BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture

Andrew Begel Steven McCanne Susan L. Graham
University of California, Berkeley

February 24, 1999

Abstract

A *packet filter* is a programmable selection criterion for classifying or selecting packets from a packet stream in a generic, reusable fashion. Previous work on packet filters falls roughly into two categories, namely those efforts that investigate flexible and extensible filter abstractions but sacrifice performance, and those that focus on low-level, optimized filtering representations but sacrifice flexibility. Applications like network monitoring and intrusion detection, however, require both high-level expressiveness and raw performance. In this paper, we propose a fully general packet filter framework that affords both a high degree of flexibility *and* good performance. In our framework a packet filter is expressed in a high-level language that is compiled into a highly efficient native implementation. The optimization phase of the compiler uses a new flowgraph set relation called *edge dominators* and a novel optimization technique that we call "redundant predicate elimination", in which we interleave partial redundancy elimination, predicate assertion propagation, and flowgraph edge elimination to carry out the filter predicate optimization. Our resulting packet-filtering framework, which we call BPF+, derives from the BSD packet filter (BPF), and includes a filter program translator, a byte code optimizer, a byte code safety verifier to allow code to migrate across protection boundaries, and a just-in-time assembler to convert byte codes to efficient native code. Despite the high degree of flexibility afforded by our generalized framework, our performance measurements show that our system achieves performance comparable to state-of-the-art packet filter architectures and better than hand-coded filters written in C.

1 Introduction

Over the past decade, a number of innovative research efforts have built upon each other by iteratively refining the concept of a *packet filter*. First proposed by Mogul, Rashid, and Accetta in 1987 [15], a packet filter in its simplest form is a programmable abstraction for a boolean predicate function applied to a stream of packets to select some specific subset of that stream. While this filtering model has been heavily exploited for network monitoring, traffic collection, performance measurement, and user-level protocol demultiplexing, more recently, filtering has been proposed for packet classification in routers (e.g., for real-time services or layer-four switching) [12] [20], firewall filtering, and intrusion detection [18].

The earliest representations for packet filters were based on an imperative execution model. In this form, a packet filter is represented as a sequence of instructions that conform to some abstract virtual machine, much as modern Java byte codes represent programs that can be executed on a Java virtual machine. Mogul *et al.*'s original packet filter (known as the CMU/Stanford packet filter or CSPF) was based on a stack-oriented virtual machine, where selected packet contents could be pushed on a stack and boolean and arithmetic operations could be performed over these stack operands. The BSD packet filter (BPF) modernized CSPF with a higher-performance register-model instruction set. Subsequent research introduced a number of further improvements: the Mach Packet Filter (MPF) extended BPF to efficiently support an arbitrary number of independent filters [23]; PathFinder provided a new virtual machine abstraction based on pattern-matching that achieved impressive performance enhancements and was amenable to hardware implementation [2]; and DPF enhanced Pathfinder's core model with dynamic-code generation (DCG) to exploit run-time knowledge for even greater performance [6].

More recent work on packet classification for "layer four switching" has focused on table-based representations of predicate templates to yield very high filtering performance. Srinivasan *et al.* [20] proposed a special data structure that they call a "grid of tries" to reduce the common case of source/destination classification to a few memory references, while Lakshman and Stiliadis [12] elegantly cast packet classification as the multidimensional point location problem from computational geometry.

Neither the works on imperative models nor those on fast layer-four switching address the issue of compiling an abstract, declarative representation of a packet filter into a low-level form. They also do not consider the minimization of computation by exploiting semantic redundancies across multiple, independent filters in a generalizable fashion. Work on such optimizations has not been forthcoming for good rea-

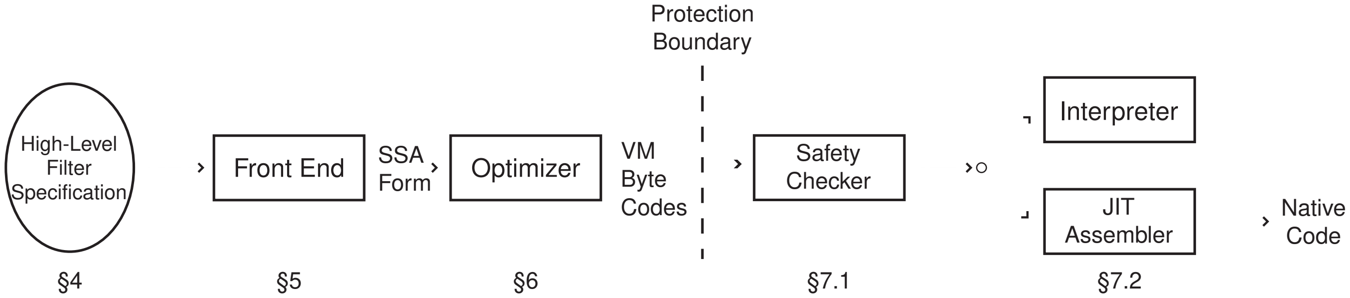

Figure 1: System architecture diagram for BPF+. A filter, represented in a high-level language, is compiled and optimized into the BPF+ virtual machine intermediate representation. After traversing protection boundary, the protected domain verifies the filter code specification, and either interprets the byte codes or assembles them on-the-fly into native code.

son. If we model a packet filter program as a function of boolean predicates, then we can reduce filter optimization to the "decision tree reduction" [9] problem. Since this problem is "NP complete", we know that filter optimization is a hard problem. As a natural consequence, many decision tree reduction works have relied upon *heuristics* for optimization [4] [14] [16] [19].

Fortunately, many packet filters have a regular structure that we can use to our advantage in our optimization framework. One way to exploit this structure is to account for it in the underlying filtering engine itself. Both PathFinder and MPF are based on this design principle: PathFinder utilizes a template-based matching scheme that is nicely amenable to the computation required for parsing packet headers, while MPF extends BPF with specific opcodes that provide a particular solution tuned to demultiplexing.

Although these sorts of assumptions are an important component of any overall packet filter system, they fail to address what we believe is the ripest opportunity for packet filter optimization: the application of *global optimization* algorithms across the filter predicate flow graph to minimize the average path length through that graph. In contrast, the MPF extensions of BPF, PathFinder, and DPF all use pattern-matching heuristics that operate *locally*, e.g., they do not necessarily eliminate common subexpressions across the predicates, nor do they detect the equivalence of semantically equivalent boolean expressions. In fact, they either restrict the set of expressible filters to those with a regular structure that can be matched by simple patterns, or they require that the "filter programmer" expresses the filter in a compact and already-optimized low-level representation. Although this may be a reasonable design assumption in "low level" environments (e.g., where an OS protocol module creates a packet filter to match its signature traffic as in the x-kernel [8]), it is less applicable to "high level" domains (e.g., where a user specifies a filter in an expressive high-level language and a compiler generates the actual low-level filter code). In this latter case, the front end code generator would typically translate a complex filter expression into a number of redundant packet subpredicates; thus, optimization becomes especially important to eliminate the redundant code.

In this paper, we propose optimization techniques that exploit well-known data-flow optimization algorithms in a novel way for the generalized optimization of packet filters. Our data-flow algorithm, which we call "redundant predicate elimination," interleaves partial redundancy elimination, predicate assertion propagation, and flowgraph edge elimination to effect predicate optimization. In particular, we employ a set relationship called *edge dominators* that extends the traditional node dominator relationship from flowgraph nodes to edges and provides the key ingredient for our predicate optimizations. Finally, we leverage the pattern-matching heuristic, developed in the PathFinder and DPF work, in our back end, as a lookup table optimization performed after the removal of redundant predicates.

Armed with our global data-flow optimizations, we can afford the flexibility of a high-level representation for packet filters since we can compile and optimize them into native implementations that achieve state-of-the-art performance from the resulting packet-filter code. To realize a practical system based on this design, we crafted a generalized architecture for packet filtering, which we call BPF+, built around our suite of filter optimizations. Our system design derives from many years of the BPF development effort and leverages extensive feedback from a large user population as well as integration efforts into several common Unix platforms. Several of the components described herein are mature pieces of work that have yet to be described in the literature[1] while others are comparatively new.

As depicted in Figure 1, the BPF+ system consists of a number of sequentially arranged components that transform a high-level filter language specification into an low-level executable packet filter:

- The input to the front end is a high-level language for filter expressions based on the declarative predicate syntax

[1] This is a very rough attempt at maintaining anonymity for the SIGCOMM review process. The final paper will depict the history of this work explicitly.

used in the Lawrence Berkeley National Laboratory's packet capture library *libpcap* [10] and network monitoring tool *tcpdump* [11].

- The BPF+ compiler translates the predicate language into an imperative, control-flow graph representation. The particular intermediate form we use is called "static single assignment" (SSA) [5], which is well-suited for our optimization algorithms.

- The SSA intermediate representation is fed forward to the code optimizer, which performs both global and local data-flow optimizations over the control-flow graph form of the intermediate code. The output of the optimizer is a byte code representation that conforms to the BPF+ virtual machine model, which is a RISC-like register-based variant of the accumulator-based virtual machine definition of the original BPF pseudo-machine [13].

- The BPF+ byte codes are then delivered to an execution environment, e.g., across the user-kernel boundary to implement user-defined protocol demultiplexing, or across the network and into a switching element to implement an externally-defined network service like policy-based traffic management.

- Once received in the target protected domain, the safety verifier ensures the program's integrity.

- Finally, a "just in time" (JIT) assembler translates the optimized and safety-verified byte codes into native code and performs optional machine-dependent optimization. This last stage is omitted if the target environment is an interpreter rather than native hardware, e.g., as with the BPF kernel implementation, which interprets filters in the byte code form.

In the remainder of this paper, we motivate, describe and evaluate the components of the BPF+ architecture. We first outline related packet filtering technologies and identify certain limitations of this current art. We then present the BPF+ front end: its high-level filtering language, the virtual machine model, and the compiler that generates the SSA intermediate form. Next, we describe our optimization framework based on the set of local and global data-flow algorithms and their interactions. Subsequently, we describe the back end that verifies the integrity of the byte-code representation and optionally transforms that representation into a native machine code. To demonstrate the efficacy of our approach, we then present measurements of our implementation that show that BPF+ performance is comparable to existing packet filter implementations despite its enhanced flexibility. Finally, we summarize our plans for future work and conclude.

2 Related Work

In its widely used form, the BPF kernel sub-system represents each user-specified filter as a separate entity. For every packet received, each filter in turn is run over that packet and each filter that accepts the packet is given its own copy. Hence, if BPF were used to implement user-level protocols, for instance, the demultiplexing overhead would scale linearly with the number of filters, e.g., a busy server with many simultaneous network connections would suffer linear slowdown as each connection would independently run the packet filter on its own stream.

To overcome this limitation, MPF enhanced the BPF virtual machine with instructions for efficient protocol demultiplexing. Rather than represent each filter separately, MPF exploits the structure of demultiplexing filter specifications to recognize that two filters are similar up to, say, the transport header port fields, using simple template-matching heuristics. Once MPF detects this similarity, it merges the new predicate with the existing filter automatically by expanding the existing port checks to include the new port number, for example.

PathFinder generalizes the MPF heuristic with a redesigned filtering engine that is better matched to the pattern-matching transformation. In this framework, templates called "cells" represent packet field predicates, which are chained together in a "line". This line of cells represents a logical AND operation over the constituent predicates. A collection of lines is arranged into a chain of predicates, which represents the logical OR over all lines. As lines are installed into this chain, PathFinder searches for and eliminates common prefixes.

For example, if process P requests TCP packets sent to port A and process Q requests TCP packets sent to port B, then the resulting filter logic would have the following form:

```
if link layer type = IP and
   IP fragment offset = 0 and
   IP protocol = TCP and
   TCP dest port = A
then deliver pkt to P
else if link layer type = IP and
   IP fragment offset = 0 and
   IP protocol = TCP and
   TCP dest port = B
then deliver pkt to Q
```

Upon processing the second filter, PathFinder would recognize the common prefix and simply extend the first if-clause as follows:

```
if link layer type = IP and
   IP fragment offset = 0 and
   IP protocol = TCP
then
  if TCP dest port = A
  then deliver pkt to P
```

```
else if TCP dest port = B
then deliver pkt to Q
```

Since the inner if-else statement is effectively a "switch" over the destination port field, a jump table (perhaps using a perfect hash over the target value set) could be used to implement an O(1) match, and PathFinder does precisely that.

DPF utilizes the same template-matching approach as PathFinder (templates are called "cells" in PathFinder and "atoms" in DPF), but introduces a new low-level language and employs dynamic code generation to attain performance improvements over other interpreter-based implementations. Its new language is based on a "read window" which may be shifted and masked to match words in the packet to various immediate constants. Given a filter specified in this language, DPF coalesces common prefixes into lines, performs some additional local optimizations, and dynamically generates native machine code to directly evaluate the filter.

The more recent works geared toward layer-four switching [12] [20] take the DPF and PathFinder approaches to an extreme, where the entire model is based on a set of templates that are matched against known constants (or known constant ranges).

While the template-matching model yields good performance, there are a number of shortcomings associated with the technique. For example, it is not possible to match fields in the packet header against one another, for instance, to look for packets that originate and terminate in the same network ("source network = dest network"). Nor is it possible to perform arbitrary mathematical operations on header words before matching.

DPF and PathFinder resort to a set of *ad hoc* heuristics for producing efficient filters by coalescing common prefixes. These optimizations are foiled in PathFinder when predicates are reordered. DPF, however, enforces in-order packet header traversal, thus common prefixes will always appear in the same order. However, when the filter itself does not conform to the same order as other already installed filters, prefix compression fails.

To illustrate this pathology, consider the packet filter, "all of the packets sent between host X and host Y". In a boolean framework, we would specify this filter as "(source host X and dest host Y) or (source host Y and dest host X)", and in flowgraph form, the expression would appear as in Figure 2. Here, basic blocks are represented by nodes and boolean control transfers are depicted by edges. By convention, false branches point to the left; true branches to the right.

In this case, DPF, finding no common prefix and unable to reorder the checks to obtain a common prefix, would compile the condition into two separate filters that are sequentially invoked. However, there is opportunity for optimization, which DPF by necessity must miss. If the thread of control during filter evaluation reaches the node "dest host Y," then we necessarily know that the source host is X. Furthermore, from that vantage point, we know that the source host cannot be Y and that the node pointed to by the dashed edge is redundant. But, we cannot eliminate the "source host Y" node yet because there exists another path (from the root) for which the check is not statically known. Therefore, our recourse for optimization is to transform the dashed edge so that it points to the FALSE node, thus reducing the average path length through the flowgraph (and in turn, enhancing filter execution performance).

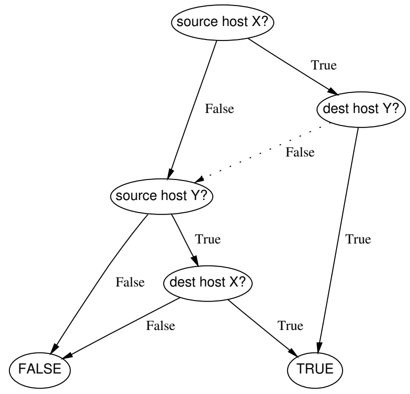

Figure 2: Control-flow graph for "(src host X and dst host Y) or (src host Y and dst host X)". The dashed edge points to a redundant predicate and may be redirected to the FALSE node.

This is the sort of global data-flow optimization we want to exploit in our packet filter optimizer. Having established this context, we can now present the core pieces of the overall system design, beginning in the next section with the BPF+ machine model.

3 The BPF+ Machine Model

Before presenting the details of the translation modules that map filter predicates to the BPF+ machine representation, we sketch in this section a high-level overview of the BPF+ machine model to establish context for the rest of the paper. This version of the BPF virtual machine represents a number of iterative refinements made over the past several years to the original BPF machine model.

The BPF+ abstract machine is a RISC-like, 32-bit, load-store architecture consisting of a set of 32 general purpose registers, a program counter, data memory, packet memory, a packet length register, and a pseudo-random register. A filter program is represented as an array of byte codes that conform to a well-defined instruction format.

The BPF+ virtual machine supports five classes of operation:

- *load* instructions copy a value into a register. The source can be an immediate value, packet data at a fixed offset, packet data at a variable offset, the packet length constant, or the scratch memory store (a reference to data beyond the end of the packet results in a return value of 0);

- the *store* instruction copies a register into a fixed location in data memory;
- *ALU* instructions perform arithmetic or logic on a register using a register or a constant as an operand and a register as the destination (division by zero causes the filter to immediately return a value of zero);
- *branch* instructions alter the flow of control, based on a comparison test between a register and an immediate value or another register; and,
- *return* instructions terminate the filter and indicate the integer-valued result of evaluation.

A filter is evaluated by initializing the packet memory to the packet in question and executing byte codes on the BPF+ machine until a return instruction is reached. The data memory is persistent and may be queried by agents external to the filter engine. The pseudo-random register is a read-only register that returns a uniformly distributed random value each time read, which is a useful primitive for building filters that can perform probabilistic sampling. To facilitate safety verification, we require that all program branches be forward (thus disallowing loops) and that the last instruction on each path be a "return". In addition to the set of conditional branch instructions, we add a lookup table instruction to abstract multiway conditional branches for later just-in-time optimization.

We omit the details of the instruction format and throughout the rest of this paper use an assembly language syntax that is relatively self-explanatory. For example, a simple BPF+ byte-code program that matches TCP packets has the following form:

```
        lh      [12], r0
        jne     r0, #ETHERTYPE_IP, L5
        lb      [23], r1
        jne     r1, #IPPROTO_TCP, L5
        ret     #TRUE
L5:     ret     #FALSE
```

Presuming Ethernet encapsulation, this filter first checks that the packet is an IP packet. If so, it then checks if the IP protocol type is TCP, in which case it branches to an instruction that returns true. In any other case, the program branches to label L5 and returns false.

This form of representation is far too low level for many applications of packet filters. In the next section, we argue that high-level filtering languages are important for a number of problem domains and we sketch the characteristics of the high-level filtering language that BPF+ employs.

4 The Predicate Language

The input to our system is a high-level filter represented in a declarative predicate language. By employing a high-level language, we hide the complexity and details of the underlying, imperative execution model of the BPF+ virtual machine. This facilitates the expression of complex boolean relationships among many different predicates using natural logical expressions rather than awkward control structures. Unlike other high-performance packet filter packages that have adopted more restrictive semantics for their packet filter abstractions (i.e., the template matching model), we retain the full generality of a programmable, control-flow graph model for our virtual filter machine.

There are many reasons to support higher-level abstractions for packet filtering. To begin with, the system should hide the details of where particular fields are located in packet and how variable-length headers must be parsed to locate those fields. For example, BPF+ refers to the IP destination address field in a packet as "IP dst host" rather than "packet[20:4]". Additionally, a seemingly simple BPF+ expression like "TCP port HTTP" turns out to have a relatively complex low-level structure that should not be a burden to the filter programmer (i.e., in this case, the packet must be IP; if fragmented, it must be the first fragment so as to contain the IP header; there may be IP options which must be skipped over to find the TCP ports; and finally both the source and the destination TCP port field must be checked against the constant 80).

This sort of high-level representation is crucial if a human user is specifying the packet filters. While a low-level pattern specification might have sufficient generality and simultaneously be amenable to an efficient implementation, a network administrator that is diagnosing network malfunctions on-the-fly or chasing down an intruder in real-time must have a flexible and easy-to-use syntax for specifying packet predicates. Thus, a high-level predicate syntax that allows one to look for, say, packets "between MIT and UCB" that are "HTTP connections" should be naturally and easily specified. To this end, the user should be able to specify which fields of the packets they want to match and connect those predicates with boolean operators "and", "or", and "not". In BPF+, the filter would look like this expression:

```
((src network MIT and dst network UCB) or
 (src network UCB and dst network MIT)) and
(TCP port HTTP)
```

By contrast, the same expression written in DPF's quite low-level SHIFT language would look like the following:

```
(((12:16 == 0x8)     &&  # IP?
  SHIFT(6 + 6 + 2)   &&  # skip Ether header
  (9:8 == 6)         &&  # TCP?
  (12:8 == 18)       &&  # src network MIT?
  (16:16 == 0x8020)  &&  # dst network UCB?
```

```
 SHIFT(20)          &&  # skip IP header
                        # (assume fixed length)
 (0:16 == 80)       &&  # src port 80?
 (2:16 == 80))          # dst port 80?
||
((12:16 == 0x8)     &&  # IP?
 SHIFT(6 + 6 + 2)   &&  # skip Ether header
 (9:8 == 6)         &&  # TCP?
 (12:16 == 0x8020)  &&  # src network UCB?
 (16:8 == 18)       &&  # dst network MIT?
 SHIFT(20)          &&  # skip IP header
                        # (assume fixed length)
 (0:16 == 80)       &&  # src port 80?
 (2:16 == 80))          # dst port 80?
```

In the middle ground between a predicate language and a fully general pattern specification language, we might interpose the ability to match various fields of the packet in relation to each other, or the ability to perform mathematical operations on the fields before matching them. Thus, for example, to track down a TCP protocol bug, we might need to extract all the packets from a trace that fall within a certain range of TCP sequence numbers, e.g., *TCP.seqno* $>$ *10000 and TCP.seqno* $<$ *11000*.

Finally, users may want to combine the aforementioned filter language approaches and compose them with a policy language that enables the runtime system to apply a filter at a particular time (e.g. for probabilistic sampling of packets meeting a particular predicate), add a filter (e.g. if the source address of an intruder has been identified), or remove a filter from use (e.g. if a particular email adversary sends unsolicited mass email only at certain times of the day).

Designing a language that meets these high-level requirements is not a difficult problem. Several languages have been devised, for example the filtering language in the Lawrence Berkeley National Laboratory's packet capture library *libpcap*, Sun's *etherfind* program, and Digital's *snoop* tool. Since the BPF+ design effort is built upon BPF, *libpcap*, and *tcpdump*, we naturally incorporated the *libpcap* language into our system. We omit the details of this well-known and widely used packet capture system, as it is well described elsewhere [10] [11].

5 The Front End

Given our high-level filter language and our low-level filter machine model, we are faced with the problem of translating filter predicates into BPF+ byte codes. Rather than integrate translation and optimization into a monolithic framework, as PathFinder and DPF have done, we have deliberately factored apart the translation stage from the optimization stage. This has a number of advantages. First, it allows us to create different front ends and high-level languages that can be optimized and carried by the same back end. Second, it allows us to evolve and develop the two stages independently. An improvement to the optimization framework need not require changes to the high-level language defined in the front end. Finally, this breakdown provides a framework for incrementally composing filters on the fly, e.g., as required by user-level protocol demultiplexing where filters are installed and removed dynamically. More specifically, a set of active filters (each individually representing a given connection fingerprint) can be maintained in predicate form so that filters may be easily inserted and deleted. Each time the set changes (because a connection starts or stops), we can invoke the optimizer and back end on the altered form to produce our new aggregate filter program.

Another advantage of the separation between the compiler and optimizer is that the code generator is greatly simplified. For example, consider the way in which we generate code for short-circuited logical predicates. In an expression like "p_0 *and* p_1", p_1 is evaluated only if p_0 is true. However, the second predicate might contain sub-predicates that have already been evaluated in the first predicate. For example, the expression may have a decomposition, in which another predicate p_4 represents a common protocol check, e.g., "(p_4 *and* p_0) *and* (p_4 *and* p_1)". Factoring out these common predicates all within the code generator would be a complex task. The optimizer, on the other hand, is well suited to the elimination of this sort of redundancy. Thus, our code generator can be relatively simple and straightforward and rely on these later optimizations to achieve efficiency.

In short, we have adopted an approach where we first transform the predicate language into an intermediate form through naive compilation, and then apply aggressive optimizations to transform the naively compiled structure into an optimized BPF+ byte-code program.

The BPF+ compiler uses off-the-shelf lexical analysis and parsing tools as well as well-known compiler techniques to convert the filter specification into a control-flow graph in SSA intermediate form. SSA is a relatively new intermediate representation used in optimizing compilers, in which the abstract data values are separated from the locations in which they are stored. The key property of SSA is that any register is written exactly once, so we assume that we have an infinite supply of registers with which to work. In turn, we rely upon a register allocator to map this unbounded number of virtual registers into a finite set of physical registers. SSA is highly amenable to many simple but effective forms of global dataflow optimization, and we heavily exploit this property in our system.

Each node in the control-flow graph generated by the BPF+ compiler is a basic block in SSA form that ends with a boolean predicate. There is one unique entry node, and flow moves through the graph until it reaches a "return" statement. At the end of each basic block, the flow may branch based on the value of the predicate. Flow may only move forward (downward through the graph); this property is enforced by the requirement that branch offsets must be positive. Thus, the entire graph is guaranteed to be acyclic.[2]

[2]The fact that BPF+ flowgraphs are acyclic simplifies data-flow calcula-

6 The Optimizer

The price that we pay for our naive SSA form code generation is many computational and logical redundancies. This results in an overabundance of code, conditional branches, and allocated registers. Thus, optimization of the generated code is vitally important for improving its performance and justifying the cost of the high-level starting point. In this section, we describe the global data-flow optimizations and peephole optimizations that are performed on the intermediate code — which remove redundancies, rearrange non-optimal code sequences and identify potential lookup tables — in order to generate efficient code.

In addition to incorporating many standard optimizations found in traditional compilers, the BPF+ optimizer introduces a novel application of the *redundant predicate elimination* global data flow analysis. This optimization is rarely found in compilers for traditional languages like C or Java because redundant predicates do not occur very often and the optimization would not be very profitable. However, in the domain of packet filter compilation, BPF+'s naive code generator produces decision trees with many redundant predicates, thereby making this optimization one of the most useful that can be applied.

The next four sections describe our optimizations in more detail. In the first section, we introduce the redundant predicate elimination and its composition from partial redundancy elimination, predicate assertion propagation, and redundant edge elimination. Then, we illustrate the peephole optimizations that are performed within the basic blocks. We also use constant folding and constant propagation to help identify and eliminate redundant computations in the global data flow phase of optimization. After the other optimizations have completed, we enter a jump table encapsulation phase to optimize linear sequences of predicates. Finally, we do register allocation and assignment to map each remaining variable to an actual register in the BPF+ virtual machine.

To get a feel for the potential of the redundant predicate elimination optimization, consider the following filter:

```
IP src host A or IP src host B
```

Without optimization, this expression is compiled into the following code[3]:

```
        lh    [12], r0
L1:     jeq   r0, #ETHERTYPE_IP, L3
        ja    L5
L3:     ld    [26], r1
        jeq   r1, #A, L11
L5:     lh    [12], r2
L6:     jeq   r2, #ETHERTYPE_IP, L8
        ja    L10
L8:     ld    [26], r3
        jeq   r3, #B, L11
L10:    ret   #FALSE
L11:    ret   #TRUE
```

Note that both predicates test whether the packet is IP. Since the first test (at L1) always occurs before the second (at L6), the second test is redundant and may be eliminated. The problem is better visualized by analyzing the program in flow graph form. Figure 3 shows the basic blocks and control edges that correspond to the filter above. By convention, false branches are to the left of true branches. The nodes are numbered for reference. The dashed boxes indicate the two predicates, *IP src host A* and *IP src host B*.

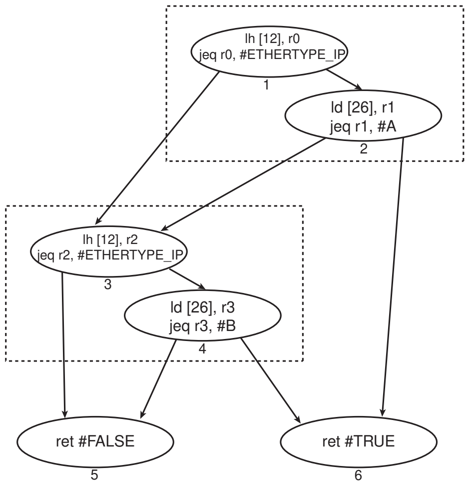

Figure 3: Unoptimized version of "IP src host A or B".

Since control must pass through N_1[4] before reaching N_3, and since N_1 and N_3 perform equivalent tests, N_3 is redundant. However, at N_3, it is not known whether the result is true or false, since either edge could have been taken on exit from N_1. On the other hand, we know the result of N_3 from the vantage point of the in-bound edges. Therefore, our approach is to find edges that point to redundant nodes, and point them past the redundancy.

For instance, along edge E_{23}[5] we know that N_1 is true; and since N_1 and N_3 perform equivalent tests, N_3 must be true from this vantage point. Thus, edge E_{23} can be deleted, and edge E_{24} inserted. Similarly, if flow passes along E_{13}, then N_3 will be false; hence, E_{13} can be replaced by E_{15}. The flow graph after these modifications is shown in Figure 4. A

tions considerably. Because all information flows only up (or only down), a minimal fixed point solution can be reached with a single top-down (or bottom-up) level-order traversal of the control-flow graph.

[3]Logic is inverted in several places to make the conditional branch code more straightforward to read. The compiler back end optimizes the order of the basic blocks to minimize the need for absolute jumps.

[4]Let N_i be a synonym for node i.

[5]Let E_{ij} denote the directed edge from N_i to N_j.

reachability analysis will discover that N_3 is now unreachable and eliminate the dead code from the graph.

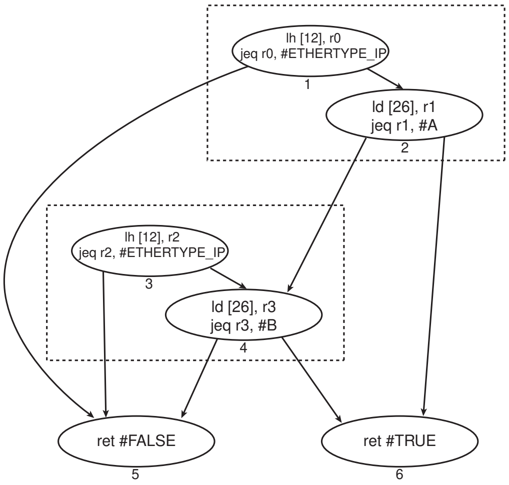

Figure 4: Moving the edges.

As is often the case in optimization algorithms, one class of optimizations will expose opportunities for others. Here, the edge movements have caused a load operation to become redundant. Since the in-degree of N_4 is reduced to one after the dead code at N_3 is eliminated, we know that N_4 and N_2 load the same value. Thus, the second load at N_4 can be removed. Figure 5 shows the flow graph in its final form.

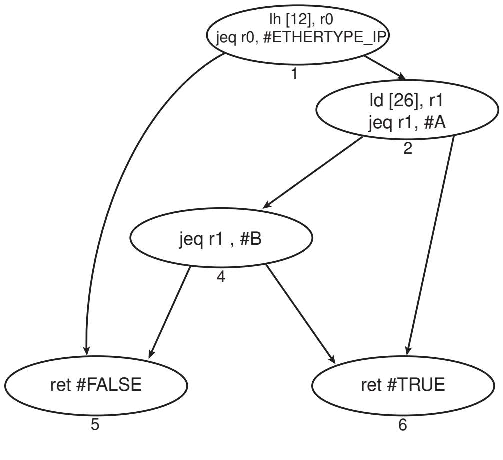

Figure 5: The optimized filter.

6.1 Redundant Predicate Elimination

Redundant predicate elimination is an optimization used to determine, at compile-time, which predicates found in the control-flow graph may be bypassed by particular flow edges. This optimization is composed of three pieces: *partial redundancy elimination*, used to eliminate redundant computation within the nodes of the control-flow graph; *predicate assertion propagation*, a data-flow analysis used to flow the values of determinable predicates down through the control-flow graph to the leaves; and *static predicate prediction*, which uses the assertion information to identify statically determinable conditional branches and bypass them whenever possible.

6.1.1 Partial Redundancy Elimination

Our use of SSA form, combined with BPF+'s acyclic control-flow graph, enables the optimizer to identify and eliminate a significant amount of redundant computation. In the code from our simple code generator, most redundancies are loads from packet memory and oft-repeated ALU operations.

In order to determine which computations are redundant, we must first establish a metric of value equivalence. We use a value numbering scheme for each register to indicate its source definition. Each definition, which can be a defining computation, a load from memory, or a register-to-register copy, is identified by a unique ID which can be used to indicate whether two variables have the same definition.

We compute the *node dominator* relation over the control-flow graph and look over every register's definition. This relation identifies which nodes must be traversed in order to go from the entry node to each node in the control-flow graph. If at a given node, the value assigned to a register has already been computed in a dominating node, the second definition is redundant.[6] We then replace the redundant computation with a register-to-register copy from the dominating defining register. Afterwards, using copy propagation, we replace all later uses of the second register with the first. A subsequent dead store elimination phase will remove the now useless register and the corresponding register-to-register copy.

This implementation only achieves partial redundancy elimination, however, since redundancies may only be identified and elided when found in dominating relationships. We shall see how the next two phases of redundant predicate elimination can improve the effectiveness of this optimization if we apply them one after another.

6.1.2 Predicate Assertion Propagation

The example shown at the beginning of Section 6 assumes a priori that we can make certain edge movements without compromising the semantics of the program. In actuality, we must be analytically precise that such transformations are legitimate. This problem can be solved through a global data-flow analysis.

The traditional approach to global data-flow problems typically involves computing set relations over the nodes of a flowgraph. However, as first seen in Cocke and Schwartz [3]

[6] Since our SSA form control-flow graph is acyclic, and each register is only defined once, we do not have to check whether the register's value might have been changed before the second definition is reached.

and later exploited by Graham and Wegman [7], applying the data-flow functions to *edges* rather than nodes can have substantial advantages. This is indeed the case for BPF+ flow graphs.

First, we adapt terminology traditionally used to describe node relationships to make the following definitions: An edge (E_0) (defined by a predecessor node *pred(*E_0*)* and a successor node *succ(*E_0*)*) *dominates* another edge (E_1), written E_0 *dom* E_1, if every possible execution path from the entry node to E_1 includes E_0. In addition, an edge (E_0) immediately dominates another edge (E_1), if E_0 dominates E_1 *and* there is no edge (E_2) such that E_0 dominates E_2 and E_2 dominates E_1.

Since every basic block ends with a predicate, an edge E represents the truth value (*sense(E)*) of a predicate (*predicate(pred(E))*) — a *true* edge (*true(pred(E))*) is traversed if the predecessor node evaluated a true condition, otherwise the *false* edge (*false(pred(E))* is traversed. Suppose an edge E_0 dominates an edge E_1. If the edge predicate of E_0 is equivalent to the predicate of the successor node N of E_1, then we know the outcome of N, when traversed from E_1. Hence, we can delete E_1 and insert a new edge from the previous predecessor of E_1 to the appropriate child of N, provided no conflicting inter-block data dependencies exist.

We use a simple data-flow algorithm to abstractly define the value of each predicate in the control-flow graph. If a predicate ends up with a statically determinable value, we may bypass the predicate with a new control-flow edge. First, we compute the edge dominator relationship[7] in a fashion similar to the node dominators algorithm given by Aho, Sethi, and Ullman [1]. The set relation, which we call *edom*, is given by:

$$\mathrm{edom}(E) = \{E\} \cup \{ \bigcap_{P \in \mathrm{pred}(E)} \mathrm{edom}(P)\}$$

We use *edom* to calculate *idom*:

$$\begin{array}{l}
\forall E \in \text{edges}, \\
\quad \mathrm{idom}(E) = \mathrm{edom}(E) - \{E\}, \\
\forall E \in \text{edges}, \\
\quad \forall F \in \mathrm{idom}(E), \\
\qquad \forall G \in \mathrm{idom}(E) - \{F\}, \\
\qquad\quad \text{if } \; G \in \mathrm{idom}(F) \\
\qquad\qquad \mathrm{idom}(E) = \mathrm{idom}(E) - \{G\}
\end{array}$$

The immediate dominator relation forms a forest of trees, where each edge (E) in the control-flow graph is a node in a tree. The predecessor of each node is its immediate dominator and its successors are those nodes which it immediately dominates. We use this tree in the next phase of predicate assertion propagation.

[7]The fact that BPF+ flowgraphs are acyclic allows us to compute this flow equation in O(E) time.

For each edge in the control-flow graph, there are a set of assertions that we can make about the values of the predicates. For instance, the false edge coming out of a node that tested the predicate $a = 6$ would contain the assertion that $a \neq 6$. In addition, the assertions for all of the edge dominators of a particular edge also hold true for that edge, since those edge dominators must be traversed in order to reach it. The assertion set relation is given by the following:

$$\begin{aligned}\mathrm{assertion}(E) = &\{<\mathrm{predicate}(\mathrm{pred}(E)), \mathrm{sense}(E)>\} \\ &\cup \mathrm{assertion}(\mathrm{idom}(E))\end{aligned}$$

Each element of the assertion set is a tuple of the predicate tested (*assertion(E).predicate*) and and the value of the proven answer (*assertion(E).sense*).

6.1.3 Static Predicate Predication

Now that we have the assertion set for each edge, we are ready to use this information to predict statically determinable predicates. In general, the problem of proving that a set of assertions implies a certain result is NP-complete, however, there is a small set of rules that we can use in practice to prove many assertions about the predicates typically found in packet filters. The rules used by BPF+ are shown in Table 1.

Beyond these few entries, a generalized theorem prover would be necessary to make more involved implications from the given set of assertions. However, it turns out that the most-used implications come from the `jeq` and `jne` entries of the table.

For a particular edge E, if the assertions in *assertion(E)* statically prove *predicate(succ(E))* to be true or false, then on this path, edge E may bypass the redundant predicate and we may remap the edge's successor to the predicted child of *succ(E)*. We may do this only with the guarantee that the edge movement does not violate data dependencies that occur later on in the flow graph. Specifically, if any registers defined in the node to be bypassed are used by any other node on the predicted path, we must forbid the movement.

Formally, the algorithm looks like this:

$$\begin{array}{l}
\forall E \in \text{edges}, \\
\quad \forall (\mathrm{pred}, \mathrm{sense}) \in \mathrm{assertion}(E), \\
\qquad \textit{let } N = \mathrm{succ}(E), \\
\qquad\quad P = \mathrm{predicate}(N), \\
\qquad \textit{in} \\
\qquad\quad \textit{if table}(\mathrm{pred}, \mathrm{sense}, P) = \mathrm{TRUE} \\
\qquad\qquad \mathrm{succ}(E) = \mathrm{succ}(\mathrm{true}(N)) \\
\qquad\quad \textit{if table}(\mathrm{pred}, \mathrm{sense}, P) = \mathrm{FALSE} \\
\qquad\qquad \mathrm{succ}(E) = \mathrm{succ}(\mathrm{false}(N))
\end{array}$$

Input							Output
Assertion			*Sense*	*Predicate*			*Sense*
jeq	#lval	#rval	TRUE	jeq	#lval	#rval	TRUE
jeq	#lval	#rval	TRUE	jne	#lval	#rval	FALSE
jeq	#lval	#rval	TRUE	jlt	#lval	#rval	FALSE
jeq	#lval	#rval	TRUE	jgt	#lval	#rval	FALSE
jeq	#lval	#rval	FALSE	jeq	#lval	#rval	FALSE
jeq	#lval	#rval	FALSE	jne	#lval	#rval	TRUE
jeq	#lval	#rval1	TRUE	jeq	#lval	#rval2	FALSE
jne	#lval	#rval	TRUE	jne	#lval	#rval	TRUE
jne	#lval	#rval	TRUE	jeq	#lval	#rval	FALSE
jne	#lval	#rval	FALSE	jeq	#lval	#rval	TRUE
jne	#lval	#rval	FALSE	jne	#lval	#rval	FALSE
jne	#lval	#rval1	FALSE	jne	#lval	#rval2	TRUE
jlt	#lval	#rval	TRUE	jlt	#lval	#rval	TRUE
jlt	#lval	#rval	TRUE	jeq	#lval	#rval	FALSE
jlt	#lval	#rval	TRUE	jge	#lval	#rval	FALSE
jlt	#lval	#rval	TRUE	jgt	#lval	#rval	FALSE
jlt	#lval	#rval	FALSE	jlt	#lval	#rval	FALSE
jlt	#lval	#rval	FALSE	jge	#lval	#rval	TRUE
jgt	#lval	#rval	TRUE	jgt	#lval	#rval	TRUE
jgt	#lval	#rval	TRUE	jeq	#lval	#rval	FALSE
jgt	#lval	#rval	TRUE	jle	#lval	#rval	FALSE
jgt	#lval	#rval	TRUE	jlt	#lval	#rval	FALSE
jgt	#lval	#rval	FALSE	jgt	#lval	#rval	FALSE
jgt	#lval	#rval	FALSE	jle	#lval	#rval	TRUE
jle	#lval	#rval	TRUE	jle	#lval	#rval	TRUE
jle	#lval	#rval	TRUE	jgt	#lval	#rval	FALSE
jle	#lval	#rval	FALSE	jle	#lval	#rval	FALSE
jle	#lval	#rval	FALSE	jgt	#lval	#rval	TRUE
jge	#lval	#rval	TRUE	jge	#lval	#rval	TRUE
jge	#lval	#rval	TRUE	jlt	#lval	#rval	FALSE
jge	#lval	#rval	FALSE	jge	#lval	#rval	FALSE
jge	#lval	#rval	FALSE	jlt	#lval	#rval	TRUE
All other inputs return "undefined"							

Table 1: Lookup Table for Predicate Algebra.

The combination of partial redundancy elimination, predicate assertion propagation, and static predicate prediction is repeated until there are no new changes. Each data-flow phase removes its own redundancies, and in doing so, exposes new redundancies to be removed by the next phase. Partial redundancy elimination removes data dependencies that might inhibit edge removal, whereas static predicate prediction exposes newly redundant computation.

6.2 Peephole Optimizations

During each round of the redundant predicate optimization, we perform peephole optimizations on code within each basic block. For example, an ALU operation with an identity may be removed. A load from a scratch memory location preceded by a store to the same location may be changed into a copy operation. An add or subtract immediate instruction followed by an indirect load may be merged with the built-in index calculation.

Next, we use copy propagation to track computations on constants as they move through the control-flow graph. When we have register-register operations in which one of the registers is a known constant, we can transform the operation into its equivalent register-immediate form (provided that either the operation is commutative or the transformation does not change the order). When both values (either both registers or the register in a register-immediate instruction) are known, we may perform constant folding to turn the instruction into a load-immediate of a constant value.

```
        lh    [12], r0
        jne   r0, #ETHERTYPE_IP, L19
        lb    [23], r1
        jne   r1, #IPPROTO_TCP, L19
        lh    [20], r2
        and   r2, 0x1fff, r3
        jne   r3, 0x0, L19
L7:     li    #13, r4
        lb    [14], r5
        and   r5, 0xf, r6
        lsh   r6, 0x2, r7
L11:    add   r4, r7, r8
L12:    lb    [r8 + 14], r9
L13:    li    #7, r10
        and   r9, r10, r11
L15:    li    #0, r12
L16:    sub   r11, r12, r13
        jeq   r13, 0x0, L19
        ret   #TRUE
L19:    ret   #FALSE
```

Figure 6: Unoptimized code for "tcp[13] & 7 != 0".

As seen in Figure 6, these optimizations play an important role in minimizing the computation performed. Line 7 shows a *load immediate* instruction that is used in line 11 to load the 13^{th} byte of the TCP header. Since *add* is a commutative operator, we can replace the reference to $r4$ with the immediate value 13 and change the instruction to an `add immediate`. However, since line 11 is followed by a *load byte indirect* instruction on line 12, we can just fold in the immediate 13 into the index of the *load byte indirect* (to get 27) and remove line 11 from the code.

On line 13, we notice another *load immediate* that is used on the next line. Since *and* is a commutative operator, we can perform constant propagation again and replace the reference to $r10$ with the immediate 7. On line 15, there is another *load immediate* that may be removed by constant propagation. But after its substitution, line 16 becomes a *subtract immediate* instruction — subtracting the constant #0 from $r11$. We notice that this is an ALU operation by an identity, and therefore can be removed completely. Figure 7 shows the code after all of these peephole optimizations have been performed.

6.3 Lookup Table Encapsulation

The example in Figures 4-5 showed how redundant loads can be removed. These opportunities arise often in expressions that check a packet field against a set of possibilities, as in

```
      lh      [12], r0
      jne     r0, #ETHERTYPE_IP, L14
      lb      [23], r1
      jne     r1, #IPPROTO_TCP, L14
      lh      [20], r2
      and     r2, 0x1fff, r3
      jne     r3, 0x0, L14
      lb      [14], r5
      and     r5, 0xf, r6
      lsh     r6, 0x2, r7
      lb      [r7 + 27], r9
      and     r9, 0x7, r11
      jeq     r11, 0x0, L14
      ret     #TRUE
L14:  ret     #FALSE
```

Figure 7: "tcp[13] & 7 != 0" after peephole optimization.

ip src host A or B or C. The code generator output for this expression is:

```
      lh      [12], r0
      jne     r0, #ETHERTYPE_IP, L4
      ld      [26], r1
      jeq     r1, #A, L13
L4:   lh      [12], r2
      jne     r2, #ETHERTYPE_IP, L8
      ld      [26], r3
      jeq     r3, #B, L13
L8:   lh      [12], r4
      jne     r4, #ETHERTYPE_IP, L12
      ld      [26], r5
      jeq     r5, #C, L13
L12:  ret     #FALSE
L13:  ret     #TRUE
```

After the peephole optimization and redundancy elimination phases have completed, the filter has been reduced to the following:

```
      lh      [12], r0
      jne     r0, #ETHERTYPE_IP, L6
      ld      [26], r1
L3:   jeq     r1, #A, L7
      jeq     r1, #B, L7
      jeq     r1, #C, L7
L6:   ret     #FALSE
L7:   ret     #TRUE
```

Note the contiguous sequence of conditional branches starting at line 3. We can optimize this linear chain of conditional branches, especially when the chain is long, by arranging it into a lookup table instruction. In general, to identify potential lookup tables, we traverse the control-flow graph looking for chains of blocks containing only conditional branches. Lookup table chains have the following properties: the chain's backbone is linked by all false or all true branches; all of the other branches point to the same exit node; each element of the chain dominates the rest of the chain; and all of the conditional branches in the chain test the same value. The example code after lookup table enscapulation is shown below:

```
      lh        [12], r0
      jne       r0, #ETHERTYPE_IP, L4
      ld        [26], r1
      or table  r1, #A, #B, #C, L5
L4:   ret       #FALSE
L5:   ret       #TRUE
```

While this approach finds most of the lookup tables, we find that we can expose more lookup table chains by simply reordering the constituent nodes of a more general chain. However, we may only reorder a node if there are no data dependencies that would be altered. We ensure this by requiring that the block to be moved be empty of all computation, save the final conditional branch. This is not as restrictive as it sounds, due to the effectiveness of our partial redundancy elimination.

Once the lookup tables have been abstracted, we will use heuristics (described later) to turn them into combinations of linear search, binary search and hashtable lookup. Thus, we can incorporate the core design structure and optimizations of PathFinder and DPF as a low-level optimization at the tail end of our optimization framework.

6.4 Register Allocation and Assignment

Before we can run our intermediate code on the BPF+ virtual machine, we have to map the virtual registers that remain in the optimized code into the 32 real registers available in the virtual machine.

We use a graph-building algorithm to perform this task. Each register is represented by a node in a graph. For each register, we compute a liveness range (i.e., a lifetime), which is the list of basic blocks between a register's definition and its last use. When two registers have overlapping lifetimes, we place an edge between them. This results in an *interference graph*. The registers in a connected subgraph of the interference graph have lifetimes that interfere with one another, although they might not all be live at the same time.

Each subgraph's virtual registers may be assigned to physical registers independently of the other subgraphs because their lifetimes do not intersect. Two virtual registers in a subgraph may be assigned to the same physical register if there is no edge between them. We use a simple graph coloring scheme to assign physical registers to each register.

The size of each subgraph is typically small and is generally bounded by the size of the largest predicate. Registers often have short lifetimes because after optimization, their predicates are computed and used only once. In fact, most registers are live in only one basic block. Those that live longer tend to occur in OR and AND chains which have already been collapsed into lookup tables by the lookup table encapsulation phase.

7 The Back End

7.1 Safety Verifier

Since the BPF+ filter code interpreter is run in a protected domain, the validity of the program must be checked. A user task must be prevented from installing a program that would execute an infinite loop, or would cause memory faults by reading, writing, or jumping out of bounds.

In a program, a loop is represented as a jump to a previously executed piece of code. In most correct programs, each iteration of the loop will check a predicate to determine whether to continue or exit out of the loop. However, in general, the value of this predicate cannot be predicted at compile-time, and is often dependent on the inputs to the program. Since any program that runs in a protected domain must terminate, and since the protected domain should not trust user code, we must be able to identify which programs will loop forever and which will terminate. Consequently, the protected domain must solve the *halting problem* when accepting a filter program. In general, this is impractical, but by adopting fairly benign restrictions, verification can be made trivial. Namely, filter programs must be acyclic, with all branches forwardly directed.[8]

Further verification entails checking that all opcodes are valid, that all jumps are forward and within bounds, that the terminating operation is a return instruction, and that all reads and writes to memory are within bounds. If a malicious filter program were allowed to indiscriminantly read or write data, it could corrupt the protected memory space. In BPF+, loads and stores to scratch memory are indexed by an immediate, thus, we can verify their validity during this phase. However, since we cannot prove what the bounds on an indirect load from packet memory will be, we employ runtime bounds checks on each load to ensure safety. If any load tries to read out of bounds memory, the filter is stopped and the packet is discarded.

7.2 JIT Assembler

Once the filter program has passed the safety verifier, it may be run in the BPF+ virtual machine or may be JIT assembled into native code. The speed advantages of an assembled filter program should be clear, and indeed, our results show that assembled programs run up to 6 times faster than their interpreted counterparts on an UltraSPARC IIi processor.

There are two phases of JIT assembly. First, we translate the lookup table abstractions into an optimized sequence of linear, binary or hash checks of the values inside. Then, since the target machine often has tighter register availability constraints than the BPF+ virtual machine, we perform another phase of register assignment.

[8] Any acyclic program can be expressed using only forward jumps.

7.2.1 Lookup Table Translation

The first stage of the BPF+ assembler translates each lookup table instruction into an optimized sequence of native code instructions. A naive approach might just translate the table into a linear sequence of predicates, but this is no better than what we started with. When there are more than several predicates, the overhead causes the lookup to slow down linearly with the number of predicates.

Consequently, we may turn the table into a balanced binary tree based on the values in the table. This would have the effect of making the average case lookup equal to the worst case lookup. The overhead of the lookup would slow down as the log of the number of predicates.

As a third alternative, we can turn this table into a hashtable with a perfect hash function (since we know all of the entries at compile-time) and get constant time access. For small numbers of predicates, the overhead involved in computing the hash function may be too great, but for larger tables, this approach works well.

How do we know which one to pick? Currently, we use a static heuristic based on an evaluation of how each representation performs as a function of the number of predicates. Recent papers by Yang, Uh, and Whalley [21] [22] suggest the use of a profile-driven approach to determine whether to implement multiway branches using hash lookup, or to simply reorder the branches in a sequential lookup to reduce the dynamic number of branches encountered during program execution.

7.2.2 Register Assignment

The native code phase of register assignment is somewhat more delicate than the first phase, due to the greater register pressure found in most architectures. In an UltraSPARC with register windows, our simple assignment scheme is constricted to the use of 20 registers. An assembler for an x86 is constrained to only six.

If there are enough registers in the native code to run a particular filter directly, we skip this second register assignment phase. However, when we must compress a filter's use of registers, we rerun the register assignment algorithm used before with one change. Instead of using liveness ranges that are sets of basic blocks, we construct a register's lifetime as the set of pseudo instructions between its definition and last use. This finer granularity lets us reuse registers within a basic block, thereby minimizing our use of registers subject only to data dependencies.

If we still cannot fit the filter in the specified smaller number of registers, we must take the drastic step of spilling extra values to memory. We use a graph coloring algorithm to identify where spills must take place and add in the auxiliary code for spilling and restoring the data values.

8 Evaluation

To demonstrate the efficacy of our compiler and optimization framework, we have built all of the components described herein, culminating in a comprehensive implementation of the BPF+ architecture. In this section, we measure the performance characteristics of the BPF+ compiler — its ability to generate and optimize BPF+ byte codes, and the speedup in filter execution that we attain from JIT assembly. We also compare the effectiveness of our global data-flow optimization against the optimizations performed by an optimizing C compiler. We show that for the packet filter application, our optimizations are far more effective than those utilized by the C compiler.

Our experiments illustrate several measures of performance which we think have not been addressed in earlier work. In particular, we draw a distinction between measurements of filters that use independent high-level predicates and measurements of filters that use predicates which may be coalesced into a lookup table.

Our experiments were run on a Sun Ultra 10 workstation with a 300 Mhz Ultra IIi processor. One hundred thousand trials were run per experiment;[9] the running time for each filter was measured with the CPU tick register, enabling us to get accurate cycle counts of the time spent on each individual filter.

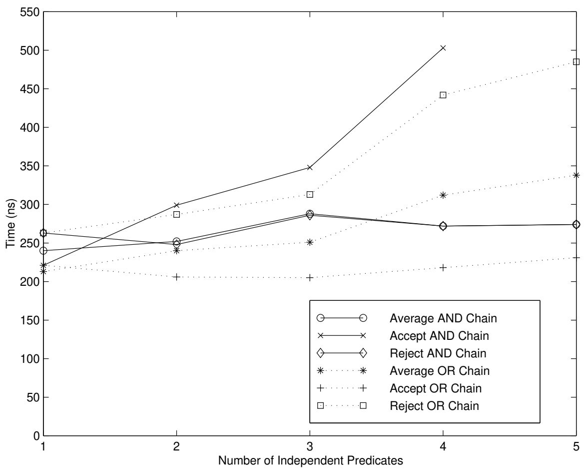

Figure 8: Time to recognize packets with various numbers of independent predicates. Lower numbers are better.

In Figure 8, we show the speed of filtering various numbers of independent predicates – TCP, src A, dst B, port C, and network D connected in a chain by either "and" or "or". There are six measurements shown (of the optimized JIT assembled filters), three showing the average, accept and reject times for the chains linked together by "and", and three showing the same results for the same chains linked together by "or". As expected, the time to reject an OR chain has the same upward trend as the time to accept an AND chain.[10]

In contrast, the time to accept an OR chain stays low because the earlier predicates, if matched, halt the filter and return TRUE immediately. The average time reported for both AND and OR chains are similar and hover between 200 ns and 300 ns. This is comparable to filter speeds reported in the literature.

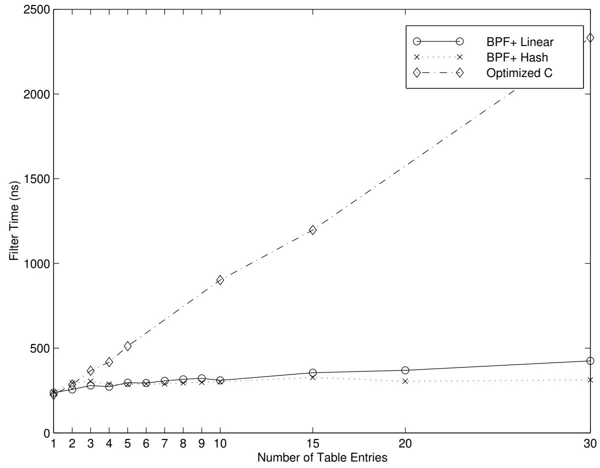

Figure 9: Time to recognize TCP packets with various numbers of source hosts. Lower numbers are better.

In Figure 9, we show, for non-independent predicates, the speed of filtering when a lookup table is implemented by a linear sequence of conditional branches, an O(1) perfect hash function (each hash table entry has one conditional branch to ensure a match), and the equivalent filter coded in C and run through the GCC (egcs-2.91.60) optimizer. BPF+ performs better than C in both cases, primarily due to BPF+'s redundant predicate elimination. Since redundant predicates do not often occur in user-level C code, GCC does not perform the elimination optimization that BPF+ does. In addition, the translation of filter code into native machine code has lowered the penalty that we pay for increased numbers of conditional branches in the final filter.

In addition to these measures, we also examine the speedup attained when using the various optimizations found in BPF+. In Figures 10 and 11, we show the filter times for unoptimized interpreted, optimized interpreted, unoptimized JIT assembled, and optimized JIT assembled packet filters for both independent and non-independent predicates.

For independent predicates, the speedup grows dramatically (from 3.5x to 9x) as the number of filters increases, which shows the effectiveness of our optimization algorithms and JIT assembler. The speedup due to optimization alone

[9] The packet trials are from a capture session that recorded normal network traffic in the UCB computer science domain.

[10] The last "Accept AND chain" measurement is left off the graph because the particular expression was never accepted.

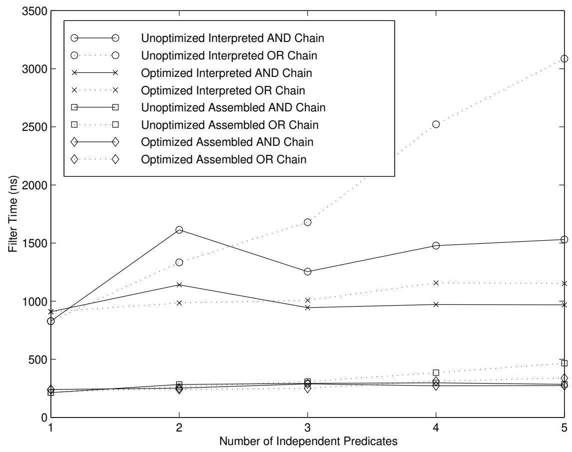

Figure 10: Time to recognize TCP packets with various numbers of independent predicates. Lower numbers are better.

varies from 1.3x to 2x for unoptimized code, and from zero speedup to 1.4x for optimized code. The speedup due to the JIT assembly by itself varies from 3.9x to 6.6x for unoptimized code, and from 3.3x to 5x for optimized code.

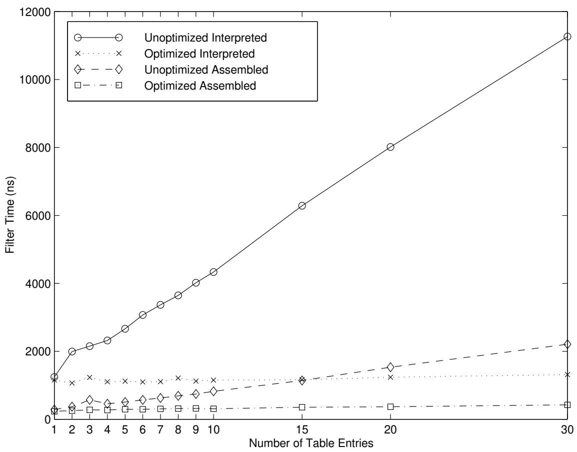

Figure 11: Time to recognize TCP packets with various numbers of source hosts. Lower numbers are better.

When we look at the non-independent predicates, we see a more dramatic story. The unoptimized, interpreted filter shows striking evidence of the naive code generation's production of redundant predicates. The optimized, interpreted filter strips out almost all of these redundancies. The trends for both assembled filters are the same as the interpreted filters, but the overall running time is much improved. The speedup due to optimization varies from 1.1x to 8.6x for interpreted code, and from 1.2x to 5.2x for assembled code, while the speedup due to assembly runs from 4.1x to 5.5x for unoptimized code, and from 2.6x to 4.9x for optimized code.

While the improvement for non-independent predicates is more dramatic than for the independent predicates, we feel that their use in combination more accurately reflects the type of filters used by the network community. For example, on two large (27 and 29 predicates) filters used daily by Vern Paxson at Lawrence Berkeley National Laboratory, we see speedups of 32x and 36x between unoptimized, interpreted code and optimized, assembled code.

Overall, our measurements indicate that optimization is an important factor in packet filter performance, especially when compiled from a high-level source language such as the one BPF+ uses. The template-matching heuristics that PathFinder and DPF use are effective in discovering lookup tables when filters are written in a low-level way, however, they will not work for more general types of filters. We had hoped to compare our results to those reported by the current state-of-the-art, DPF, but did not have access to their experimental data or their platform. However, if we adjust their published measurements of speed vs. table size to account for differences in the processor speed, our data suggests that the performance is similar.

9 Future Work and Summary

There are several different directions to explore in future development of BPF+. We have chosen to use a high-level functional predicate language based on *tcpdump*; we could add primitives that side effect the store to implement user-level state variables and enable user-level demultiplexing. We might also add the ability to specify large tables of packet information to be matched in a filter.

In the BPF+ virtual machine instruction set, we would like to add the ability to use backward branches, in order to allow loops in the code. This would provide the ability to parse IPv6 "stacked headers" as well as the ability to implement other, more general control structures. Not only would this change have an impact on the implementation of our optimization algorithms, but it would also impact the ability of the safety verifier to ensure that code migrated across the protection boundary does not enter into an infinite loop. Necula's proof-carrying code work [17] appears to be a suitable framework in which to define and enforce a semantics for the protected execution of packet filters.

BPF+ packet filters currently return a boolean true or false value. Some users have expressed interest in a more complicated return result that indicates which of the predicates in the filter matched the packet. This is a hard problem because the code generator creates many more predicates than are specified by the user. After passing through the optimizer, there may not even *be* a mapping from the resulting predicate expression back to the user-specified expression. However, for many purposes, just knowing selected information about the packet may suffice, e.g. in an intrusion detector that uses many different ways to detect intruders, if a packet source matches the source found in a large intruder table, we might just want to know the packet's source address, and not care about any of the other predicates that may have matched.

Our experience with BPF+ has proven that you can start with a high-level language and can compile and optimize packet filters into an efficient implementation. Through the novel application of the "redundant predicate elimination" global data-flow optimization, our high-level boolean predicate language can be compiled, optimized, and optionally JIT assembled, into a form that performs as well or better than the current state-of-the-art packet filter packages.

10 Acknowledgements

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] Mary L. Bailey, Burra Gopal, Michael A. Pagels, and Larry L. Peterson. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, November 1994.

[3] J. Cocke and J. Schwartz. *Programming Languages and Their Compilers*. NYU, Courant Inst., TR., Second Revised Version, April 1970.

[4] J. R. B. Cockett and J. A. Herrera. Decision tree reduction. *Journal of the ACM*, 37(4):815–842, October 1990.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[6] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of SIGCOMM '96*, pages 53–59, Stanford, CA, August 1996. ACM.

[7] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[8] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[9] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

[10] Van Jacobson, Craig Leres, and Steven McCanne. *pcap(3)*. Available via anonymous ftp to `ftp.ee.lbl.gov`, June 1989.

[11] Van Jacobson, Craig Leres, and Steven McCanne. *tcpdump(1)*. Available via anonymous ftp to `ftp.ee.lbl.gov`, June 1989.

[12] T.V. Lakshman and D. Stiliadis. High speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of SIGCOMM '98*, September 1998.

[13] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, pages 259–269, San Diego, CA, January 1993. USENIX.

[14] R. S. Michalski. Designing extended-entry decision tables and optimal decision trees using decision diagrams. Technical Report UIUCDCS-R-78-898, Univ. of Illinois at Urbana-Champaign, Urbana-Champaign, Ill., 1978.

[15] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987. ACM.

[16] B. M. E. Moret, M. G. Thomason, and R. C. Gonzalez. The activity of a variable and its relation to decision trees. *ACM Transactions on Programming Languages and Systems*, 2(4):580–595, October 1980.

[17] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Wa., October 1996.

[18] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the Seventh USENIX Security Symposium*, San Antonio, TX, January 1998. ACM.

[19] J. R. Quinlan. Inductive inference as a tool for the construction of efficient classification programs. In T. Mickalski, R., Carbonell, J. and Mitchell, editor, *Machine Learning: an Artificial Intelligence Approach*. Tioga, Palo Alto, CA, 1983.

[20] V. Srinivasan, George Varghese, Subash Suri, and Marcel Waldvogel. Fast scalable algorithms for level four switching. In *Proceedings of SIGCOMM '98*, September 1998.

[21] G.R. Uh and D. B. Whalley. Coalescing conditional branches into efficient indirect jumps. In *Proceedings of the International Static Analysis Symposium*, pages 315–329, September 1997.

[22] Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Improving performance by branch reordering. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 130–141, Montreal, Canada, June 1998.

[23] Masanobu Yuhara, Brian Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 153–165, San Francisco, CA, January 1994. USENIX.