DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation

Dawson R. Engler, and M. Frans Kaashoek
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.

{engler,kaashoek}@lcs.mit.edu

Abstract

Fast and flexible message demultiplexing are well-established goals in the networking community [1, 18, 22]. Currently, however, network architects have had to sacrifice one for the other. We present a new packet-filter system, DPF (Dynamic Packet Filters), that provides both the traditional flexibility of packet filters [18] and the speed of hand-crafted demultiplexing routines [3]. DPF filters run 10–50 times faster than the fastest packet filters reported in the literature [1, 17, 18, 27]. DPF's performance is either equivalent to or, when it can exploit runtime information, superior to hand-coded demultiplexors. DPF achieves high performance by using a carefully-designed declarative packet-filter language that is aggressively optimized using dynamic code generation. The contributions of this work are: (1) a detailed description of the DPF design, (2) discussion of the use of dynamic code generation and quantitative results on its performance impact, (3) quantitative results on how DPF is used in the Aegis kernel to export network devices safely and securely to user space so that UDP and TCP can be implemented efficiently as user-level libraries, and (4) the unrestricted release of DPF into the public domain.

1 Introduction

Rapid demultiplexing is important in order to neither waste processing cycles nor add latency to end-to-end message time [1]. Application-specific demultiplexing is important so that applications can explore new protocols without kernel modification [18]. Previously, networking architects have had to trade flexibility for performance to demultiplex messages efficiently. For example, the packet-filter model [18] is flexible, but its cost has precluded its use in high-performance networking systems. Instead, networking systems use hand-crafted demultiplexing routines that can only be extended or altered by kernel architects [13, 23, 26]. The packet filter system described in this paper, Dynamic Packet Filters (DPF), achieves these twin goals of flexibility and performance by using a carefully-designed declarative packet-filter language that is amenable to aggressive dynamic code generation. DPF filters run 13–26 times faster than PATHFINDER filters, the fastest numbers reported in the literature [1, 17, 18, 27]. DPF's performance is equivalent to, and in some situations can even exceed, the performance of hand-coded demultiplexors.

Message demultiplexing is the process of determining which application a message is for. Like other packet demultiplexors, DPF uses packet filters to demultiplex messages. Packet filters are predicates written in a small safe language [18]. Logically, a packet filter examines all incoming network packets; those messages that satisfy its predicate are delivered to its associated application. This approach allows new protocols to be implemented outside of the kernel and then downloaded into the packet filter driver, greatly increasing flexibility [18].

Packet filters are interpreted, which entails a high computational cost. This cost is sufficiently high that, to the best of our knowledge, all high-performance networking systems except for the x-kernel project [1, 12] avoid them completely. For example, the Peregrine RPC system [13], the remote read/write model of Thekkath et al. [23], and the Active message model [26] all use hard-wired in-kernel demultiplexing routines. We show that applying dynamic code generation to packet filters can improve their performance to the point that they are equivalent in efficiency to (and in some cases faster than) hand-crafted in-kernel routines, while retaining all of the flexibility of the packet filter model.

The key in our approach to making filters run fast is dynamic code generation. Dynamic code generation is the creation of executable code at run time. Its most important property is that it allows the use of runtime information to improve code generation. Dynamic code generation (or *dynamic compilation*) is directly analogous to static compilation except that, since it occurs at runtime, it must be made much more efficient. Dynamic code generation can be used to compile a packet filter specification down to actual machine code. While dynamic code generation has been considered a possible mechanism for improving the performance of packet filters since their inception [18], it has been deemed too "complicated" to implement [18]. A contribution of our work is to show that dynamic code generation can be used in a straightforward manner.

We have designed, implemented, and tested DPF. User-level measurements show that DPF demultiplexes messages 25–50 faster than MPF [27] and 13–26 faster than PATHFINDER [1], the fastest packet filter demultiplexor in the literature. In addition, DPF scales better with the length of a filter than PATHFINDER, making it better suited for deep protocol stacks. We have also integrated DPF with Aegis, an exokernel operating system [10]. DPF allows Aegis to export the Ethernet interface safely and efficiently to applications; this structure allows all protocols to be implemented completely as libraries in user space. Measurements from Aegis's user-level UDP and TCP implementations, which use DPF for filtering packets, show that this structure can achieve performance as good as or better than hard-coded in-kernel implementations while retaining a

This work was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985 and by a NSF National Young Investigator Award.

```
(
  # Check Ethernet header
  (12:16 == 0x8) &&            # IP datagram?
  # Skip over ether header (14 bytes)
  (SHIFT(6 + 6 + 2)) &&

  # Check IP header
  (9:8 == 6) &&                # Check protocol : TCP is 6
  (12:32 == 0xc00c4501) &&     # Check IP src addr (192.12.69.1)
  # Skip past IP header (assume fixed sized; 20 bytes)
  (SHIFT(20)) &&

  # TCP header
  (0:16 == 1234) &&    # Check source port (2 bytes)
  (2:16 == 4321)       # Check destination port (2 bytes)
)
```

Figure 1: DPF filter to recognize TCP/IP packets.

high degree of flexibility.

The remainder of the paper is structured as follows. Section 2 discusses the design of DPF and its filter language. Section 3 describes DPF's implementation using dynamic code generation. Section 4 reports on DPF's performance and compares its performance to other packet filters. Section 5 describes how we use DPF to run UDP and TCP as user-level libraries. Section 6 relates DPF to other work. In Section 7 we conclude.

2 Dynamic Packet Filter

DPF's packet filter language is a declarative language that is used by protocols to describe the message headers that they are looking for. Similarly to the languages used in MPF [27] and PATHFINDER [1], it can handle a wide range of possible protocols and supports fragmentation.

A DPF packet filter is composed of a sequence of boolean comparisons (or *atoms*) linked by conjunctions. Logically, a filter's atoms are checked in order after packet reception (our declarative language avoids most side-effects, allowing re-ordering for efficiency). If all atoms are true, then the filter accepts and the packet is delivered to the associated application.

Figure 1 shows an example packet filter. Each atom is specified using the standard set of arithmetic operators, with the addition of the ':' and SHIFT operators. The ':' operator is used to read packet data. It takes as arguments a base (specified in bytes) and a size (specified in bits). The base can be either a constant or an indirect load from the message itself. The SHIFT operation shifts the base of the message pointer, allowing arbitrary predicates to be composed using relative addressing.

A new filter is merged into the set of active filters as follows:

1. Filters are written by clients, and then preprocessed into a low-level representation. This form is then shipped to the packet filter engine.

2. The packet-filter engine parses the filter into its constituent atoms, which are checked for syntactic and semantic errors and turned into an expression tree. As a performance trick, checking is elided for atoms that are duplicates of already vetted atoms. For example, when TCP filters are downloaded, only the first filter's atoms are subject to rigorous checking — the others can be verified with a byte comparison that ensures they are identical.

3. As in PATHFINDER, the set of active filters is stored in a trie data structure. New filters are merged into the trie along the path with the largest prefix match. The point of this operation is to collapse similar prefixes, eliminating unnecessary computation. This optimization is profitable due to the fact that many network communications use the same set of protocols (e.g., TCP/IP, UDP) and, as a result, their associated filters will check the same set of conditions. Merging the filters eliminates duplicate checks. Both equivalent and disjunctive atoms are merged. Two atoms are *disjunctive* if they share an identical expression tree on one side of the equality operator, but compare to different constants. When found, these atoms are merged and a hash table is used to determine which (if any) of the merged atoms were satisfied.

4. Atoms that were not merged into the main filter trie are connected to it with an "or" branch: during message demultiplexing these branches are checked one at a time until a filter accepts or no more branches remain. Currently these branches are sorted by reference count and checked from largest reference count to smallest. After a new or branch is created, the atoms that it contains are traversed, and an unlinked code fragment is generated from them (we discuss this operation below).

5. After merging the new filter into the master filter, if code was generated, the system re-traverses the master filter, copying the code associated with each changed atom into a linear block of memory. Cleanup is performed (e.g., jumps are back-patched and the instruction cache is flushed) and execution continues.

2.1 DPF/Operating system interface

The DPF system is modular and easily integrated into network subsystems. Its public interface (given in Table 1) is comprised of three functions: dpf_insert (used to insert a filter), dpf_demux (used to demultiplex messages), and dpf_delete (used to delete filters). System specific details of message alignment, minimum packet size, cache flushing, and a memory allocation function are provided as defaults that can be overridden.

3 Implementation

This section first discusses how DPF uses dynamic code generation and then reports on a number of optimizations that it performs.

3.1 Dynamic code generation

DPF exploits dynamic code generation in two ways: by using it to eliminate interpretation overhead by compiling packet filters into executable code, and by using filter constants to aggressively optimize this executable code. For portability, DPF uses the machine-independent VCODE dynamic code generation system [8].

Because an atom represents a single comparison, the code generated for it is mapped to a basic block of machine instructions: i.e., control enters from the top of an atom and leaves at the bottom (via a branch). Code generation is done with one pass over the atom's expression tree. This pass performs register allocation, simple delay slot scheduling, and instruction selection and emission. Register allocation and delay slot scheduling are modeled on Proebsting [19]. Instruction selection is a matter of translating the operators of the atom's expression tree to the instructions supplied by the underlying hardware. There is a direct mapping between these operations and machine instructions (e.g., "&" maps to `and`); as a result, instruction selection is simple. Finally, code is emitted using a set of simple macros. As noted above, each atom is a basic block, and ends with a jump and (possibly) a delay slot: these are left unfilled until the code is instantiated.

Operation	Description
int dpf_insert(struct dpf_ir *filter)	dpf_insert inserts filter filter. It fails if the filter is malformed, if memory allocation required to incorporate it fails, or, optionally, if the filter is already present. On success it returns a filter id.
int dpf_delete(int fid)	Delete filter fid inserted using dpf_insert. It fails if the filter does not exist.
int (*dpf_demux)(void *msg, unsigned nbytes)	The function pointer dpf_demux is called to demultiplex a packet. It takes as arguments a pointer to the message (msg) and a message size, in bytes (nbytes). It returns the id of the accepting filter, or "0" if no filter accepts.

Table 1: DPF Interface

Since dynamic code generation occurs at runtime, its main cost is the time required to emit code. We have taken care to ensure that the compilation process described in the previous paragraph happens only once — when an atom is first installed. By caching this code, subsequent recompilations of an atom (such as needed when new branches are added to the main filter trie) can be performed by copying the cached code.

To minimize the need to regenerate the demultiplexor function, DPF tracks the common case where a filter was merged with the main filter trie without adding new nodes or requiring modification to existing ones. Operationally, this situation translates to a filter only requiring entries to be added to the hash tables used to check disjunctions. As long as new hashing functions do not have to be created (e.g., to check for collisions, or to use a different hashing strategy), code does not have to be regenerated.

As we describe above, the current DPF implementation copies atom blocks into contiguous memory to construct the demultiplexing routine. An alternative approach would be to thread the code blocks using jumps. This method would provide cheaper code generation since forming the jumps between blocks is (usually) more efficient than copying the several words of memory containing atom instructions. However, it makes demultiplexing routines less efficient since control would not "fall through" from one atom to the next but, rather, would have to be forcibly redirected using a jump. We intend to investigate this alternative as we scale DPF to support environments with tens of thousands of filters,

3.2 Optimizations

Packet filter optimizations can be either intra-filter [17] (within a filter) or inter-filter [27, 1] (between filters). Dynamic code generation is an intra-filter optimization. In this paper, we are interested in isolating the effects of dynamic code generation. To this end, we re-implemented the applicable inter-filter optimizations used by PATHFINDER (the fastest packet filter engine in the literature) in the context of DPF; by holding this optimization framework constant, we can obtain a clear picture of the effects of dynamic code. To the best of our knowledge, the optimizations presented in this subsection are not implemented by PATHFINDER or any other current packet filter engine.

Runtime constants The simplest optimization is encoding filter constants directly into the instruction stream. For example, the filter presented in Figure 1 lists the source and destination ports as well as the source address. Because these values are constant over the filter's lifetime, they can be incorporated as immediate operands in the instruction stream itself, eliminating indirections.

Since many of these constants are filter-specific as opposed to protocol-specific (e.g., they are known only after a connection is established), a statically coded demultiplexor cannot access them directly. Instead, it must track such information in memory vectors (or hash tables) and access it via costly indirection. In other words, since DPF can exploit filter-specific constants, it can be faster than hand-crafted demultiplexors. For example, in Figure 2 the check for the source IP address (192.12.69.1) can be encoded directly in the instruction stream.

Fast disjunctions Disjunctions compare a given expression against a range of possible values. In systems such as MPF and PATHFINDER, these values are not known a priori, and so a general-purpose, possibly expensive hash function must be used, along with checks for collisions, etc. However, since DPF knows both the number and the actual values that must be compared, it can exploit this information to make disjunction resolution more efficient. For example, if the number of words to compare against is small (say 1–3), the hash table is completely eliminated and comparisons are performed directly (as a series of branches), saving the cost of hash table lookup. As another example, since the number and value of keys are known at runtime, DPF can select among several hash functions to obtain the best distribution and then the chosen function directly in the instruction stream. Furthermore, since DPF knows at code-generation time whether keys have collided, it can eliminate collision checks if no collisions have occurred.

The mechanics of this optimization are similar to the manner in which optimizing compilers treat C `switch` statements, which require that a given expression be matched against a constant. A small range of values is searched directly, sparse values are matched using binary search, and dense ranges are matched using an indirect jump.

Subsequent optimizations only require a preprocessing step, not complete dynamic code generation. However, they aid dynamic code generation's effectiveness.

Atom coalescing The first optimization in this category is the coalescing of adjacent atoms. For example, assuming word alignment, the checks of source and destination ports in Figure 1 examine two two-byte quantities that are contained in the same word, allowing us to collapse this comparison into a single operation:

```
# TCP header before coalescing
(0:16 == 1234) &&   # Check source port
(2:16 == 4321)      # Check destination port

# TCP header after coalescing
(0:32 == 283182290)  # 283182290 ==
                     # ((4321 << 16) | 1234)
```

Because this optimization eliminates a control flow change, it can save more than the few instructions required for the load, comparison and branch. For example, in architectures with branch prediction hardware, eliminating branches implies one fewer possible misprediction. Note that atom coalescing is most useful when the current alignment is known or can be predicted (see below).

Alignment estimation Because neither packet filters nor messages can be trusted, traditional packet-filter systems perform a number of checks to guard against errors. For example, packet-filter engines treat every load and store as potentially unaligned. Memory operations must therefore first be checked for proper alignment to eliminate unaligned access traps. We optimize this process by finding a lower bound for the alignment of the message-pointer base register. The general algorithm is to record the effects of every SHIFT instruction on the base message register. For example, this register starts word-aligned: if all subsequent shifts are by constants that are multiples of four, then this bound stays the same. However, if a shift by two bytes occurs, then the lower bound on this register becomes two.

As a side-effect of alignment information propagation DPF eliminates constant SHIFT operations by adding its operand to the message offset of subsequent atoms. This optimization saves about one addition instruction for each protocol header.

There are two complications to alignment estimation, both arising from indirect loads. If shifts are performed using indirect loads then the value loaded cannot be determined statically and must be pessimistically assumed to be only byte-aligned. Fortunately, computations are usually performed on this value before it is used to shift the message pointer. For example, consider a possible expression to shift past a variable-sized IP header:

```
# skip over variable IP header
(SHIFT((4:8 & 0xf) << 2))
```

Shifting the result to the left by 2 (multiplying by 4) ensures that the message register is word-aligned. The alignment heuristic works by traversing the expression tree of shift instructions looking for such operations (e.g., multiplication, left-shift, and masks).

Bounds-check aggregation To ensure protection, packet filters are constrained from accessing memory outside the current message. In traditional packet-filters, this protection is enforced by checking each message reference for legality. For better performance, DPF eliminates bounds checks by using aggregation. This optimization works by scanning forward from each shift instruction to find the load with the largest offset. Before the shift is performed, we check that this offset is valid. If it is, then execution continues, and no more bound checks are performed until the next SHIFT instruction. If the check fails, then the filters along this path are considered to have rejected and are skipped. An extension to this optimization is to simply scan forward until either a shift using an indirect load is found, and only then recalculate the bounds check. This optimization aggregates all bounds checks to a single point in time and has the pleasant side effect of being a cheap way of eliminating packets.

3.3 Results of optimizations

Figure 2 demonstrates the net effect of these optimizations by showing the code DPF generates for the first three atoms of the filter described in Figure 1. As can be seen, DPF generates efficient code: all alignment checks have been eliminated, all filter constants have been encoded in the instruction stream, all delay slots have been filled, and the optimal number of registers (2) has been used. Furthermore, we have been able to exploit the fact that the minimum message size is 64 bytes: since the filter only references the first 40 bytes of the message, all bounds checks have been eliminated.

An additional benefit of dynamic code generation is that the fixed cost of starting the packet-filter engine itself is small. In the example we look at, only two registers are needed. As a result, the latency of invoking the packet filter from an interrupt handler (in a manner similar to some hand-crafted message demultiplexors such as Thekkath et al. [23]) is small, since the handler must only save and restore a handful of registers to classify a packet. In contrast, an interpretive packet-filter engine is a more heavy-weight entity that has large register requirements; these requirements cause a commensurate increase in register saves and restores, increasing its latency.

```
# Check Ethernet header
# (12:16 == 0x8) &&
lhu   t1, 12(a0)     # 12:16 (a0 holds pointer to message)
li    t0, 0x8        # Load 0x8
bne   t1,t0,$next    # Branch to next filter if not equal

# SHIFT(6 + 6 + 2) && (is optimized away)

# Check protocol: TCP is 6
# (9:8 == 6) &&
lbu   t1,9+14(a0)    # 9:8 after propogating ethernet header size
li    t0,6           # load 6
bne   t1,t0,$next    # Branch to next filter if not equal
```

Figure 2: Generated MIPS R3000 code to recognize the first three atoms of an Ethernet/IP/TCP header (after optimization).

3.4 Concerns

The main concern arising from the use of dynamic code generation is portability. Fortunately, compiler technology has reached the threshold where portable dynamic code generation systems are beginning to be made freely available. For instance, DPF uses the public-domain VCODE dynamic code generation system [8]. Language support for dynamic code generation has even been added to high-level languages such as C [4, 9] making the generation code code at runtime no more complex than the implementation of statically generated code. However, even if an implementor eschews the use of a portable system, writing the machine-specific pieces of DPF is not difficult, since the amount of machine-specific code that must be written is small: packet filters use a simple, restricted set of operations (e.g., all operations are unsigned, freeing implementors from supporting floating-point and signed integer operation). Our initial implementation of DPF took such an approach. Not including the instruction macros themselves (which are freely available in systems such as the New Jersey Toolkit [20]), it was a few hundred lines of machine-dependent source. Compared to the overall effort required in other parts of the kernel, this is a rather small amount of code to write and, in our experience, can take as little as one to two days to construct. Finally, the use of dynamic code generation can be a simplifying mechanism, since it removes the need for a packet-filter expression evaluator.

4 Performance

In this section we evaluate DPF's performance. Our experiments are based on those of PATHFINDER [1]. The PATHFINDER and MPF numbers were taken from the literature [1] (we were able to independently verify those for MPF). To ensure meaningful comparisons between the systems, we ran our experiments on the same hardware (a DECstation 5000/200) and in user space. One million trials were run per experiment; the total time was then divided to give the time for a single run. Time was measured using the `getrusage` system call.

Disjunction overhead The time to perform a disjunction varies as the number of dependent filters increases. In Table 2 we vary the number of filters, keeping them identical in all but a single disjunctive atom (the destination port). As the number of dependent

Header type	1 filter	2 filters	>= 3 filters
Fixed	1.14	1.23	1.39
Variable	1.27	1.35	1.51

Table 2: Time to recognize both fixed and variable sized TCP/IP headers; times are in microseconds.

Filter	Unexpected	Expected
MPF	71	35
PATHFINDER	39	19
DPF	1.5	1.5

Table 3: Time to recognize 10 fixed-sized TCP/IP headers; times are in microseconds.

filters increases, more comparisons must be performed. When this number becomes greater than three, we generate a simple hashing function to efficiently match disjunctions. The difference between the time to recognize a single filter and the time to recognize a given filter using hash tables is almost 20%, showing that the optimizations DPF performs based on the number of hashed keys are worthwhile.

Classification overhead Figure 3 presents the time to classify packets destined for one of ten TCP/IP filters for each of the three systems. The numbers for MPF and PATHFINDER were taken from [1]. To ensure a meaningful comparision, we have been careful to use equivalent filters.

Because interpretation is expensive, both MPF and PATHFINDER maintain a cache of recently recognized filters; on message arrival, this cache is checked first. Given DPF's filter classification speed, our current implementation does not use caching. As can be seen, DPF is 25–50 times faster than MPF and 13–26 times faster than PATHFINDER, depending on whether or not the filter was cached ("expected"). Clearly, dynamic code generation is profitable.

Scalability This experiment measures the recognition overhead of each additional level in the protocol stack. Each level is represented by two simple atoms, so that a depth of two implies four atoms, a depth of three six atoms, etc. This experiment was performed using a single active filter. As Figure 3 shows, the DPF implementation is approximately 35–40 times faster than PATHFINDER, again because of the use of dynamic code generation.

Insertion overhead The time to insert a new filter depends significantly on the number of nodes it contains. In our current system, the "worst case" overhead is when a filter matches all nodes but the last one, since all nodes in a filter must be checked for equality against all others. The cost of this operation for the TCP/IP filter used in our classification experiment is 220 microseconds. This is approximately a factor of three slower than PATHFINDER. The bulk of this difference is not caused by the overhead of dynamic code generation, which accounts for less than 40% of insertion cost. Rather, it is due to the overhead of translating the filter representation of client filters into the internal representation used by DPF (30%) and by the simple algorithms we use to merge filters into the existing filter structure (greater than 30%). Both of these overheads are amenable to reduction through the use of more sophisticated algorithms. While we intend to improve these times, we believe that the current overhead is an acceptable tradeoff for an order-of-magnitude performance improvement in packet recognition, since it is repaid with just a handful of packet classifications.

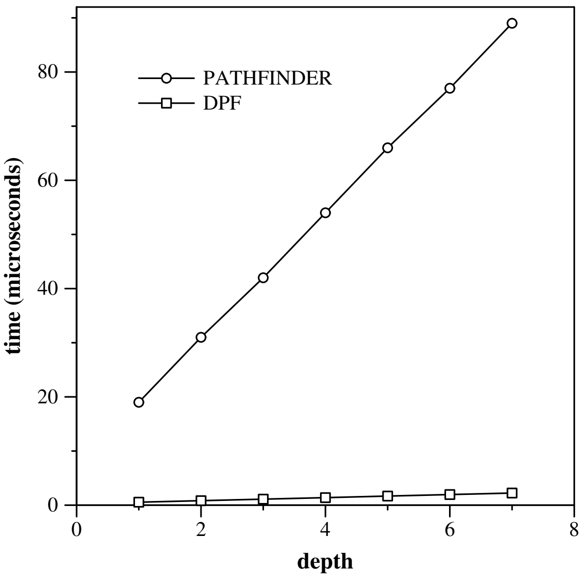

Figure 3: Time to recognize message versus depth of protocol stack; times are in microseconds.

Comparison to hand coded Clark et al. [3] give an instruction count of 57 instructions for an optimized hand-coded IP demultiplexor. Without disjunctions, our implementation requires approximately 18 instructions. The main reason for this is that DPF can aggressively exploit runtime constants. For example, by incorporating constants into the instruction stream, it eliminates table lookups to check allowed values. Note that DPF generated code is as fast as hand-crafted demultiplexors even when it cannot use runtime information: given the restricted domain in which filters are written, DPF can perform the same optimizations that current compilers do for packet filter expressions.

5 Using DPF

We have integrated DPF with the Aegis exokernel [10]. Although our implementation is for an exokernel, DPF is mostly independent of the operating system. For example, adding DPF to a standard run-of-the-mill UNIX kernel should be relatively straightforward.

DPF allows Aegis to export the Ethernet interface securely and efficiently to applications. Without a packet filter, the kernel would need to query every application or network server on every packet reception to determine who the packet was for. Packet filters can be viewed as application code that is downloaded into the kernel. Protocol knowledge is limited to the application, while the protection checks required to determine packet ownership are couched in a language understood by the kernel. Fault isolation of the downloaded code is ensured by careful language design (to bound run time) and runtime checks (to protect against wild memory references and unsafe operations).

By separating protection (determining who the packet is for) from authorization and management (setting up connections, sessions, managing retransmissions, etc.), fast network multiplexing is possible while still supporting complete application-level flexibility. This allows protocols to be implemented as user-level libraries, which can be tailored to application needs. As pointed out by many others, such user-level protocols have many benefits [6, 7, 15, 21, 25].

The design of the DPF was heavily influenced by the pervasive reliance Aegis places on application use of library operating sys-

Protocol	Latency	Throughput
UDP	309	1.03
TCP	412	1.04

Table 4: Latency and throughput for UDP and TCP over Ethernet. Latency in microseconds. Throughput in Mbyte/s.

tems. In particular, is not uncommon for many differently-authored protocol implementations to be active on the same Aegis system. The concrete effect of this situation is that there can be many differently structured packet filter specifications for each protocol family. The challenge such diversity places on a packet filter system is that if proper care is not taken in language design, it is very difficult to merge these differently written packet filters, even though their functionality is equivalent. To counter this difficulty we designed the DPF language so that functionally equivalent but structurally different packet filters can be put in a canonical form so that these filters can be merged with other packet filters in the same protocol family. The key to this process is the use of a declarative language instead of an imperative one.

The one problem with the use of a packet filter is ensuring that that a filter does not "lie" and accept packets destined for another process. Simple security precautions such as only allowing a trusted server to install filters can be used to address this problem. On a system that assumes no malicious processes, the DPF language is simple enough that in many cases even the use of a trusted server can be avoided by statically checking a new filter to ensure that it cannot accept packets belonging to another; by avoiding the use of any central authority, extensibility is increased.

Using DPF we have implemented several protocols at user-level, including UDP and TCP. We use these implementations in all our applications to interact with the rest of our computing environment (e.g., with our NFS server to retrieve and store files). The UDP library uses one filter per connection. The TCP library uses one filter for listening for a connection and one filter per established connection. Both UDP and TCP are straight implementations of RFC 768 and RFC 793, respectively. The TCP implementation is not fully TCP compliant (it lacks support for fluent internetworking such as fast retransmit, fast recovery, and good buffering strategies), but it is complete enough to communicate correctly and efficiently with other TCP implementations in other operating systems.

Table 4 lists the latency and throughput of the user-level UDP and TCP libraries that use DPF. The experiments were performed on a pair of 40-Mhz DECstation 5000/200s and measured using a cycle counter. Latency was measured by sending a 60-byte message with a 4-byte payload and waiting for a reply. Throughput for UDP was measured by sending a train of six packets of 1,500 bytes and waiting for a quick response. Throughput for TCP was measured by writing 10 Mbytes in 8-Kbyte chunks over a TCP connection. The maximum segment size was 1,500 bytes and the window size was set to 8 Kbyte. Larger window size increases the throughput. Both the UDP and TCP libraries checksum the pseudo-header, IP header, UDP header, and user data.

The performance of the user-level implementations of UDP and TCP using DPF are as good as hard-coded in-kernel implementations and better than other user-level implementations of the internet protocols in user space over Ethernet [25, 15]. In fact, our user-level implementations perform close to the possible limits; Thekkath and Levy measure 340 microseconds between 25-MHz DECstation 5000/200s for a user-level reliable raw RPC protocol [22], while we are measuring 309 microseconds on a faster machine (40-MHz) but using IP and UDP with checksumming. Part of the good performance is due to Aegis's efficiency [10], but from these numbers it is clear that DPF does not form a bottleneck for achieving high performance communication while maintaining a high degree of flexibility.

6 Related work

There have been four packet filter systems described in the literature [1, 17, 18, 27]. All of these systems use interpretation; additionally, PATHFINDER considers the effect of hardware assistance [1]. To the best of our knowledge, previous systems have not used the optimizations we describe in this paper.

In the context of language design, DPF shares many similarities with PATHFINDER. While DPF was developed independently of PATHFINDER, its declarative style is similar to PATHFINDER's (e.g., the use of predicates and simple boolean atoms). The main difference is generality. DPF appears to include a richer set of operations than PATHFINDER, which restricts operations in order to allow an easier encoding in hardware; since we compile to machine code, we are unfettered by this constraint. One cost of a richer language is that there are many structurally different representations of functionally equivalent atoms (e.g., (4 + 0:16) is equivalent to (0:16 + 4)). The presence of such structural artifacts can work against inter-filter optimizations that rely on detecting similarities. In order to counter such artifacts, we have developed a set of tree rewriting rules that are applied to each atom to yield a single canonical form.

This is the first paper to systematically use and investigate the effects of dynamic code generation on packet filter performance. The idea of using dynamic code generation to improve packet filter performance is an old one: Mogul et al. [18] mentioned dynamic code generation as a possibility for improving packet filter performance, but considered it "too complicated." Thekkath et al. [24] mentioned the use of dynamic code generation for packet filters as well, but did not implement it. [1] Recently, Minshall and collegues have experimented with the use of dynamic code generation in BPF but have not reported on any results. [2] The system we developed was first mentioned in Engler and Proebsting as a motivation for dynamic code generation [11]. We hope that the performance improvements we demonstrate are a vindication of the packet filter model for efficient demultiplexing, allowing packet filters to be as useful in the context of Gb/sec networks as they are on Mb/sec Ethernets.

The general use of dynamic code generation to speed up language execution has a venerable tradition. Deutsch used it to implement Smalltalk [5]; it was later used in Self [2]. We were also influenced by Massalin's use of dynamic code generation in the context of operating system calls [16].

Recent work has concentrated on improving language-level support for dynamic code generation. Leone et al. [14] describe language support for dynamic code generation in the context of a restricted functional language. VCODE [8] and 'C [9] provide a portable, efficient means of generating machine code "on the fly" (both systems are available publicly). Consel and Noël [4] describe a technique for specializing programs with respect to run-time invariants.

7 Conclusion

We have presented a packet filter system, DPF, that uses dynamic code generation to improve the performance of its packet-classifier engine by 13–26 times that of the best numbers presented in the literature. Since the code DPF generates can exploit filter constants that are not available to static implementations, it can even be faster than hand-crafted demultiplexing routines. Its performance shows that packet filters can work well even in the context of Gb/sec networks;

[1] Personal communication with Chandu Thekkath.

[2] Personal communication with Greg Minshall.

the simplicity of the system demonstrates that this performance can be achieved with modest effort.

Acknowledgments

We thank Héctor M. Briceño and Dominic Sartorio for their help with the development of DPF. We also thank Eddie Kohler and Anthony Joseph for carefully proofreading this paper.

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] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of PLDI '89*, pages 146–160, Portland, OR, June 1989.

[3] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[4] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, St. Petersburg, FL, January 1996.

[5] P. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of 11th POPL*, pages 297–302, Salt Lake City, UT, January 1984.

[6] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1994*, pages 2–13, London, UK, August 1994.

[7] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Clamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1994*, pages 14–24, London, UK, August 1994.

[8] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996. `http://www.pdos.lcs.mit.edu/~engler/vcode.html`.

[9] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 22th Annual Symposium on Principles of Programming Languages*, 1995.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[11] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Proceedings of ASPLOS-VI*, pages 263–272, October 1994.

[12] N.C. Hutchinson and L.L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Trans. on Soft. Eng.*, 17(1), January 1991.

[13] D. B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. Technical Report TR91-151, Rice University, March 1991.

[14] M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, Copenhagen, Denmark, June 1994.

[15] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, 1993.

[16] H. Massalin. *Synthesis: an efficient implementation of fundamental operating system services*. PhD thesis, Columbia University, 1992.

[17] 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.

[18] 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.

[19] T. A. Proebsting and C. N. Fischer. Linear-time optimal code scheduling for delayed-load architectures. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[20] N. Ramsey and M. F. Fernandez. The New Jersey machine-code toolkit. In *1995 Winter USENIX*, December 1995.

[21] V. Buch T. von Eicken, A. Basu and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[22] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[23] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating data and control transfer in distributed operating systems. In *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 2–11, October 1994.

[24] C.A. Thekkath, H.M. Levy, and E.D. Lazowska. Efficient support for multicomputing on ATM networks. Technical Report TR93-04-03, University of Washington, April 1993.

[25] C.A. Thekkath, T.D. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1993*, pages 64–73, San Francisco, California, October 1993.

[26] T. von Eicken, A. Basu, and V. Buch. Low-latency communication over ATM networks using active messages. *IEEE Micro*, pages 46–53, February 1995.

[27] 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.