN

Table 5.7 shows the performance of an IP router with ECN under Moab's dynamic resource control scheme (Dynamic ECN) and under hybrid resource control scheme (Hybrid ECN). The results show that the hybrid resource control benefits the ECN configuration by 41% over the dynamic resource control scheme.

Table 5.7. IP router with ECN

Dynamic ECN (Cycles)	Hybrid ECN (Cycles)
31781	18802

CHAPTER 6

RELATED WORK

Our work is most closely related to the PLAN [15] and SNAP [21] languages from the SwitchWare project at the University of Pennsylvania. Both PLAN and its successor SNAP are domain-specific languages designed to bound the resource consumption of active code. The resource bounding restrictions we impose in RBCyclone are similar to those in PLAN (Section 4.2). However, their work differs from ours in the following ways.

First, the SwitchWare project focused only on designing a resource-bounded language. They did not explore how conservative the statically computed bounds can be and how to cope with that pessimism. Our work complements theirs, focusing on combining conservative static estimates with dynamic checks to build a low-overhead execution environment. Second, we focus on the domain of fast-path active extensions that are deployed using the control channel, while PLAN and SNAP are both designed to be deployed directly in data packets. Third, we use a familiar C-like programming language, Cyclone [16], and impose restrictions that are necessary to bound resource consumption, while PLAN and SNAP are completely new domain-specific languages. Fourth, we examine a flexible set of existing networking components to show by example that the restrictions we impose in RBCyclone are not overly constraining (Section 5). It is not entirely clear whether PLAN could easily support all of these examples.

Other closely related efforts are the Open Kernel Environment (OKE) [6] and the "OKE Corral" [14]. The OKE is a safe execution environment in the Linux kernel, designed to run untrusted user extensions written in Cyclone. The OKE Corral is an active network environment based on the OKE and, like RBClick, draws heavily

from the Click modular router. However, the resource control techniques in OKE are purely dynamic and rely on asynchronous termination to enforce resource limits. In contrast, our goals for the hybrid technique have been to avoid asynchronous termination, taking advantage of statically-predicted resource bounds augmented by dynamic checks.

Proof Carrying Code (PCC) is a novel technique in which untrusted code carries an efficiently checkable proof of its resource boundedness. PCC can be quite effective in minimizing the overhead of runtime checks [23]. However, currently, PCC is practical only for small programs.

Our implementation was done in the context of Janos, an active network node operating system [32] that currently supports only Java-based active applications written for the ANTS2 and Bees environments [29]. Java-based active applications in Janos are much slower compared to the fast-path in the kernel. RBClick adds safe fast-path active networking to Janos.

Similar to RBClick in Janos, many other active networking systems provide extensibility close to the in-kernel fast-path [14, 20, 8]. A common differentiator between these and our work is that all of these systems use purely dynamic techniques for resource control.

Outside of the domain of active networking, hybrid resource control is similar to low-overhead message passing using Active Messages [34]. An Active Message carries the address of a remote program that is executed nonpreemptively as a message handler on the stack of the message-receiving thread. Since an Active Message is not allocated a thread of its own, it must not block and it must complete execution in a short period of time. Hybrid resource control uses the exact same principle to efficiently run active code. The important difference between the two is that hybrid resource control is designed for untrusted mobile code. Therefore, it has to automatically infer and enforce the resource bounds to ensure safety and security. In Active Messages, on the other hand, there is no mobile code and the programmer is trusted to obey the restrictions placed of the target execution environment.

Optimistic Active Messages (OAMs) [35] improve upon Active Messages and provide low-overhead message passing without any programming restrictions. In OAMs, the system optimistically schedules arbitrary active message code on the stack of the message-receiving thread. If the message handler tries to block, or takes too long to execute, it is spawned in a separate thread. So, OAMs not only have the advantage of low overhead for short programs, same as Active Messages and hybrid resource control, but they also support a general purpose programming model. The principles of OAMs may help relax the constrained programming model of hybrid resource control.

Extensible operating systems, such as the SPIN operating system [5], allow application-specific extensions directly in the kernel. Similar to RBClick in Moab, the SPIN operating system relies on a type-safe language to achieve safety from user extensions. (SPIN and its extensions are written in Modula-3, a type-safe language.) However, SPIN uses dynamic resource control, a round-robin preemptive CPU scheduler, to manage the resources available to user extensions.

In [11], the authors present techniques for controlling and predicting CPU use of active code in networks of heterogeneous nodes. Their techniques for predicting CPU usage could be used in conjunction with hybrid resource control. However, their resource control techniques require precise resource bounds to allocate resources, while hybrid resource control only needs approximate estimates of resource upper bounds to admit code.

Finally, in recent work on supporting real-time applications on general-purpose operating systems [12] the authors propose a mechanism that uses high-precision timers to schedule applications at a fine granularity. Such a mechanism could also be used by hybrid resource control to control the CPU resource. However, fine-grained timers do not avoid asynchronous termination or the overheads due to the kernel-user boundary.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

In this thesis, we have presented hybrid resource control, a resource control technique for fast-path active extensions. Hybrid resource control estimates CPU usage upper bounds on extensions by using static analysis and making realistic assumptions about their cache behavior. The guarantees provided by upper bounds help reduce the overhead due to runtime checks and avoid asynchronous termination of active extensions. We also presented the design of RBClick and its companion tools RBCyclone and CAT. Our prototype implementation of RBClick uses the hybrid resource control technique to control the resources consumed by untrusted user extensions in the Janos kernel.

To facilitate memory protection and enable static analysis of code, RBClick uses RBCyclone, a resource bounded variant of Cyclone. We have shown that the restrictions imposed by RBCyclone still offer a flexible programming model by examining a version of Click and showing that all its elements can be written with the restrictions of RBCyclone.

Our analysis of existing and new elements shows that hybrid resource control can be successfully used on a wide variety of extensions. Our measurements of forwarding rates show that the hybrid technique can help improve the forwarding rate of active extensions by up to a factor of two, compared to the purely dynamic resource control technique in Janos. However, the performance improvement does not result in significant throughput benefits for CPU-intensive extensions, such as triple-DES encryption.

Our focus in this thesis has been to show the feasibility and potential performance benefits of the hybrid approach. Our code analysis tool is very crude and

does not understand simple control-flow of programs. In future work, all the tools in CAT and RBCyclone preprocessor can be combined into an RBCyclone compiler derived from the Cyclone compiler. The RBCyclone compiler can use compiler optimization techniques to tighten the estimated resource bounds.

Another complementary option is to explore runtime a measurements-based techniques to estimate tighter resource bounds on untrusted code, as used in [9, 33]. One major difficulty with such measurements-based techniques is that of predicting a typical workload for active code. In general, this is impossible to do. However, by using user-supplied workloads and combining statistical analysis with control-flow analysis, it may be possible to predict tight resource upper bounds with very high probability.

APPENDIX A

ANALYSIS OF CLICK EXTENSIONS

We manually studied the source code of all 234 elements in Click version 1.2.1[1] to determine the fraction of its elements that were resource-bounded. Based on their potential resource usage, we classified the elements into the following seven categories:

1. **E1:** Resource usage *Constant*
2. **E2:** Resource usage *Proportional to the length of the packet*
3. **E3:** Resource usage *Proportional to some protocol header length*, e.g., CheckIPHeader consumes resources proportional to the IP header length.
4. **E4:** Resource usage *Proportional to the length of the configuration of an element*, e.g., the size of the Static routing table in LookupIPRoute.
5. **E5:** Resource usage *Proportional to some value in the configuration of an element*, e.g., the Tee element gets the number of outputs from its configuration.
6. **E6:** Resource usage *Proportional to some field in a protocol header*, e.g., the ICMPError element consumes resources proportional to the IP hlen header field.
7. **E7:** Resource usage *Potentially resource-unbounded*, e.g., the ARP element searches through a data structure whose length is determined by the number

[1] We studied the Click version current at the time this work began, version 1.2.1, released June 2001. The current version, 1.2.4, released May 2002, has 252 elements.

of packets it has seen in the last 5 min. Such elements are considered to be potentially resource-unbounded.

Clearly, elements in categories E1–E4 are bounded in their resource usage. Elements in E5 are also bounded because we can compute the bounds at configuration time. Elements in category E6 can be bound by enforcing an upper bound on the value of the corresponding header field, e.g., the length of the IP header can be constrained to be always less than 64 bytes. Therefore, elements in categories E1–E6 are resource-bounded and elements in E7 are potentially resource-unbounded.

Table A.1 shows the distribution of 234 Click elements into various categories. As seen from the table, only 23 elements (9.83%) fall into category E7 and hence could potentially consume unbounded amount of resources. The remaining 90% of the elements consume bounded amount of resources. The 23 elements in E7 cannot be bound because they use unbounded data structures, like linked lists and open hash tables. We could convert these elements to category E5 by re-coding the corresponding data structures to have a configurable maximum number of data items, which would make all 234 elements statically resource-bounded.

This study indicates that most fast-path extensions consume bounded resources. Another important implication of the above study is that fast-path extensions can be coded in a programming language in which all loops have static upper bounds.

Table A.1. Classification of Click elements

Category	Number	%age
E1	114	48.72%
E2	33	14.1%
E3	15	6.41%
E4	37	15.80%
E5	4	1.71%
E6	8	3.42%
E7	23	9.83%
Total	234	100%

REFERENCES

[1] Active Network NodeOS Working Group. NodeOS interface specification. Available as http://www.cs.princeton.edu/nsg/papers/nodeos.ps, Jan. 2000.

[2] D. S. Alexander, K. G. Anagnostakis, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. The Price of Safety in an Active Network. *Journal of Communications and Networks*, 3(1):4–18, Mar. 2001.

[3] G. Back. *Isolation, Resource Management and Sharing in the KaffeOS Java Runtime System*. PhD thesis, University of Utah, May 2002.

[4] F. Baker. Requirements for IP Version 4 Routers. *RFC 1812*, June 1995.

[5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Dec. 1995.

[6] H. Bos and B. Samwel. Safe Kernel Programming in the OKE. In *Proceedings of the Fifth IEEE Conference on Open Architectures and Network Programming (OPENARCH 2002)*, June 2002.

[7] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, Seattle, WA, June 1997.

[8] D. Decasper, Z. Dittia, G. M. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *ACM SIGCOMM*, pages 229–240, Sept. 1998.

[9] S. Edgar and A. Burns. Statistical Analysis of WCET for Scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 2001.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, Oct. 1997.

[11] V. Galtier, K. Mills, Y. Carlinet, S. Bush, and A. Kulkarni. Predicting and Controlling Resource Usage in a Heterogeneous Active Network. In *Proceedings of the Third International Workshop on Active Middleware Services*, pages 35–44. IEEE Computer Society, Aug. 2001.

[12] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting Time-Sensitive Applications on a Commodity OS. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*. Usenix, Dec. 2002.

[13] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based Memory Management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.

[14] H. Bos and B. Samwel. The OKE Corral: Code Organisation and Reconfiguration at Runtime using Active Linking. In *International Workshop on Active Networks*, Dec. 2002.

[15] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93, 1998.

[16] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Chene, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[17] S. Karlin and L. Peterson. Maximum Packet Rates for Full-Duplex Ethernet. Technical Report TR-645-02, Princeton University, Feb. 2002.

[18] E. Kohler. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, Feb. 2001.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[20] P. Menage. RCANE: A Resource Controlled Framework for Active Network Services. In *The First International Working Conference on Active Networks (IWAN '99)*, volume 1653, pages 25–36, June 1999.

[21] J. Moore and S. Nettles. Practical Programmable Packets. In *Proceedings of the 20th Conference on Computer Communications (INFOCOM)*, Apr. 2001.

[22] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231, Dec. 1999.

[23] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Oct. 1996.

[24] P. Patel, D. Wetherall, J. Lepreau, and A. Whitaker. TCP Meets Mobile Code. In *Proc. of the Ninth Workshop on Hot Topics in Operating Systems*. IEEE Computer Society, May 2003.

[25] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading Transport Protocols using Untrusted Mobile Code. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM, Oct. 2003.

[26] L. Peterson, Y. Gottlieb, S. Schwab, S. Rho, M. Hibler, P. Tullmann, J. Lepreau, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, Mar. 2001.

[27] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *RFC 3168*, Sept. 2001.

[28] J. M. Smith, K. Calvert, S. Murphy, H. K. Orman, and L. L. Peterson. Activating Networks: A Progress Report. *IEEE Computer*, 32(4):32–41, Apr. 1999.

[29] T. Stack, E. Eide, and J. Lepreau. Bees: A Secure, Resource-Controlled, Java-Based Execution Environment. In *Proceedings of the Sixth IEEE Conference on Open Architectures and Network Programming (OpenArch 2003)*, Apr. 2003.

[30] M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: an architecture for reliable device drivers. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Sept. 2002.

[31] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[32] P. Tullmann, M. Hibler, and J. Lepreau. Janos: A Java-Oriented OS for Active Network Nodes. *IEEE Journal on Selected Areas in Communications*, 19(3):501–510, Mar. 2001.

[33] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[34] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[35] D. A. Wallach, W. C. Hsieh, K. Johnson, M. F. Kaashoek, and W. E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[36] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of the First IEEE Conference on Open Architectures and Network Programming (OpenArch 1998)*, Apr. 1998.

[37] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.

[38] T. Wolf and S. Choi. Aggregated Hierarchical Multicast for Active Networks. In *Proceedings of the IEEE Military Communications Conference (MILCOM)*, Oct. 2001.