Appears in "Proceedings of the First Symposium on Operating Systems Design and Implementation," Usenix Association, November 1994.

PATHFINDER: A Pattern-Based Packet Classifier

Mary L. Bailey   Burra Gopal   Michael A. Pagels   Larry L. Peterson
Prasenjit Sarkar*

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

This paper describes a pattern-based approach to building packet classifiers. One novelty of the approach is that it can be implemented efficiently in both software and hardware. A performance study shows that the software implementation is about twice as fast as existing mechanisms, and that the hardware implementation is currently able to keep up with OC-12 (622Mbps) network links and is likely to operate at gigabit speeds in the near future.

1 Introduction

A packet classifier is a mechanism that inspects incoming network packets, and based on the values found in select header fields, determines how each is to be processed. A classifier can be thought of as assigning a tag (type) to each packet, or alternatively, identifying the *flow* or *path* to which the packet belongs. We call this mechanism a packet classifier rather than a packet filter because it more accurately describes the function being performed—it classifies all packets, rather than filtering out select packets. Packet classifier is also becoming the accepted name in the networking community.

An operating system can take advantage of a classified packet in several ways. First, it can use this information to determine what code should be invoked to process the packet. For example, it might select a user-level protocol stack [5], or it might call a procedure that has been optimized to handle packets that follow a particular path through the system [8, 1]. Second, the OS can use the classification information to acquire the resources needed to process the packet. For example, it might select a buffer into which a packet should be placed [3], or it might determine the priority at which the packet should be processed.

The OS programs a classifier by giving it a set of *specifications*, each of which is bound to some identifier. When the classifier matches an incoming packet to a particular specification, it returns the corresponding classification identifier. The specification can either be given by a block of imperative code, in which case the classifier can be viewed as an interpretor (this is the case with three well-known systems, MPF [9], BPF [6] and CSPF [7]), or the specification can be declarative, that is, given by a pattern to be matched (this is the case for the classifier presented in this paper).

This paper describes a new packet classifier, called PATHFINDER, which maps incoming packets into an identifier for the *path* that the packet will traverse through the system. For the purpose of this paper, a path is an abstraction that encapsulates the code block and all the resources required to process the packet [8]. This paper concentrates on how PATHFINDER determines the path to which a packet belongs, and not on how the OS implements and exploits paths.

PATHFINDER has many of the same goals as existing packet classifiers, particularly MPF: it works for a wide range of protocols, it correctly handles fragmentation, it is able to process variable length headers, and it is efficiently implemented. In addition, PATHFINDER has two design goals that set it apart from existing packet classifiers. First, it is designed to support the composition of multiple independently specified patterns. The idea is that each protocol generates a protocol-specific pattern, and that by traversing the protocol graph at the time a network connection is opened and collecting a pattern from each protocol, a compound pattern that classifies packets for that entire path can be constructed. Second, PATHFINDER is designed so it can be efficiently implemented in both software and hardware. The importance of this goal is twofold: (1) it is important to classify a packet as early as possible so no resources are unnecessarily wasted, and sometimes this means running the classifier on the network adaptor; and (2) on high-speed networks, it might be necessary to implement the classifier in hardware in order to keep pace with network speeds.

This paper analyzes the merits of both implementations in terms of performance and functionality. The software implementation is twice as fast as any existing packet classifier, and is probably fast enough for all but the most demanding network loads. The only complication is whether or not

*This work supported in part by National Science Foundation Grant NCR-9204393 and ARPA Contract DABT63-91-C-0030. Additional information about this work, including software distribution, can be found at http://www.cs.arizona.edu/xkernel/www.

the host can make the classification decision soon enough to yield the maximum benefit. The hardware implementation is currently able to keep up with OC-12 (622Mbps) network links, and is likely to operate at gigabit speeds in the near future. A combination of both hardware and software implementations appears optimal, with the hardware acting as a cache for the software.

This paper is organized as follows. Section 2 gives an overview of PATHFINDER's design and Section 3 sketches how PATHFINDER has been implemented in both software and hardware. Section 4 then reports on the performance of these two implementations. Finally, Section 5 discusses several issues raised by this work, and Section 6 offers some concluding remarks.

2 Design Overview

This section describes PATHFINDER at an abstract level, focusing on how a specification is defined, and how PATHFINDER uses this specification to classify incoming packets. We start with PATHFINDER's most basic elements, and then incrementally add the machinery needed to handle more complicated cases.

```
# src port (2 bytes)
tcp_cell1 = (0, 2, 0xffff, 1234)
# dest port (2 bytes)
tcp_cell2 = (2, 2, 0xffff, 4321)
tcp_line = (tcp_cell1, tcp_cell2)
# 20 byte fixed size hdr
tcp_header = (FIXED, 20, 1)
tcp_pattern = (tcp_header, tcp_line, NULL)
```

Figure 1: TCP Example

```
# ip src addr 192.12.69.1 (4 bytes)
ip_cell1 = (12, 4, 0xffffffff, 0xc00c4501)
# IP more fragments bit (1 byte)
ip_cell2 = (6, 1, 0x20, 0x20)
# IP msg id (2 bytes)
ip_cell3 = (4, 2, 0xffff, NULL)
# TCP's protl id (1 byte)
ip_cell4 = (9, 1, 0xff, 6)
ip_line1 = (ip_cell1, ip_cell2, ip_cell4)
ip_line2 = (ip_cell1, ip_cell3)
# 20 byte fixed size hdr
ip_header = (FIXED, 20, 1)
# ip pattern for fragmented packets
ip_pattern = (ip_header, ip_line1, ip_line2)
```

Figure 2: IP Example

2.1 Basic Mechanism

The underlying primitive element in PATHFINDER is the *cell*. It is given by the tuple $\langle$*offset, length, mask, value*$\rangle$. Two operations are permitted on a cell: it can be compared to a packet, or it can be loaded from a packet. In the former case, *length* bytes at *offset* bytes from some reference point in the packet (more on this below) are masked out and compared to *value*. The cell is said to match if the masked packet contents equal the specified value. That is, the cell extracts some information from a packet and compares it with a known value stored in it. In the latter case, *length* bytes at the specified *offset* are masked and loaded into *value*. That is, the information extracted by a cell can be used to generate a new value to store in it.

A *line* corresponds to a set of cells. If all the cells in a line match a packet, then the line is said to match the packet. A line is said to be loaded when one or more of the cells that make up the line is loaded. Intuitively, a line attempts to match (is loaded from) a single protocol header; each cell matches (is loaded from) a different field in that header.

In the simplest case, a protocol specifies a *pattern* that it wants to match by giving a $\langle$*hdrlen,line*$\rangle$ pair, where *hdrlen* specifies the number of bytes in this protocol's header. This value is used to determine where the next protocol's header begins, that is, the sum of the *hdrlen*'s of all preceding protocols, plus a cell's *offset*, determines the absolute offset from the beginning of the packet for that cell. The pattern specified by a protocol is said to match when a packet that contains the header of the protocol matches the line in the pattern.

For example, Figure 1 gives a PATHFINDER specification `tcp_pattern` for the TCP protocol. It contains a header `tcp_header` which specifies that the TCP header is 20 bytes long (ignore the exact format for now), and a line `tcp_line` which contains cells that match packets meant for a particular TCP connection between source and destination ports 1234 and 4321. (Ignore the third component of `tcp_pattern` for now.) For simplicity, `tcp_pattern` is defined in several steps; it could also be given in a single, nested statement.

The full specification (*composite pattern*) that classifies packets on a particular path is constructed from the individual specifications (*patterns*) of each protocol that lie on that path. PATHFINDER associates each composite pattern with an identifier unique to that path. When PATHFINDER classifies a packet, it associates the packet with the identifier bound to the longest matching composite pattern.

In the x-kernel, for example, an `EstablishPath` operation is used to set up a path. This operation calls the sequence of protocols starting with the topmost protocol and ending at the device driver, and at each protocol in this chain, concatenates that protocol's pattern to the composite pattern. The device driver protocol then programs PATHFINDER with the resulting compound pattern, bound to an identifier for the path just traversed.

2.2 Fragmentation

To handle fragmentation, PATHFINDER actually associates a pair of lines with each pattern. (This corresponds to the third component of `tcp_pattern` in Figure 1.) This second line, which we sometimes call the *dual line*, is considered an *attribute* of the first (*primal*) line—it is an inactive template that contains cells to be loaded when the first line matches. The idea is that when the first line matches the first fragment of a multi-fragment sequence, some cells of the dual line are loaded with values found in that first fragment (e.g., a message id field), and this newly generated line becomes active and matches the subsequent fragments.

For example, Figure 2 gives a pattern specification for the IP protocol. To identify the individual fragments of a multi-fragment message, IP includes a unique message identifier *msgid* in the IP header of each fragment. The *morefrags* bit in the header of the first fragment indicates whether there are any other fragments in that message. In `ip_pattern`, `ip_line1` filters packets coming from the IP host 192.12.69.1. The dual line `ip_line2` contains cells to extract the IP source address and msgid from the first fragment of a multi-fragment message. Since no value is specified in `ip_cell3`, is is loaded from the initial fragment. Note that trailing fragments do not have the headers of the protocols above IP. Hence, without the dual line, PATHFINDER cannot associate them with the identifier bound to the longest matching composite pattern. However, when these fragments match the dual line, they can be correctly classified as falling on the same path as the first fragment.

Note that some of the cells that make up the first line are also present in the dual line. Also notice that in this IP example, the specified pattern expects only fragmented packets. In fact, both fragmented and non-fragmented packets can arrive. This is handled by having IP contribute two patterns to the composite pattern—one that matches unfragmented packets and one that matches fragmented packets—either of which might contribute to the IP portion of the composite pattern matching an incoming packet. In other words, each protocol submits one or more patterns that specify all the types of packets that they accept, where each pattern is defined by two lines.

2.3 Out-of-Order Delivery

If fragments of a message are delivered out-of-order, PATHFINDER might miss the longest matching composite pattern for later fragments that arrive before the first one. Usually, only the first fragment has the information that can trigger dual lines that correctly classify subsequent fragments. Hence, if the later fragments arrive before the first fragment, there must be a facility to *postpone* the classification of these fragments until the first fragment arrives. PATHFINDER handles this by associating a *postpone* attribute with each cell—if this attribute is set, then the classification of packets that do not match the cell is postponed until a packet that matches the cell arrives.

For example, IP might submit a pattern that sets the postpone attribute for the cell that checks for the *morefrags* bits of the IP header; i.e., `ip_cell2` in Figure 2.

2.4 Variable Length Headers

The last complication is that not all protocols have fixed headers, as suggested in the preceding two examples. To handle variable length headers, the *hdrlen* field can be given by a cell that is loaded from the message. Figure 3 extends the example specification for IP to include variable length headers (the TCP example would be extended similarly). Here, the length of the header is obtained from the `ip_hdrcell` cell in the packet header. In this case, the length is specified in words (4 bytes). Thus, we multiply this length by 4 (the third field in `ip_hdrcell`) to obtain the length in bytes.

```
# get lower 4 bits from offset 0
ip_hdrcell = (0, 1, 0x0f, NULL)
ip_header = (VARIABLE, ip_hdrcell, 4)
```

Figure 3: IP Example with Variable Length Header

2.5 Additional Features

In addition to the standard features just described, PATHFINDER also includes more advanced features that might prove useful in certain settings. For example, it includes the ability to handle range comparisons, handle multiple SIPP or XTP-style headers in a single protocol, and peek into the headers of other protocols. It is also possible to add more power to PATHFINDER by allowing a cell to perform more sophisticated operations than simple boolean comparisons.

3 Implementation

This section briefly sketches how PATHFINDER has been implemented in both software and hardware. Notice that the hardware implementation, by its very nature, must be viewed as a cache—finite space will limit the hardware to being programmed to match only a fixed number of patterns at a time. Given this limitation, we also considered the question of what other restrictions we might place on the hardware in an effort to improve its performance. The software version of PATHFINDER implements the full specification described in Section 2.

3.1 Software

The software version of PATHFINDER is implemented as a directed acyclic graph (DAG). Each node in the DAG corresponds to a cell. There are also special nodes, called *header nodes*, that act as demarcation points between the list of cells—i.e., a line—that make up patterns for different protocols. These header nodes contain information about header lengths, so absolute offsets can be computed correctly.

Translating the specification of a pattern to its DAG representation is straightforward: (1) a header node is created for the header-tuple, (2) a list of cell nodes is created for the primal line and then appended to the header node, and (3) a list of cells is created for the dual line and then added as an attribute of the *last cell* of the primal line. Note that until the dual line is activated by a match of all the cells of the primal line, it does not exist in the DAG; it is only an attribute of some cell in the DAG.

The DAG edges that link the header node and the list of cells corresponding to a single line are called *AND* edges. Figure 4 illustrates the set of cells that make up a TCP line, and the corresponding header node. It has only *AND* edges.

The DAG supports a second kind edge, called the *OR* edge, that allows the multiple lines that correspond to a single protocol to be linked together; either the first line or the second line yields the same path identifier. When a new pattern is inserted into the DAG, either by the OS or because the dual line in a protocol's pattern is activated, the DAG is traversed to determine the longest current prefix match. Only the remainder of the cells is added to the DAG by using the *OR* edge to separate this suffix from the cell that immediately follows the prefix. In this way, PATHFINDER avoids redundant computations that would have to be done if it tried to match a packet against multiple copies of the same cell. Note that if the longest prefix match is not identified, then the DAG degenerates into a linear list of patterns that must be checked one after the other. This would result in behavior similar to BPF.

Figure 5 shows a pair of lines that represent IP; *OR* edges are denoted by dashed lines. In this example, the bottom line, which is followed when the *morefrags* cell does not match, corresponds to a second pattern submitted by IP to capture non-fragmented packets. The $\langle$`line2`$\rangle$ associated with the *protid* cell is an attribute of the cell that represents the dual line for that pattern. A match on the first line triggers the dual line, which is then activated and added to the DAG.

When adding additional cells, attention is also paid to whether there currently exists a cell with exactly the same offset/length/mask triple, differing only in the value field. When this occurs, the two cells are merged into one, and implemented as a hash table that maps values into edges.

When classifying an incoming packet, the packet is matched against the cells in the nodes of the DAG until the longest matching line (*AND*-linked set of cells) is found. The corresponding path identifier is then returned. The DAG is constructed in such a way that the longest matching line can be obtained by simply traversing the *AND*-edges whenever there is a match, and the *OR*-edges whenever there is no match, until a leaf node is reached.

If a packet does not match a cell and the cell has the postpone attribute set, PATHFINDER postpones the classification of that packet until a packet that matches the cell actually arrives (or a timer expires). The postponed classification is then attempted once more (or the packet is discarded). As explained earlier, this is sufficient to correctly classify a packet whose fragments are delivered out-of-order.

3.2 Hardware

The hardware prototype is implemented using an Altera 881152GC192 FPGA. The design was specified using Synario ABEL-HDL and verified using the Silos Simulator. It was then fit onto the device using Altera's MaxPlus-II device kit. Altera FPGA's were chosen because of their speed; other FPGA's can be used by simply using the appropriate device kit in the Synario environment. FPGA's were chosen for the prototype to facilitate quick evaluation of the design. A production version would likely be implemented using custom VLSI to increase both the speed and size of the circuit.

Each PATHFINDER feature was evaluated for its impact on the speed and utility of the prototype. Moreover, there were tradeoffs between the complexity of each cell, the number of cells per line, and the number of lines. The resulting prototype implements the basic cell functionality given by the tuple $\langle$*offset, length, mask, value*$\rangle$, and composes multiple cells to form a line. For speed, the prototype processes a 16-bit input stream rather than a byte stream. Because fragmentation is common and has reasonable complexity, the hardware prototype supports the handling of fragmentation. Other complexities are not handled due to their impact on the design, as discussed below.

Each cell contains an offset/mask/value triple; the length is fixed at 16 bits, the size of the input stream. As in the basic PATHFINDER mechanism, the offset, mask and value are used to compare the input stream with the stored data in the cell. The cell matches if the masked packet content equals the specified value at the correct offset in the packet. We considered having the value specified by ranges instead of matching on a single value. This would involve using two comparators in each cell, one for each of the upper and lower bound of the range, instead of the simple equality check. Due to the limited space in the Altera FPGA, we chose to implement more simple cells rather than fewer complex cells. Thus each cell matches a single value.

As stated above, the hardware prototype processes a 16-bit stream rather than a byte stream. This decision was made for performance considerations, and will be discussed in more detail in Section 4. It does place two restrictions on the classifier: (1) all cells have a fixed length of two bytes, and (2) all cells must start on even byte boundaries. These two restrictions only affect the way patterns are specified, and do not restrict the generality of the classifier, as the *mask*

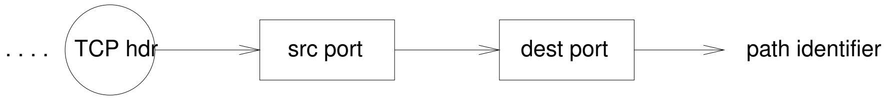

Figure 4: Simple Line

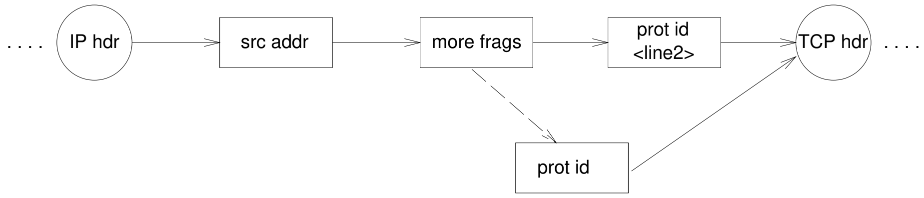

Figure 5: Pair of Lines

can always be used to ignore parts of the cell that are not of interest. It may mean that additional cells are required to handle the case where the two bytes of interest begin at an odd byte offset. Here, two cells are now required, instead of one. The 16-bit design does not preclude packets whose length in bytes are odd—we assume that network adaptors can pad such packets to make their length an even number of bytes.

To handle fragmentation, we initially planned to associate two complete lines for each pattern, one for the primal line and one for the dual line. However, we found that in most cases, there were some number of identical cells in the two lines, and thus, we saved space by using those cells in both lines, rather than replicating them. As a result, there are three types of cells: (1) those used to match both the primal and dual lines, (2) those used in just the primal line, and (3) those used in just the dual line.

The prototype does not handle out-of-order fragments using the POSTPONE facility, as this would require buffering on the network adaptor as packet processing cannot be done immediately. The synchronization logic for the POSTPONE facility would considerably slow the clock rate of the prototype, since it would require a relatively complex interface with the memory on the network adaptor board. In general, we did not feel that out-of-order packets are common enough to justify the increased complexity required in the hardware.

The prototype also *cannot* handle variable length headers; i.e., all headers are assumed to be fixed. This allows the byte offsets of all patterns for a single packet type to be determined when the patterns are loaded into the chip, which vastly simplifies the hardware design and allows for very high clock rates in the prototype. Once again, variable headers are not common enough to justify the necessary additional hardware.

The final restriction of the FPGA implementation is the fixed and limited amount of space available for each pattern. In the hardware, the patterns for all protocols in a given packet are combined to create a single (larger) pattern. The size of this pattern depends on the number of cells available for each pattern. This limits the number of cells that can be implemented.

In spite of the fact that it handles only a subset of the PATHFINDER specification, the hardware prototype can handle handle the vast majority of network packets, including TCP/IP and RPC packets. In the current implementation, one FPGA contains a single pattern consisting of twelve 16-bit cells. As stated above, a custom VLSI implementation would increase both the number of cells and the resulting speed, and is likely the better candidate for a production implementation. As technology increases the size and speed of FPGA's, more space will be available for use in either adding more cells, more patterns, or more features.

The hardware prototype is designed to be placed on the network adaptor board. The application does not have to explicitly program the hardware; in fact, it need not be aware that the hardware exists. The PATHFINDER software accepts the pattern provided the application and checks whether it violates any of the restrictions imposed by the hardware implementation. After verification, the pattern can be downloaded onto the network adaptor board. The hardware prototype allows a pattern to be loaded while concurrent matching of other patterns take place, in order to maximize utility of the hardware.

When the hardware detects a pattern, it sets an output bit to indicate that a match has been detected. The network adaptor may have additional circuitry to place the packet in a specific buffer, either in host memory or in adaptor memory. If an incoming packet does not match any pattern in the hardware, then it is simply processed by the PATHFINDER software.

4 Performance

4.1 Software

The software version of PATHFINDER is about 5,000 lines of C code. We compiled it on each platform using the native C compiler with full optimization.

4.1.1 Lookup Costs

Table 1 compares the per-packet latency of PATHFINDER and MPF [9] on the DECStation 5000/200 for a TCP/IP protocol stack having ten open TCP/IP connections that wait for packets to arrive. The experiments were done with both cold and warm caches. They model the scenario where a total of 100000 TCP/IP packets that contain just the TCP/IP header with no data are recieved by a host machine. A given packet might be intended for any one of the 10 TCP/IP connections, and there is no way to determine which one it is without invoking a classifier. Hence, 1/100000-th of the time taken by PATHFINDER to classify all these packets exactly measures its per-packet latency. Note that since both PATHFINDER and MPF use hash-tables, these results do not vary as the number of connections increases. We see that PATHFINDER is about twice as fast as MPF with both cold and warm caches.

PATHFINDER performs better than MPF for three reasons. First, the DAG is carefully constructed so that just one path in it needs to be traversed to classify the packet. Second, the operations performed by a cell have simple semantics so that they can be implemented efficiently. Finally, the warm cache is implemented in a unique way: instead of storing the set of most recently matched composite patterns in a separate DAG and matching a packet against this DAG first, whenever a packet matches a composite pattern (either this is the first packet that matched that composite pattern, or the packet did not match the existing cache), a few key (unique/unchanging) fields in it are identified. These keys are then used to create a new *key*-pattern. It is associated with the same path-identifier as the corresponding composite pattern and is inserted into the original DAG in such a way that an incoming packet is matched against the key-pattern *before* it is matched against the composite pattern, i.e., the key-pattern acts like a cache for the composite pattern. Since key-patterns are chosen so that they typically have fewer cells than the composite pattern, the lookup latency of PATHFINDER which depends on the number of cells examined (see below) can also be reduced. Note that this is exactly how PATHFINDER handles fragmentation. In fact, the keys to be extracted can be specified explicitly in the dual line of a pattern even if the corresponding protocol does not handle fragmentation.

Classifier	Cache State: Cold	Cache State: Warm
MPF	71	35
PATHFINDER	39	19

Table 1: PATHFINDER and MPF Per-Packet Latencies (μs)

We next examine how the cost varies with the *depth* of the protocol stack (number of protocols) when its breadth (number of open connections) at each level is kept constant at one. The results are given in Table 2. Here, we introduce a series of dummy protocols, each with patterns that look at two header fields. The experimental conditions were similar to those above. They were again carried out on the DECStation 5000/200, this time with a cold cache only, since the warm cache does not show any variation with depth. We see that the per-cell latency of PATHFINDER on this machine is about 6 μs.

	Stack Depth						
	1	2	3	4	5	6	7
Lookup Latency	19	31	42	54	66	77	89

Table 2: PATHFINDER Lookup Latency (μs) versus Depth of Protocol Stack

Finally, we measured the lookup latency of PATHFINDER on various processors for a TCP/IP protocol stack having ten open connections. Both cold and warm caches were measured in a way similar to the first experiment. The results are given in Table 3.

4.1.2 Setup and Remove Costs

We also measured the time taken by PATHFINDER to setup a pattern for a new protocol and for an existing protocol, and compared them with the equivalent times in MPF. The results are given in Table 4. As can be seen, PATHFINDER takes significantly longer to set up a pattern than does MPF. There are several reasons for this. (These reasons also help to explain why PATHFINDER is faster than MPF in the lookup case.)

Classifier	Protocol State: New	Protocol State: Existing
MPF	33	< 2
PATHFINDER	78	45

Table 4: PATHFINDER and MPF Setup Costs (μs)

First, PATHFINDER has to see whether a prefix of the pattern matches any existing pattern before it can decide what suffix of the pattern must be inserted into the DAG. PATHFINDER also includes code that automatically orders cells in the DAG, so that it can retrieve the identifier corresponding to longest matching composite pattern even when two cells have masks/values that partially overlap.

Second, there is no way to determine ahead of time which part of a new pattern will get merged and which part will form a new branch in the DAG. This is why the time to setup a pattern for an existing protocol is not significantly lower than the corresponding time for a new protocol. This former time somewhat lower, however, since the DAG structure need not be modified when some parts of the new pattern get merged with the existing pattern(s) for that protocol. (If the new pattern is composite, then the patterns corresponding to the lower protocols always get merged into the DAG, but PATHFINDER cannot determine even this a priori.)

	Processor			
Cache State	SparcIPC 25 MHz	Sparc10/51 50 MHz	DS5000/200 25 MHz	DS3000/600 175 MHz
Cold Cache	63	15	39	10
Warm Cache	34	8	19	4

Table 3: PATHFINDER Lookup Latency (μs) on Various Processors

Third, the DAG structure has a common root-node (the header-node of the lowest protoccol) from which all insertions should begin: there is no 'handle' into the DAG where the header-cell of a given protocol begins, and as a consequence, insertions cannot 'start' from some interior point in the DAG. Moreover, a protocol can have multiple header-cells, and to handle that, it is necessary to traverse the DAG starting at the root to determine at which header-cell the insertions should start.

Finally, PATHFINDER has to allocate memory for every protocol-specified cell and make a copy of its contents before inserting it into the DAG. Similarly, it has to free memory after deleting a cell from the DAG.

The time it takes to remove the last pattern of a protocol from the DAG is about the same as the time it takes to setup a pattern for a new protocol. Similarly, the time it takes to remove other patterns of a protocol is about the same as the time it takes to setup a pattern for an existing protocol. Note, however, that setups and removes are not done frequently—lookups are most critical. The software implementation tries to trade setup/remove latencies for lookup latency.

4.1.3 Fragmentation Overhead

The fragmentation overhead of PATHFINDER is simply the time taken to: (1) activate a dual line by creating a new line from the template stored as an attribute of the last cell of the corresponding primal line, and (2) insert the new line into the DAG. Note that these steps automatically create a cache to speedup lookups for later fragments of the message. The fragmentation overhead for the IP protocol is 83 μs. Since step (2) simply sets-up a new pattern for an existing protocol, the overhead of step (1) comes to approximately 38 μs. Since the cost of handling fragmentation is slightly more than twice the time taken for a lookup with a cold cache, the automatic creation of a cache pays off if a message contains more than 3 fragments.

4.1.4 Space Requirements

The space required for the DAG implementation is minimal. On the DECStation 5000/200, each cell-node in the DAG occupies about 32 bytes, a cell-node with a hash-table occupies about 540 bytes, and a header-node occupies 8 to 16 bytes depending on whether the header is of fixed or variable length. Each TCP/IP pattern used in these experiments contains 6 cell-nodes and 2 header-nodes, and therefore occupies a modest 208 bytes.

4.2 Hardware

The hardware prototype was implemented using an Altera 881152GC192 FPGA. This FPGA was sufficiently large to hold twelve 16-bit cells, comprising a single pattern. The hardware prototype was tested for correctness with the following tests:

- Single Packet Testing: The lines were loaded with masks, offsets, and values, and a number of packets were streamed through to ensure that the matching portion of the chip functioned correctly.

- TCP/IP fragments back to back: This test case checks the dual line operation of the hardware prototype. A fragmented TCP packet stream was streamed through the circuit to ensure that fragmentation was implemented properly. The circuit was also tested in non-dual mode for correct behavior when the input stream consisted of unfragmented packets.

- Critical timing test: When two packets arrive back-to-back without delay, the start-in signal of the second packet should clear the state pertaining to the first packet and matching should proceed immediately.

- Concurrent Programming and Matching: The pattern was programmed while a packet was being processed to ensure that the current packet was processed correctly, and that the programming took effect in the following packet.

The implementation was tested at 50 and 100 MHz using simplified delay analysis. The design ran properly at 50 MHz, but did not completely function at 100 MHz. The major cause of failure at 100 MHz was asynchronous inputs. These errors may have been due to the simplified delay model. We are in the process of verifying the design using a more accurate timing model, and expect the circuit to function correctly at 100 MHz. Translating this into per-packet latency, it would take approximately 200ns to process a 40-byte TCP/IP message; i.e., one that contains a header and no data. This speed is sufficient to handle data rates of 1.2 Gbps.

5 Discussion

5.1 Putting Classifier Latency in Perspective

The first question raised by this work is "how fast is fast enough?". The issue is at what latency does the the time it takes to classify incoming packets become a limiting factor on network throughput. Table 5 provides the maximum amount of time available to classify packets in worse-case situations. For example, if a system is receiving back-to-back, minimal length IP packets on a 155 Mbps network, the next message will arrive within 1.25 microseconds. If we ignore the ability to buffer packets—that is, we think of data as streaming through the classifier—then the classifier must be able to keep up with the worse-case arrival rate.

	Network fabric bandwidth (Mbs)				
Transfer item	10	100	155	622	1244
8-bits	800	80	52	13	6.5
16-bits	1600	160	104	26	13
IP header	16000	1600	1040	260	130
TCP/IP header	32000	3200	1080	520	260

Table 5: Worse Case Arrival Periods (ns)

As can be seen, the current hardware implementation running at 50 MHz is sufficient to stream data through at OC-12 speeds (622 Mbps). Table 6 provides processing periods, on a per-byte basis, for various clock rates. It shows, for example, that a 1.2 Gbps network would require a processing period of 6.5ns per byte, which could be met by a 100 MHz implementation.

	Clock frequency (MHz)			
Data unit size	50	66	100	125
8-bits	10	7.5	5	4

Table 6: Hardware Processing Time (ns)

Unlike the hardware version, the software implementation has the luxury of buffering packets. Therefore, we can evaluate it in terms of processing complete packets, rather than just headers, in the allotted time. Even then, one can argue that if the classifier can't keep up with the incoming packets, then the host has little hope of dealing with the workload being generated by the application. However, it is interesting to note that if back-to-back 4KByte packets arrive on a 622 Mbps link, then the classifier has approximately 100μs to classify each packet. 1KByte packets would require 25μs per packet, which is near the limit of the software implementation.

5.2 Host versus Adaptor

Beyond the question of speed, however, there remain others issues pertaining to whether classifier functionality should be implemented on the host or on the network adaptor. Putting the classifier on the host obviously implies the software implementation. This has the advantage of tracking improvements in host processor performance, but leaves the question of whether incoming packets are buffered in the host or on the adaptor. Buffering in the host is sufficient for determining what code should be invoked to process the packet (i.e., it makes user-level protocol stacks possible), but it is too late to affect all decisions as to what resources—in particular, memory buffers—are consumed by the packet. As a consequence, it does not support a mechanism like fbufs [3]. Buffering on the adaptor implies that incoming packets are held in adaptor memory and only header information is copied to host memory so the classifier can inspect it. Once the classifier makes a decision, the packet can be moved into host memory using either DMA or Programmed I/O. This approach has the complication of requiring the management of two disjoint buffer pools. It is also not clear how much adaptor buffering is sufficient.

The alternative is to locate PATHFINDER on the adaptor, where it could either be implemented as software running on a general-purpose processor, or as special-purpose hardware such as that described in Section 3.2. Both have the advantage of being able to classify packets soon enough to make the appropriate resource assignment decisions. A general-purpose processor would require buffering on the adaptor, since as illustrated in Section 5.1, even the fastest processors are not able to keep up with emerging network bandwidths. This approach also suffers from the problem of the adaptor's processor not necessarily keeping pace with the host's processor. The special-purpose hardware streams data through, and so must also keep up with network speeds. Fortunately, the data given in Section 4 suggests that is a reasonable expectation.

One final possibility is to put the classifier on both the host and the adaptor, and to treat the hardware implementation as a cache. We have experimented with this configuration on Sparc workstations connected by the ATOMIC network [2]. In this example, the ATOMIC adaptor has a 3-MIPS on-board processor, which was powerful enough to implement only a very restricted version of the software implementation of PATHFINDER. The adaptor-version was able to match only a very small set of patterns, with the on-host implementation catching those not flagged by the adaptor.

5.3 Future Considerations

ATM is an emerging standard for gigabit speed networks. Each ATM cell has a *virtual circuit identifier* (VCI) that can be used to demultiplex it [4]. One could argue that the availability of VCI's diminishes the need for a PATHFINDER-like classifier.

First, it is not yet settled how "plentiful" VCIs will be, that is, will VCIs be an abundant resource that can be cheaply allocated by the host operating system, or will the network control (and charge for) each VCI used by an end system.

If the latter scenario happens, one might aggregate multiple application-level connections onto a single VCI, and therefore need to employ a packet classifier.

Second, even if one can afford to allocate a a VCI to each connection, it is sometimes desirable to demultiplex incoming messages on an even finer level; e.g., program PATHFINDER to know about individual reply messages or messages of a particular type. Whether PATHFINDER can be programmed quickly enough to accommodate such fine-grain demultiplexing is a subject of further study.

Finally, even if ATM becomes a wide-spread network technology, other technologies, including ethernet, will still exist. In order to design an operating system to take advantage of the early demultiplexing made possible by VCIs—e.g., to support user-level protocols—it will be necessary to correctly and efficiently classify packets arriving on networks that do not support early demultiplexing.

6 Concluding Remarks

PATHFINDER is a pattern-based approach for building packet classifiers that can be implemented both in hardware and software. The software implementation is about twice as fast as any existing packet classifier, but cannot be expected to sustain gigabit network rates. The hardware implementation is currently able to keep up with 622Mbps network speeds, and can be expected to sustain gigabit speeds.

We envision two scenarios in which one or both of the implementations can be effectively employed. In the first, packets are buffered on the network adaptor, packet headers are loaded from the adaptor onto the host, the software implementation of PATHFINDER running on the host processes these headers, and the result of the classification governs how the rest of the packet is processed (e.g., where in host memory it copied to). In this scenario, the adaptor must have enough buffering to absorb bursts of data arriving at network bandwidth, but we assume the software implementation need not sustain network speeds for an arbitrarily long period of time. This is a valid assumption if the application program running on that host also cannot consume data at network speeds for an arbitrary period of time.

In the second scenario, a hardware implementation of PATHFINDER running on the network adaptor acts as a cache of the software implementation running on the host; any packets not classified by the hardware are processed by the software implementation. Because the hardware implementation can keep up with network speeds, this approach has the advantage of not requiring any buffering on the network adaptor. It also has the advantage of being able to sustain network speeds for an arbitrary length of time, but this would only be important if the application program does process the incoming data (e.g., it is simply copied to the framebuffer).

References

[1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, Oct. 1993.

[2] D. Cohen, G. Finn, R. Felderman, and A. DeSchon. ATOMIC: A low-cost, very-high-speed, local communications architecture. In *Proceedings of the 1993 Conference on Parallel Processing*, Aug. 1993.

[3] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, Dec. 1993.

[4] P. Druschel, L. L. Peterson, and B. S. Davie. Experience with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM '94 Symposium*, Aug. 1994.

[5] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Dec. 1993.

[6] S. McCnne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX '93 Winter Conference*, pages 259–269, Jan. 1993.

[7] 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, Nov. 1987.

[8] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Petersonand, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report 94-20, Department of Computer Science, University of Arizona, June 1994.

[9] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Winter 1994 Usenix Conference*, Jan. 1994.