Applying General Compiler Optimizations to a Packet Filter Generator

Andrew Begel
Dept. of Computer Science
UC Berkeley
Berkeley, CA 94720

abegel@cs.berkeley.edu

Abstract

This paper describes the architecture of the BSD Packet Filter package, which consists of a code generator, optimizer and virtual machine. The ability to do per-packet statistical sampling was added to the package. In addition, several modifications were made to the optimizer to increase the quality of the code output. While these modifications don't produce drastic improvements in the resulting code, they do illustrate the advantages of using an SSA-like representation for the intermediate byte code.

1. Introduction

A packet filter is a tool used to monitor and control network traffic. It is able to check every packet that comes by its network port and if it matches a predicate, pass it to the next higher level of control. Packet filters may be installed in user space, kernel space, network cards, or even inside routers.

The ability to monitor and filter network traffic is important for diagnostic and security reasons. If the network gets congested, an administrator can use a packet filter to search for the spamming packets and optionally filter them out of the network. If a user wishes to make sure that their WWW browser is not sending out their credit card information unencrypted, they can use a packet filter to monitor all TCP packets leaving their computer whose destination is WWW port 80. Packet filters can be deployed in routers to provide filtering for security. This past March, a teardrop attack was launched on all of the computers on the UC Berkeley campus. This attack disabled most of the Windows NT machines by sending a fragmented packet to the DNS port. Network administrators used a packet filter to block all packets from the supposed source of the attack for several months.

Packet filters can be implemented in a number of ways, but all implementations must be fast. A typical workstation connects to a network at 100 Mb/s. Assuming an average packet size of around 100bytes, a typical filter must be fast enough to process 100,000 packets per second. Installing a packet filter in an OC-12 router running at 622 Mb/s must be able to filter 622,000 packets per second, and still route network traffic!

Packet filters must also be able to be developed and deployed relatively quickly. Each filter works on the basis that examination of a packet header is enough to determine the packet's fate. Therefore, being able to easily describe properties of the packet header that should be in the predicate is just as important as speed.

A final requirement on packet filters is that they be safe. If a packet filter is going to be downloaded into OS kernels and routers, it better not crash or in some way compromise system

integrity. Either the packet filters must be created in a safe language, or they must be sandboxed to prevent unauthorized access.

This paper discusses the BPF code generation and optimization architecture. It adds a new sampling feature to BPF and rearchitects the optimization infrastructure to achieve greater levels of code elimination and simplification.

2. Related Work

The BSD Packet Filter (BPF) package [4], created by S. McCanne and V. Jacobson at LBL in 1992, introduced a virtual machine for running safe packet filters. Using a simple predicate language, users could specify properties of packet headers, such as *TCP and SRC beech.cs.berkeley.edu*, which would be compiled into instructions for the VM. The VM ensured safety by prohibiting backward branches, checking forward branches to see whether they stayed within the code block, and checking for division by 0 and writes to read-only registers.

BPF was based on work done by Mogul, Rashid and Accetta on kernel-level packet filters [5]. The packet filter used a stack-based virtual machine and was general enough to be programmed for new packet types as they became available.

Since then, there have been several new packet filters that improve upon BPF, each one better than the last. Mach Packet Filter [6] was an extension to BPF that incorporated caching and the ability to handle fragmented packets. It claimed to be over four times faster than BPF due to its caching of filter results across network ports.

Pathfinder [1] implements a declarative language for filter specification. Each predicate is a tuple of <*offset, length, mask, value*> which may be joined together. Pathfinder takes all of the predicates and unifies them into a prefix tree of patterns. This tree, which connects patterns in a DAG linked by AND and OR nodes, minimizes the number of times that a particular pattern must be checked. If there are two predicates that have the same offset/length/mask triple but differ in the value, they are merged into a hashtable which maps values into edges in the DAG. Pathfinder also employs caching and supports packet fragmentation. It performs about twice as fast as MPF in its software-based implementation.

DPF [3], from Engler and Kaashoek at MIT, is the latest work on packet filters. It aggressively optimizes packet filters using dynamic code generation and gets about 13-26 times the performance of Pathfinder. This means that DPF is 100-200 times as fast as the original BPF packet filter! Much of this performance comes from the compilation to native machine code. The other packet filters compile into a VM, as BPF does. DPF employs a declarative filter language similar to Pathfinder's. It also uses similar heuristics for code generation such as using a trie for merging filters and using hashtables for multiple values for a particular offset/length/mask triple. It also coalesces small neighboring patterns into larger ones to take advantage of the large machine word size.

Both PathFinder and DPF use a trie data structure to remove redundant expressions in multiple packet filters. While this is a good scheme in this particular situation, it isn't general enough to handle all types of filters. In particular this trie lacks support for logical negation, making the filter *NOT TCP* extremely hard to express. We feel that BPF's compiler optimization approach is more general and can optimize more situations than both Pathfinder and DPF's trie.

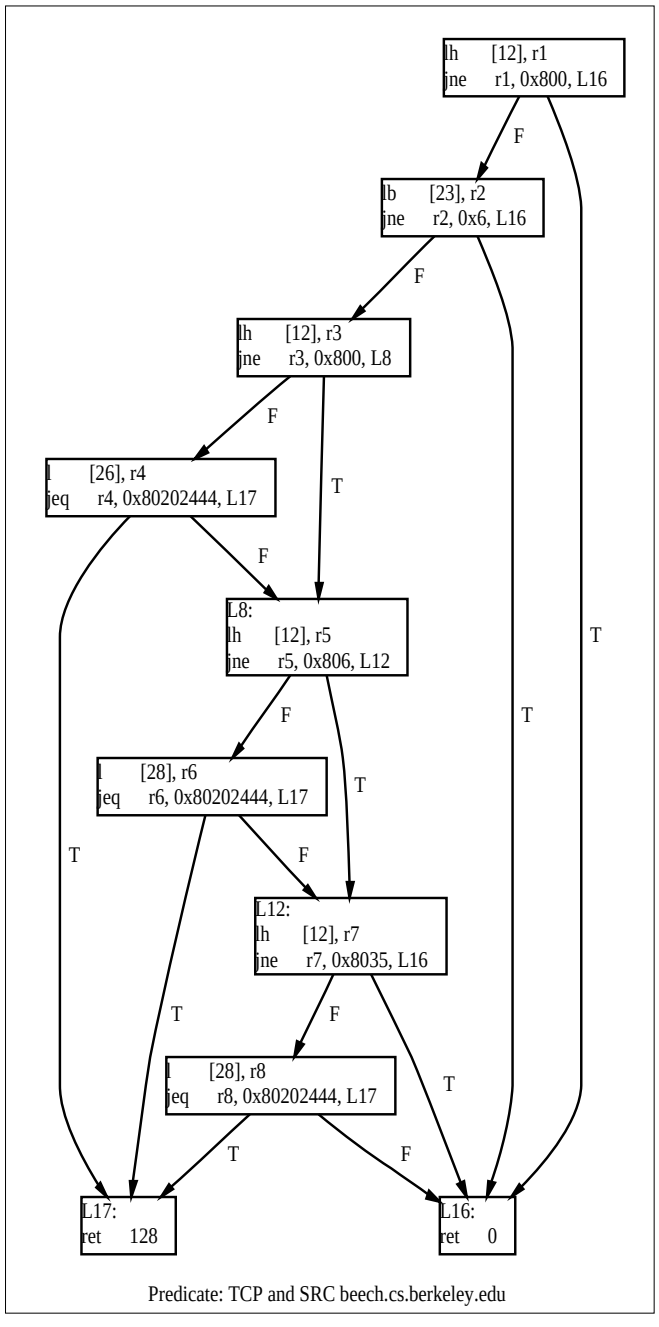
3. BPF Code Generator

BPF exposes a simple API to the end-user. The user uses BPF to generate code and compile packets predicate to an internal VM code specification. This may be optionally optimized to produce simplified VM code. Then, this filter is installed into the BPF VM and packet filtering begins on the network. Each time the predicate is satisfied, a user-defined routine is called with the matching packet as an argument.

BPF packet predicates can be described with a small grammar. The user specifies a series of individual predicates that can be joined together by ANDs and ORs and optionally negated. These individual predicates can be simple packet types, such as IP or TCP, or they can be more complex checks for specific port numbers and source and destination hosts. To simplify the code generation, each predicate is generated independently and joined together by AND points and OR points.

For example, to generate code for *TCP*, we must first do a check for *IP*. We can't have a TCP packet without it being an IP packet as well. So we generate code for IP and then check byte 23 for a 0x6, which will indicate whether this IP packet is also a TCP packet. In another example, to generate code for *SRC beech.cs.berkeley.edu*, we first do a check to see the type of the packet. We won't know where to look for the source address unless we know whether this is IP, ARP or a RARP packet. If it is IP, then we look for the source at byte 26. If not, then we look at byte 28.

When we combine predicates, the redundant checks become quite numerous. For example, *TCP and SRC beech.cs.berkeley.edu* generate four redundant operations that could potentially be optimized away. On larger predicates, the opportunities for optimization grow fast.



Predicate: TCP and SRC beech.cs.berkeley.edu

Here is the unoptimized source code for *TCP and SRC beech.cs.berkeley.edu*:

```
// Check for IP
          lh        [12], r1
          jne       r1, 0x800,
    L16
// Check for TCP
          lb        [23], r2
          jne       r2, 0x6, L16
// Check for IP again
          lh        [12], r3
          jne       r3, 0x800, L8
```

```
// Check for SRC beech.cs.berkeley.edu
          l         [26], r4
          jeq       r4, 0x80202444, L17
L8:
// Check for ARP
          lh        [12], r5
          jne       r5, 0x806, L12
// Check for SRC beech.cs.berkeley.edu
          l         [28], r6
          jeq       r6, 0x80202444, L17
L12:
// Check for RARP
          lh        [12], r7
          jne       r7, 0x8035, L16
// Check for SRC beech.cs.berkeley.edu
          l         [28], r8
          jeq       r8, 0x80202444, L17
L16:
// Return false
          ret       0
L17:
// Return true
          ret       128
```

The code generator module generates code for the virtual machine. It uses an unbounded register allocation scheme, but when a register is finished being used, it is recycled on a pushdown stack. Virtual machine instructions consist of an opcode, two registers and an optional immediate value. The VM has a RISC-like load-store architecture with an integrated compare-jump instruction. Filters may access data from three places: registers, the packet header, and a persistent store. This last memory is used to record statistical information across packets in order to do sampling.

One contribution of this project was to add a sampling feature to BPF. One can use the predicate *every* <*n*> <*pred*> to sample 1/nth of the packets meeting the <*pred*> predicate. The code generator makes use of the persistent store to keep a counter which increments every time the predicate is true. When this counter wraps around to zero, the sampling predicate becomes true.

Another notable feature of this VM is that there are no function calls, no stack, and backward branches are illegal. This means that looping is not possible. Lack of a way to loop ensures that all packet filters will eventually terminate. In addition, there are no register-relative jumps, which means that all jump targets are known at filter-load time and can be verified not to jump backwards.

4. BPF Optimizer

This section will describe the original BPF optimizer and the modifications made by this project to increase its effectiveness. The optimizer begins by doing a topological sort on the blocks in the control flow graph. It then repeatedly computes the dominator tree, the backwards control-flow transitive closure, the in-edges of each node, def-use chains for each register and memory location, and the edge-dominator graph for all control flow edges. Finally, it performs several control-flow optimizations on the resulting information until no more changes are detected.

These control flow optimizations consist of constant propagation, partial evaluation, common subexpression elimination, dead store elimination, and conditional elimination. The optimizer uses Cocke and Schwartz [2] value numbering to keep track of values stored in the registers. Each block keeps track of the values that are used inside. The control flow is used to flow values between basic blocks. If there are no merge conflicts, the next basic block receives the values from all of its predecessors. If any conflicts arise, those values are removed in the new block.

When the optimizer notices a statement using a previously computing value, it will substitute the value in place of the operation. If the value is constant, it records that in an auxiliary data structure. During constant propagation, if any value used in an expression is constant, the immediate value is substituted for its use. During partial evaluation, if the optimizer notices that an arithmetic operation is using two constant values, it will evaluate the expression at compile time and replace the operation with a load immediate and the result of the calculation. If only one operand is constant, the register-register ALU operation will be replaced by an equivalent register-immediate operation. NOP operations, such as add zero to a register or multiply a register by one, are removed.

During dead store elimination, the optimizer traverses the control-flow graph and notes the last position that a register is defined. If there are no subsequent uses, the definition is removed.

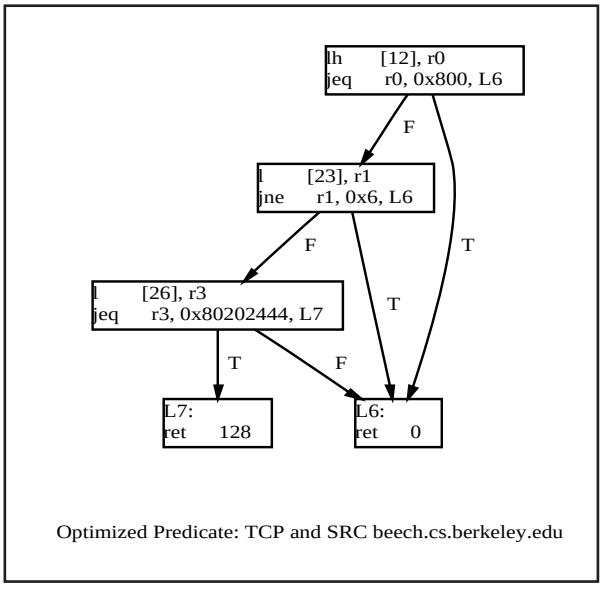
The most interesting optimization is the conditional elimination. This optimization is the one that gets rid of redundant compare-jumps. It operates on edges and edge-dominators instead of the traditional node and node-dominators, thus allowing the optimizer to remove extraneous edges from the graph. Once the extraneous edges are removed, the optimizer is able to elide complete basic blocks if they have no more incoming edges.

The first part checks if an edge points to a block whose outgoing true and false edges point to the same child. If so, the block's compare-jump instruction is unnecessary. If there are no use conflicts (the skipped node didn't redefine any subsequently used variables), the original edge can be retargeted to skip the useless predicate block and point straight to the final target.

The second part of the optimization allows us to eliminate redundant compare-jumps. For each edge-dominator of the current edge, check if it is equivalent to the current edge. Two edges are equivalent if the compare-jump used to take the edge-dominator is the same as the one used to take this edge. If we find a match, we know that we've already computed the answer to the original comparison, thus we know what the answer will be on this compare-jump. If the register being compared has not been redefined since the edge-dominator used it and there are no register definitions in this block that are used in subsequent blocks, we can elide the whole node and retarget the original incoming edge to the correct grandchild block.

Here is the optimized version of the earlier *TCP and SRC beech.cs.berkeley.edu* filter:

```
// Check for IP
          lh        [12], r0
          jne       r0, 0x800, L6
// Check for TCP
          lb        [23], r1
          jne       r1, 0x6, L6
// Check for SRC beech.cs.berkeley.edu
          l         [26], r3
          jeq       r3, 0x80202444, L7
L6:
// Return false
          ret       0
L7:
// Return true
          ret       128
```



Optimized Predicate: TCP and SRC beech.cs.berkeley.edu

After running this optimizer on several test cases, it was noticed that several nodes weren't getting elided from the conditional elimination. These nodes had compare-jumps that were identical to previous compare-jumps, but had merge conflicts. In several cases there were extra incoming edges from paths that had redefined the register, resetting its value number to null. If the code generator would output VM code in SSA form, this redefinition wouldn't happen, and the node could be elided.

I modified the register allocator in the code generator to allocate new registers for every definition. The number of registers used in a packet filter went up dramatically to many more than the VM actually had, but this could be handled just before the final output with a final register allocation phase. Since all definitions go to new registers, I was able to eliminate large chunks of code from the BPF optimizer and simplify its logic.

In addition, the value numbering scheme was rewritten. Now that each register had a unique value, we only kept a small table to store them. Then, we went over the control-flow graph, and if a subexpression was computed twice, it got replaced by a reference to the register that computed it first (as long as the original definition dominated this use). Then, the optimizer performed copy propagation and dead-store elimination to get rid of the now, useless register which had computed the redundant value.

We also modified the value numbering to include loads from the packet memory. Since many of the compare-jumps were checking bits from the packets, our new optimization had the added effect of creating many identical copies of the original compare-jumps. For example, if we checked for IP twice, the original code generator would generate two definitions to two different registers. This optimization recognized that the values loaded from the packet memory were the same (packet memory is read-only) and rewrote the second load to use the value from the first register as long as the first load dominated the second. This bonus effect made the conditional elimination algorithm more effective and enabled it to elide more nodes than before.

5. Performance Measurements

I tested the optimizer against the unoptimized code generator to see how well it did. I wanted to test the optimizer against DPF and Pathfinder, but DPF didn't compile on our platform and we couldn't find an implementation of Pathfinder. This would have been very interesting because BPF is the only packet filter to use general compiler optimization techniques to optimize the

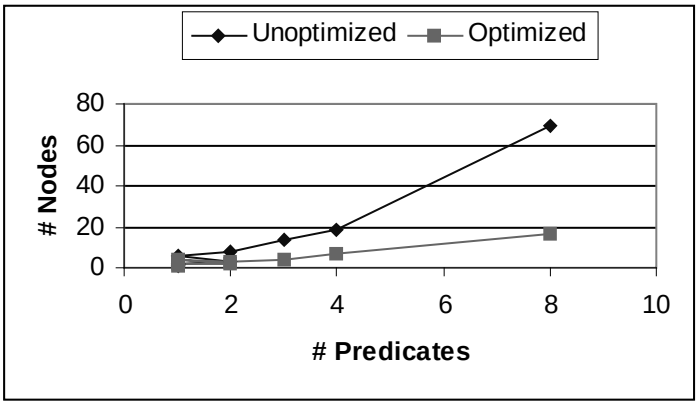


Figure 1: This graph shows how well the optimizer does at eliding control-flow nodes from several filters.

packet filter. Both Pathfinder and DPF use only heuristics to generate efficient code. A comparison between these two techniques would have been illuminating to see how well compiler optimization performs against the heuristics.

The two measures used to compare the optimized performance with the unoptimized performance are the control-flow graph size and BPF's speed at processing packets. The first measure shows how many nodes the optimizer was able to remove from the control flow graph. The second measure gives the reader an idea about how much effect those node elisions actually had on the resulting packet filter performance.

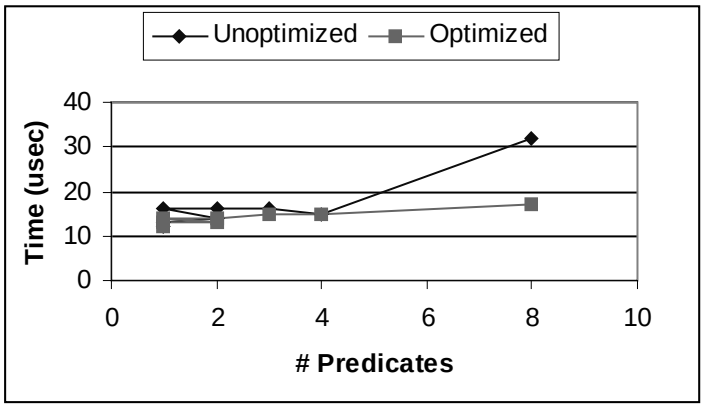


Figure 2 : This figure shows the actual effect of optimization on packet filter execution time.

The optimizer is quite successful at removing redundant nodes from the control flow graph of each filter. However, the performance gains from the pruning are less evident, as seen in Figure 2.

I tried to test the new optimizer against the old optimizer, but found that the new optimizations did not make much difference with respect to the measures here. The code generated was quite different between the two optimizers, but I can't find a measure that will distinguish between their code quality.

6. Future Work

At this point, the only thing left to implement is a register allocator to map virtual registers to actual registers in the VM. Since the packet filter code generator uses only a subset of the VM's largely general purpose instruction set, many general compiler optimizations aren't useful. This either implies that the VM instruction set could be simplified or that we need to think of more interesting things to do with this filter package.

A few ideas for new features for the BPF package come from some heavy users at LBL. In addition to the packet sampling, which was added in this project, they would like to see support for IPv6, the ability to modify packet filters in real-time (this would be very good for securing networks against attackers), the ability to tell which part of the filter matched the packet without having to recheck in user code, the ability to sample packets on a per-connection basis (the sampling feature implemented here is per-packet), and the ability to download large tables of net addresses and ports to include in the filter.

Most of the feature requests require modifications to the source language or the VM architecture. Only the last feature, which requests the ability to download large tables of filter

data into a packet filter, would involve some added optimization. Should the filter remain encoded as a sequence of comparisons, or should we switch to a hashtable representation? DPF dynamically chooses a representation based on how many comparisons must be done in the packet filter.

It seems that much of the speed advantage of DPF over BPF comes mainly from the compilation into machine code. BPF includes an assembler into Sparc machine code. With the current compiler optimizations, it is hoped that after compilation into native code, BPF will perform on par with or better than DPF.

7. Conclusion

This project introduced several modifications to the BSD Packet Filter package. In addition to adding a statistical per-packet sampling feature, the optimizer was modified to use an SSA-like form to expose more opportunities for conditional elimination and copy propagation. We feel that the use of general compiler optimizations make BPF a more robust filtering package than the ones currently out there. Once BPF employs a native code assembler, we feel that this package will be able to outperform all others.

References

[1] M.L. Bailey, B. Gopal, M.A. Pagels, L.L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. *In Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115-123, November 1994.

[2] J. Cocke, and J.T. Schwartz. *Programming Languages and Their Compilers*. Preliminary Notes. 2nd revised version. Courant Institute of Mathematical Sciences, 1970.

[3] D.R. Engler, and M.F. Kaashoek. DPF: Fast, flexible demultiplexing using dynamic code generation. *In ACM Communication Architectures, Protocols, and Applications*. SIGCOMM 1996.

[4] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. *In USENIX Technical Conference Proceedings*, pages 259-269, San Diego, CA, Winter 1993. USENIX.

[5] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. *In Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39-51, November 1987.

[6] M. Yahara, B. Bershad, C. Maeda, and E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. *In Proceedings of the Winter 1994 USENIX Conference*. 1994.