Hybrid Resource Control of Active Extensions

Parveen Patel Jay Lepreau
University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205

{ppatel,lepreau}@cs.utah.edu http://www.cs.utah.edu/flux/

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 node 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 paper 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. This hybrid solution reduces runtime overhead and avoids the problem of asynchronous termination by delaying extension termination until times when no extension code is running, i.e., between processing of packets.**

This paper also presents the design and initial implementation of the key parts of a hybrid resource control technique, called RBClick. RBClick is an extension of the Click modular router, customized for active networking in Janos, an active network operating system. RBClick uses a modified version of Cyclone, a type-safe version of C, to allow users to download new router extensions directly into the Janos kernel. Our measurements of forwarding rates indicate that hybrid resource control can improve the performance of router extensions by up to a factor of two.

I. Introduction

Active network technology allows users of a network to customize the computation performed at routers using mobile code. A data packet in an active network carries the mobile code or information that allows demultiplexing to the mobile code that should process the packet. This flexibility of selecting packet processing code 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 paper addresses the issue of resource control of active extensions in *discrete* or *control-plane* active networking [1]. 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. For example, an extension implementing application-level gateway functionality could be installed into the kernel to process packets at high speed, if 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 environment and runtime checks are performed to monitor its resource usage. On detection of misbehavior, misbehaving active code is asynchronously terminated. Asynchronous termination can affect the integrity of data structures shared by multiple extensions. More importantly, such termination requires what is effectively a user-kernel boundary (implemented with or without the hardware MMU) to protect the integrity of any data structures shared between untrusted code and trusted "kernel" code [2]. The checks required to implement such a boundary have significant cost, as shown in this paper and elsewhere [3].

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. Difficulty in modeling complex features of modern computers, such as multiple-issue, pipelining, and caching, also add to the degree of pessimism. In our measurements, ignoring the effects of caching alone can cause analysis of x86 assembly code to be pessimistic by up to a factor of 20.

Therefore, it is hard to choose between the two; sandboxing incurs runtime overhead and causes asynchronous termination, while static analysis is very conservative.

This paper proposes *hybrid resource control*, a resource control mechanism that uses a combination of *static analysis* and *runtime accounting*. Static analysis techniques are used to statically predict resource upper bounds of active code. These resource bounds are then used to control admission to the lightly protected kernel execution environment, guarantee termination, and reduce the overhead from 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.

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, under agreement F30602–99–1–0503. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

We have developed a prototype of the key parts of hybrid resource control in an environment for active extensions, called *RBClick*, "Resource Bounded Click." RBClick is an extension of the Click modular router [4], implemented in Janos, an active network operating system [5] 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 a resource-bounded variant of Cyclone [6], a type-safe version of C, called RBCyclone. RBClick estimates resource bounds on active extensions; acceptable extensions are then loaded into the Janos kernel.

This paper evaluates the usefulness of RBClick along two dimensions: programming flexibility 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 could be written with the resource bounding restrictions of RBCyclone. To estimate the performance improvement from reducing the overhead of runtime checks, we compare RBClick to resource sandboxing of unmodified Click. We found that 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 paper are:

- A proposal for hybrid resource control (Section III).
- An initial design and implementation of a hybrid resource control technique in RBClick, an extension of the Click modular router (Section IV).
- A preliminary evaluation of RBClick (Section V).

II. Related work

Our work is most closely related to the PLAN [7] and SNAP [8] 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 IV-C). However, their work differs from ours in the following ways.

First, the SwitchWare project focused only on designing a resource-bounded language but 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 [6], 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 V). It is not entirely clear whether PLAN could easily support all of these examples.

Another set of closely related efforts are the Open Kernel Environment (OKE) [9] and the "OKE Corral" [10]. 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 [11]. However, currently, PCC is practical only for small programs.

Our implementation was done in the context of Janos, an active network node operating system [5] that currently supports only Java-based active applications written for the ANTS2 and Bees environments [12]. 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 [10], [13], [14]. A common differentiator between these and our work is that all of these systems use purely dynamic techniques for resource control.

Finally, in recent CPU-modeling work [15] 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.

III. Hybrid Resource Control

In this section, we present *hybrid resource control*, a resource control technique for active node operating systems that uses a combination of *static analysis* and *runtime resource accounting* to efficiently control the resources consumed by *best-effort* active extensions. By best-effort we refer to extensions that do not require any QoS guarantees.

Hybrid resource control uses static analysis to infer conservative static upper bounds on resource requirements of active extensions. These resource upper bounds are used to control admission to the loosely protected, high-speed extension execution environment in the kernel. Static upper bounds on the CPU time ensure that only extensions that are guaranteed to terminate in a reasonable amount of time are admitted. This guarantee helps to reduce runtime checks and avoid undesirable forced termination of active extensions.

However, static resource analysis is conservative in nature. Therefore, completely relying on static upper bounds for admission can be very constraining, i.e., the system will reject extensions that would actually consume resource within the acceptable limits. Therefore, hybrid resource control admits some over-limit extensions but performs low-overhead runtime resource accounting to enforce acceptable behavior. The system admits extensions whose resource bounds fall within a predetermined constant factor of the acceptable limits and lets them run without interruptions. However, the system performs resource accounting at appropriate times to ensure that extensions are behaving correctly. If an extension is found to be consistently violating the acceptable limits, it is unloaded the next time it is found idle. Note that with hybrid resource control the system does not need to perform asynchronous termination of extensions. All extensions are guaranteed to terminate, so a misbehaving extension will eventually finish processing and get unloaded.

A. Benefits of the hybrid approach

Hybrid resource control as described above offers the following benefits in the context of active networking:

No asynchronous termination: Asynchronous termination of extension code that shares state with other extensions is undesirable because of increased risk in corruption of state. As discussed above, hybrid resource control never terminates an extension asynchronously but waits for it to finish execution, i.e., processing of a single packet or event.

Reduced runtime overhead: Resource bounds on active extension code provide hints about the runtime behavior of an extension. These hints can be used to reduce the overhead of runtime monitoring. For example, the termination guarantee can be used to reduce the number of checks performed at the interface between the active code and the kernel. In RBClick, we use this guarantee to eliminate function calls to poll the network interfaces at system call entry points (further discussed in section V). This guarantee is also used to eliminate the locking overhead of shared system data structures when calls are made from user-to-kernel or kernel-to-user.

Flexibility: The flexibility of the hybrid technique depends on the tightness of resource bounds: rejection rate of legitimate code increases with the amount of pessimism in estimation of resource bounds. However, this can be partly overcome by inserting *poll points* in the extension code at the appropriate places and dividing the extension code into pieces that individually never violate the acceptable resource limits. At these poll points the system reads the clock, and if the extension is taking too long to execute, it polls the network interfaces and stores packets in internal queues to avoid packet drops.

Figure 1 shows the code that executes at each poll point. 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 average case, the processing overhead of a poll point is very low. In our experiments on a P III 850 MHz machine, this overhead is less than 100 nsec. The poll points let us trade off constraints due to pessimism of resource estimates with the overhead of runtime checks. Note that the above type of flexibility is only useful in polling-mode systems and not in interrupt-based systems.

```
new_timestamp = poll_timestamp() ;
if (new_timestamp − old_timestamp > threshold) then
    poll_interfaces() ;
endif
old_timestamp = new_timestamp ;
```

Fig. 1
CODE EXECUTED AT A POLL POINT.

DoS prevention: Hybrid resource control can be used to reduce the risk of denial-of-service (DoS) attacks on active nodes because a buggy, malicious or particularly badly written active extension implementation can be rejected at an early stage. In the absence of static checking, malicious code needs to be loaded and executed at least once before it is finally rejected, thus increasing the risk of DoS. For example, an active node may admit extension code only if executing it will never cause packet drop at its input queue.

Path Selection: Static checking provides early information for the user introducing an active extension to select a successful *deployment path* through the network. The deployment path determine which route the packets using a particular extension should take through the network. In the absence of static checks, the user introducing an active extension is forced to select a deployment path only by trial and error.

B. Limitations of the hybrid approach

There are two limitations often associated with techniques based on static analysis: *Conservatism of static analysis* and *reduced programming flexibility*. In the following, we discuss how these limitations are not a bottleneck for applicability of hybrid resource control to active extensions.

Conservatism of static analysis: We believe that a moderate amount of conservatism in static analysis (from our experience, up to a factor of 40 for CPU and up to a factor of 10 for memory and network bandwidth resources); is not a serious concern for the applicability of hybrid resource control for many interesting classes of extensions. This is true because of two reasons. First, hybrid resource control uses statically predicted resource bounds only to make admission decisions and guarantee graceful exit. The actual scheduling of resources is done based on precise runtime accounting. Therefore, admitting an extension based on statically predicted bounds does not always affect the scheduling efficiency of the system. Second, many active extensions fall into the category of soft real-time applications. Therefore, in practice, it is possible to optimistically raise the acceptable upper bound on recurring resources such as CPU and network bandwidth. The penalty for doing so is not high. In the worst-case scenario, a router may drop packets at its interfaces, which is not an unexpected behavior for best-effort services. In the case of

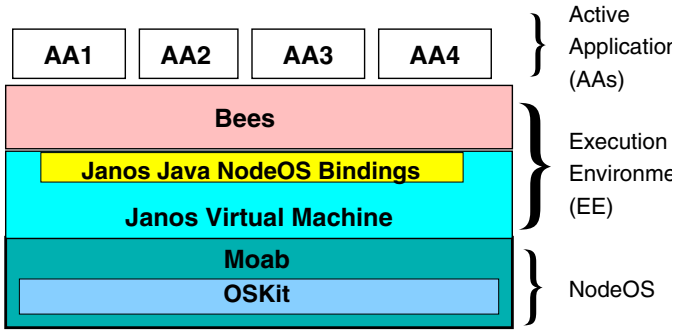

Fig. 2
SOFTWARE ARCHITECTURE OF JANOS AND THE CORRESPONDING DARPA ACTIVE NETWORK NODE ARCHITECTURE.

fixed resources, such as memory, the upper bound should be tighter. Our experience with Click router shows that this is true.

Note that the DoS prevention mechanism discussed above is more useful when the resource bounds are tighter. One way to overcome this limitation is to simply over-provision the resources. More interestingly, runtime measurements combined with statistical analysis techniques can be used to estimate tighter resource bounds with probabilistic guarantees, as in [16], [17]. However, their solutions are not directly applicable in our case because of the untrusted nature of active code; predicting a *typical* work-load for active code is difficult and the active code may also attack the runtime measurement system. We are currently exploring these techniques to build an effective guard against DoS attacks.

Reduced programming flexibility: The halting problem precludes us from performing a complete static analysis on all programs written in a general-purpose programming language. Therefore, to get definite results in a finite amount of time, static analysis techniques often use constrained programming models. For example, SNAP does not allow backward jumps or looping constructs [8]. We believe that to achieve resource boundedness, a general-purpose programming language need not be overly constrained. For example, the restrictions we impose in RBCyclone in section IV-C, are not too constraining for many interesting classes of active extensions. To support this argument, we manually studied all the elements of a particular version of Click [4] and found that all elements can be statically bounded in their resource usage. More details of this study are discussed in section V.

IV. RESOURCE BOUNDED CLICK - RBCLICK

In this section, we present the design of RBClick, an active network environment for best-effort active extensions, which implements hybrid resource control. To discuss RBClick in terms of established terminology, we borrow active networking terminology from the NodeOS specification [18] and use Click terminology as defined in [4].

A. Background

RBClick is designed to be implemented in Janos, an active network operating system that implements the NodeOS and EE layers of an active node [5]. Figure 2 shows a block diagram of the software layers in Janos and their correspondence with the DARPA active network node architecture [18]. Janos supports Java-based active applications on top of the Bees execution environment 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 [19] that implements the active networks community standard NodeOS API specification [18].

Bees-based active applications on Janos forward packets at least four times slower than the fast-path forwarding in Moab [12]. Clearly, Bees environment is not suited for active extensions that want to add custom processing to the fast-path. RBClick is meant to bridge this gap and support high-speed active extensions directly in the Moab kernel.

B. Overview

RBClick is an extension of the Click modular router [4], customized to support untrusted active extensions on a NodeOS. An active extension for RBClick is a graph (or configuration) of packet processing elements, specified in the Click language [4], along with the code for some "new" elements. However, RBClick ensures that all extension configurations are resource bounded, as explained later in section IV-D. RBClick leverages on a significant collection of router extensions available as part of the standard Click distribution. The use of unmodified Click elements as a "trusted base" gives RBClick an opportunity to evolve with Click.

Although the collection of trusted elements in RBClick is sufficient to build a number of interesting extensions, many or even most users will need to download their own code elements to extend the available functionality. RBClick allows untrusted users to download Click-like code elements written in a restricted and type-safe language, RBCyclone, a resource bounded variant of Cyclone [6]. Cyclone is a type-safe version of C and provides control over data representation and memory management. User-supplied untrusted RBCyclone elements can interact normally with trusted Click elements in a single graph.

RBClick limits the resources consumed by active extensions using hybrid resource control. Hybrid resource control technique safeguards the NodeOS kernel against rogue code by admitting only "safe" extensions. RBClick performs code analysis on extension configurations, and any untrusted elements included in them, and admits only those extensions whose resource upper bounds fall under acceptable resource thresholds on the node. In addition to techniques for automatic static analysis of untrusted elements, we have done manual analysis of trusted Click elements and found those elements to be bounded in their resource usage.

We have done an initial implementation of RBClick in Moab. Currently, RBClick instantiates each RBClick extension configuration in a special "lightweight" NodeOS domain [18].

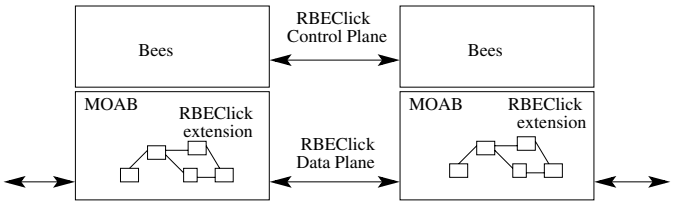

Fig. 3

RBCLICK AND JANOS. THE CONTROL PLANE FOR RBCLICK ROUTER CONFIGURATIONS IS IMPLEMENTED VIA THE BEES EE ON MOAB.

All RBClick extension domains are managed using the hybrid resource control technique. Note that RBClick only implements support for the data path of active extensions, the control path is supported via the Bees EE, as shown in figure 3.

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

C. Resource Bounded Cyclone - RBCyclone

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. In this section, we discuss how we ensure memory and type safety, and resource boundedness of untrusted elements using RBCyclone.

Although virtual memory techniques can ensure complete memory safety from untrusted code, they seem too heavy-weight for use with active extensions. In addition, they do nothing to solve the control flow problems raised by untrusted code, such as asynchronous termination. Therefore, we decided to use relatively lightweight language-based technology.

Interfacing Click to a high-level type-safe language, such as, Java, OCaml, Scheme etc. seems to involve prohibitive performance costs. Overheads of interpretation, marshaling, and garbage collection are the dominating costs [20]. Therefore, we decided to use a type-safe but C-like language, Cyclone [6]. The type-safety of Cyclone guarantees memory protection and its compatibility with C/C++ makes it easier and efficient to interface with Click. To ensure safety and resource boundedness of untrusted code, RBCyclone imposes the following additional restrictions on Cyclone.

1) Namespace Control: RBCyclone elements have access to only two external namespaces: *RBClick_NS* and *CYC_NS*. Untrusted elements cannot directly allocate or otherwise modify the size of a Click packet. Instead, RBClick_NS provides restricted means to perform such operations. RBClick_NS 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 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 just return error messages.

2) 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 number of 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.

3) Memory Management: Cyclone uses *region-based* memory management to avoid dangling pointers and memory leaks [21]. In region-based memory management, each object lives in one region and, with the exception of the heap region which may be garbage collected, all objects in a region are deallocated simultaneously. In Cyclone, there are three kinds of regions: a heap region that lives forever, stack regions that correspond to local declarations, and dynamic regions that have lexically scoped lifetimes but permit unlimited allocation in them.

In RBCyclone, there is no explicit heap region. 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`, such that a reference in a region with larger lifetime cannot point to data in a region with smaller lifetime. For example, a reference allocated in `‘B` cannot point to data in `‘A`. The rationale for these regions is discussed below.

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 this region can be used to share transient state among code elements. In Click such state is shared by allocating extra fields in the packet but RBClick does not allow untrusted code to modify the structure of a packet. Because of its smallest lifetime, none of the other three regions can hold references pointing into this region.

Per-domain packet cache (`‘B`) – In RBClick, a packet is freed as soon as its processing is finished and hence a reference to it cannot be stored in region `‘C`. Region `‘B` provides a fixed size array to cache packets without having to copy them in region `‘C`. 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 [22]. This region is allocated at domain creation time and its size is specified in an extension's

RBClick configuration.

**Region for per-domain memory (`C) –** Region `C is used by a domain to keep state that persists between packets. This region is allocated when a domain is created and destroyed only when the domain is terminated. Memory once allocated in this region, stays allocated for the lifetime of the domain. Also note that elements can communicate with each other by allocating persistent state variables in this region, for example, elements could maintain flow state in this region.

**Region for global shared memory (`D) –** This region is designated for memory that is shared between multiple domains, for example, routing tables. The memory in this region can be obtained using special names for memory areas in this region. We plan to provide a standard filesystem like interface to access memory in region `D.

Implementing these four regions is conceptually simple and only requires modifications to the type-checking system in the Cyclone compiler. However, we haven't yet done these modifications, and our current RBCyclone pre-processor maps `A to a lexically scoped dynamic region and `B, `C, and `D to the heap region in Cyclone.

D. Static Analysis

Static analysis predicts resource upper bounds on active code. In RBClick, static analysis of an active extension is done in two stages: code analysis of individual untrusted elements and graph analysis of its RBClick extension configuration. As was noted in [15], the resource usage, especially CPU time, consumed by a program depends significantly on local conditions on an active node. Factors like 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.

1) Code Analysis of an untrusted element: Designing a sophisticated code analysis tool to estimate resource bounds was not one of the initial goals for RBClick. RBClick's purpose is demonstrating what advantages can be gained given the values for resource bounds. However, to show the feasibility of such a tool, we have designed and implemented a prototype code analysis tool (CAT) that analyzes untrusted code. Currently, code analysis works like this:

An RBCyclone preprocessor validates RBCyclone code and generates valid Cyclone code. This Cyclone code is then compiled by the Cyclone compiler to generate C code. A loop annotation tool then analyzes C code to generate annotations for begin and end points of loops. The loop annotations also include the value of static upper bound on each loop. This annotated C code is then compiled into assembly code using the `gcc` compiler. The loop annotated assembly code, thus generated, is used by CAT's assembly level code analyzer to predict resource upper bounds on untrusted code.

CAT uses simplistic models to predict resource 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 particular resources in the following manner.

CPU: CAT classifies all instructions as either memory reference, function call, or register only operations. All instructions involving memory references are assigned a fixed cost. The cost for each reference is derived from a predicted cache hit rate for untrusted code. Similarly, all instructions that involve only register operations are assigned a fixed cost. 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 system calls. For an active node, a benchmark is used to compute the cost of all system calls. A benchmark-based method to predict node specific CPU time for system calls is also used in [15].

Memory: CAT generates an upper bound on the number of times each Cyclone function calls memory allocation system calls. Since parameters to these system calls are compile-time constants, we can easily infer the upper bound on memory used in all regions by each Cyclone routine. Similarly, a call graph analysis combined with an analysis of local memory allocation routines is used to infer an upper bound on stack space used by an element.

Network: We measure network usage of an element by first calculating the fanout of an element. That is, for one input packet how many output packets does an element generate. Fanout is inferred from the upper bound on the number of *push* calls an element makes into its downstream elements. Fanout number is then combined with the MTU of all the network interfaces to arrive at a network usage number.

2) Static Analysis of an RBC*lick extension configuration:* We have also developed a tool to statically analyze RBClick extension configurations. RBClick configurations cannot have unbounded loops in them. For example, in Click [4], 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. 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 resource upper bounds on each element is used to find a resource upper bound for an active extension.

As mentioned in the section III, poll points can be used to release the pressure due to pessimistic resource bounds. 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 that are absolutely safe

for execution. These segments are then separated by inserting poll elements between them.

3) Discussion: As discussed earlier in this paper, one main issue associated with static analysis is the pessimism in static analysis. Resource upper bounds calculated using CAT tend to be very pessimistic for CPU resource. In our experiments, we found pessimism to vary between factors of 5 and 100. This pessimism results from two sources. First, static analysis often assumes worst-case, which may rarely occur in practice. Second, it is really hard to do precise static analysis for complicated modern machines. Features like multilevel caching, pipelining, and branch prediction make it difficult to precisely model the runtime behavior of hardware. We believe that in our system, pessimism up to an order of magnitude is tolerable but more than a couple of orders of magnitude is not. Therefore, we are currently researching better methods of resource estimation, including better methods of static analysis, runtime measurements, and simulation.

V. EVALUATION

We evaluate our initial implementation of RBClick along two dimensions: *programming flexibility* and *Performance*. By evaluating RBClick's flexibility, we show that our implementation of the hybrid resource control technique is flexible enough to write many interesting classes of active extensions. While by evaluating performance improvement due to RBClick, we confirm that hybrid resource control reduces the overhead due to purely runtime resource control technique currently used in Moab.

A. Programming Flexibility

To verify that our programming model in RBClick is sufficiently flexible, we analyzed all 234 code elements in Click[1] to determine the fraction of its elements whose resource use could be statically bounded. Based on their potential resource use, we categorized 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 unbounded*, e.g., ARP element searches through a data structure whose length is determined by the number of packets it has seen in the last 5 min. Such elements are considered to have potentially unbounded resource usage.

[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.

TABLE I
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%

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 made to behave in a bounded manner if we always precede such elements with a bounds check element. For example, we created an *int8Check* element that can be configured to discard packets in which a particular 8-bit field exceeds a predefined maximum. Another way to constrain these elements is to insert checks in the code to make sure that the value of a required packet field is within certain bounds.

The distribution of 234 Click elements in various categories is shown in Table I. As seen from the table, about 90% of the elements were originally resource bounded. There were 23 elements (the E7 category) which could potentially consume unbounded resources. These 23 elements are potentially resource unbounded because they use unbounded data structures, like linked lists and open hash tables. We could convert these elements to category E5 by recoding the data structures to have a configurable maximum number of data items, which would make all 234 elements statically resource bounded.

This study validates our claim that many interesting classes of active extensions can be coded with the loop restrictions we impose in RBCyclone (because we could statically predict the bounds on all loops in Click elements). Also, this study helped us modify Click elements so that they can be used in untrusted RBClick configurations.

B. Performance

Before evaluating performance improvement due to the hybrid technique, it is useful to look at the overheads that are incurred in Moab due to runtime checks. Note that Moab is a single-address space operating system, so the *system calls* are simple function calls and do not incur any hardware MMU overhead, such as hardware trap or copying of data. Therefore, the user-kernel boundary in Moab is a mechanism for resource control and correctness but not for memory protection.

Most of the overhead of dynamic checks is incurred at system call entry/exit points. All system calls from extension

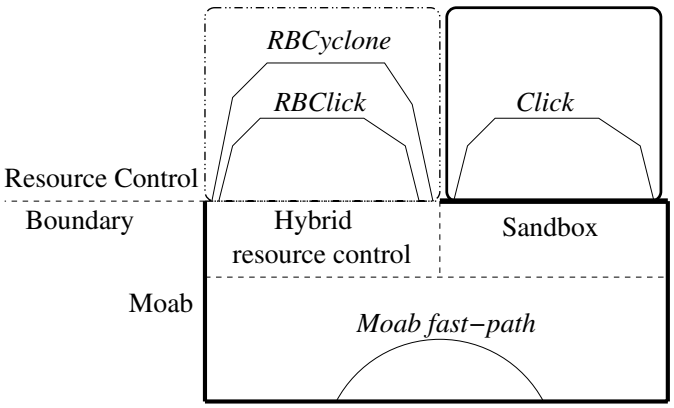

Fig. 4
EXPERIMENTAL CONFIGURATIONS

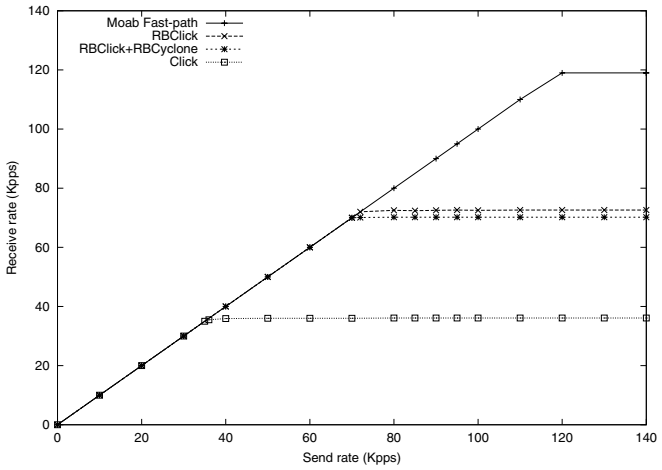

Fig. 5
PERFORMANCE OF IP FORWARDING IN FIVE DIFFERENT CONFIGURATIONS

code into Moab perform two main checks: poll all the network interfaces and mark the user thread non-terminable. These two operations cost about 2.5 μs with two network interfaces. Similarly, checks are also performed when returning from the kernel or making kernel-to-user calls (upcalls) in response to events.

With hybrid resource control, the above checks are not performed because the system is assured that calls into RBClick will eventually return. Some resource accounting is still performed at RBClick entry and exit points just as in the purely dynamic scheme.

To estimate the improvement in forwarding rate obtainable with hybrid resource control, we ran vanilla IP forwarding code in four different configurations, as shown in figure 4: IP fast-path in Moab which uses a hardwired implementation of IP (*Moab fast-path*); a Click configuration at user-level, which is Moab's default resource control technique (*Click*); an RBClick configuration running under hybrid resource control (*RBClick*); and a similar RBClick configuration with one null C++ element replaced with an equivalent RBCyclone element (*RBCyclone*). The null elements immediately push out the packets they receive. This last configuration is used to measure the overhead of boundary-crossing from trusted elements in C++, to untrusted elements in RBCyclone.

All experiments were performed on the cluster portion of the Netbed network testbed [23], in a simple three node setup where an active node running IP routing code interposes between a sender and a receiver. All three machines were 850MHz Intel Pentium IIIs with 512 MB of SDRAM and five Intel EtherExpress Pro/100+ PCI Ethernet cards

As we see from figure 5, IP forwarding under the control of the hybrid resource control technique (*RBClick*) performs much better than with purely dynamic resource control (*Click*). This improvement is because of the reduced runtime checks mentioned above. Note that both *Click* and *RBClick* do not include any MMU-imposed overhead. Therefore, the performance improvement in *RBClick* over *Click* is entirely due to the reduction in overhead of dynamic resource checks at the kernel-RBClick boundary.

Furthermore, the performance difference between *RBClick* and *RBCyclone* 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 an 850 MHz machine. 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 better than higher-level type-safe languages as an extension language for C++.

VI. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we have presented hybrid resource control, a resource control technique for best-effort active extensions, and its initial implementation in RBClick, an execution environment for active extensions. Hybrid resource control uses conservative static checks to reduce the overhead due to runtime checks and avoid asynchronous termination of active extensions. RBClick uses the hybrid technique to control the resources consumed by untrusted user extensions in the Janos kernel. Our measurements 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.

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 focus in this paper has been to show the feasibility and benefits of the hybrid approach. Our code analysis tool is very crude as of now and does not calculate tight resource bounds. In future work, we plan to explore runtime measurements-based techniques to estimate tighter resource bounds on untrusted code, as used in [16], [17]. Tighter resource bounds

can be used to effectively counter DoS attacks while maintaining a high resource utilization factor. One major issue with measurements-based techniques is that of predicting a typical workload for active code. In general, this is impossible to do. However, by requiring the code provider to also provide a workload, and then combining statistical analysis with control-flow analysis, it is possible to predict tight resource upper bounds with very high probability.

ACKNOWLEDGMENTS

The work presented in this paper has benefited from discussions with Alastair Reid, John Regehr, Wilson Hsieh, and Mike Hibler. Further, feedback from the anonymous reviewers and help from Shashi Guruprasad, Eric Eide, Sai Susarla, Scott Owens, and Kathy Gray has helped us improve the quality of this paper.

REFERENCES

[1] 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*, vol. 35, no. 1, pp. 80–86, 1997.

[2] G. V. Back and W. C. Hsieh, "Drawing the Red Line in Java," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. Rio Rico, AZ: IEEE Computer Society, Mar. 1999, pp. 116–121.

[3] G. Back, W. C. Hsieh, and J. Lepreau, "Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java," in *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*. San Diego, CA: USENIX, Oct. 2000.

[4] 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*, 1999, pp. 217–231.

[5] P. Tullmann, M. Hibler, and J. Lepreau, "Janos: A Java-Oriented OS for Active Network Nodes," *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 3, pp. 501–510, Mar. 2001.

[6] 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.

[7] 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*, 1998, pp. 86–93.

[8] J. Moore and S. Nettles, "Practical Programmable Packets," in *Proceedings of the 20th Conference on Computer Communications (INFOCOM). IEEE, Anchorage, Alaska*, Apr. 2001.

[9] 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)*, 2002.

[10] 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.

[11] 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.

[12] 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)*. IEEE, 2003.

[13] P. Menage, "RCANE: A Resource Controlled Framework for Active Network Services," in *The First International Working Conference on Active Networks (IWAN '99)*, vol. 1653. Springer-Verlag, 1999, pp. 25–36.

[14] D. Decasper, Z. Dittia, G. M. Parulkar, and B. Plattner, "Router Plugins: A Software Architecture for Next Generation Routers," in *ACM SIGCOMM*, 1998, pp. 229–240.

[15] 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*. IEEE Computer Society, Aug. 2001, pp. 35–44.

[16] S. Edgar and A. Burns, "Statistical Analysis of WCET for Scheduling," in *IEEE RTSS*, 2001.

[17] 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*, Boston, MA, 2002.

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

[19] 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*, Oct. 1997, pp. 38–51.

[20] 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*, vol. 3, no. 1, pp. 4–18, Mar. 2001.

[21] 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.

[22] T. Wolf and S. Choi, "Aggregated Hierarchical Multicast for Active Networks," in *Proceedings of IEEE MILCOM*, Oct. 2001.

[23] 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*, Boston, MA, Dec. 2002, pp. 255–270.