Programming language techniques for modular router configurations

Eddie Kohler, Benjie Chen, M. Frans Kaashoek,
Robert Morris, and Massimiliano Poletto*

Abstract

This paper applies programming language techniques to a high-level system description, both to optimize the system and to prove useful properties about it. The system in question is Click, a modular software router framework [13]. Click routers are built from components called *elements*. Elements are written in C++, but the user creates a configuration using a simple, declarative data flow language. This language is amenable to data flow analysis and other conventional programming language techniques. Applied to a router configuration, these techniques have high-level results—for example, optimizing the router or verifying its high-level properties. This paper describes several programming language techniques that have been useful in practice, including optimization tools that remove virtual function calls from router definitions and remove redundant parts of adjacent routers. We also present performance results for an extensively optimized standards-compliant IP router. On conventional PC hardware, this router can forward up to 456,000 64-byte packets per second.

1 Introduction

Click [13] is a software architecture for building routers from modular components. Click components are called *elements*; they tend to be fine-grained, and perform small, focused tasks like decrementing an IP packet's time-to-live field. Complex router behavior is built by composing elements together. This design makes Click flexible—users can implement arbitrary routing decisions, dropping and scheduling policies, and packet manipulations by rearranging elements. Element source code is written in C++, but a simple, declarative data flow language specifies how a particular router configuration is composed. This language definition is a high-level description of the system's behavior, which makes the system easy to understand and modify.

*Parallel and Distributed Operating Systems Group, Laboratory for Computer Science, MIT, Cambridge, Massachusetts. Email: click@pdos.lcs.mit.edu.

This research was supported by a National Science Foundation (NSF) Young Investigator Award and the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory under agreement number F30602-97-2-0288. In addition, Eddie Kohler was supported by a National Science Foundation Graduate Research Fellowship.

Unfortunately, Click's modularity has consequences for its performance. If a router contains many modules, then packets passing through it must traverse many module boundaries; the cost of this crossing is inexpensive in Click, but it is not free. Furthermore, the division of router functionality into modules may cause redundant computation.

In this paper, we attack the performance problems caused by Click's modularity by leveraging that modularity. In particular, we use programming language techniques, such as data flow analyses, to analyze and optimize router configurations in the Click language. When applied to router configurations, which are high-level system descriptions, these techniques can optimize the entire system and prove properties about it as a whole. For example, we can optimize configurations based on user-specified patterns, create fast versions of particular elements, remove virtual function calls from the entire system, and prove that packet data will be aligned correctly throughout the router. We can also combine definitions from multiple routers on a network into a single configuration file, and perform optimizations on each router based on global properties of the network. These optimizations make Click's IP processing time 39% faster (or 51% faster, if you include a sample optimization enabled by a multiple-router configuration). On a 700 MHz Pentium III, this Click IP router can forward up to 456,000 64-byte packets per second, which appears to be close to the maximum performance allowed by the hardware's I/O bus.

Our main contribution is demonstrating that programming language techniques applied at the level of software components can significantly increase the performance of a networking system. Other contributions include a modular, flexible, and fast IP router built on commodity PC hardware and extensive performance analysis of that router configuration.

In the rest of this paper, we give an overview of Click (Section 2) and the Click language (Section 3), describe our language-level optimizations (Section 4), present performance results (Section 5), discuss related work (Section 6), and conclude (Section 7).

2 Click

This section summarizes a recent article describing the Click system [13]. It may be safely skipped by those already familiar with Click.

Click routers are built from components called *elements*. Elements are modules that process packets; they control every aspect of router packet processing. Router configurations are directed graphs with elements as the vertices. The edges, called *connections*, represent possible paths that packets may travel. Inside a running router, elements are represented as C++ objects and connections are pointers to elements. A packet transfer from one element to the next is implemented with a single virtual function call.

Each element belongs to an *element class* that determines the element's behavior. An element's class specifies which code to execute when the element processes a

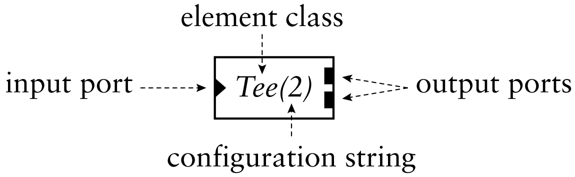


Figure 1: A sample element. Triangular ports are inputs and rectangular ports are outputs.

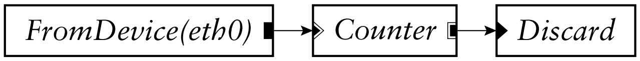


Figure 2: A router configuration that drops all packets.

packet. Each element class corresponds to a subclass of the C++ class `Element`. A do-nothing element class can be written in 9 lines of C++.

Each element also has *input* and *output ports*, which serve as the endpoints for packet transfers. Every connection leads from an output port on one element to an input port on another. Each element has an arbitrary number of each kind of port. Different ports can have different semantics; for example, the second output port is sometimes used for erroneous packets.

Finally, each element has an optional *configuration string* that provides configuration arguments, such as maximum queue lengths or RED parameters. The configuration string is generally used to initialize parts of an element's private state. Figure 1 shows how we diagram an element, and Figure 2 shows a simple router configuration.

Elements tend to be fine-grained: each element performs a simple, well-specified task. Complex functionality is implemented by composing elements together, as this gives the user more flexibility when creating configurations. Figure 3 shows a basic, standards-compliant two-interface IP router configuration. There are 16 elements on its forwarding path. We adapted this IP router configuration for our benchmarks.

Click supports two packet transfer mechanisms, called *push* and *pull*. In push processing, a packet is generated at a source and passed downstream to its destination. In pull processing, the destination element picks one of its input ports and asks that source element to return a packet. The source element returns a packet or a null pointer (indicating that no packet is available). Here, the destination element is in control—the dual of push processing. In a running router, each packet transfer is implemented by a single virtual function call. There are two relevant virtual functions, `push` and `pull`, corresponding to push and pull processing respectively. Figure 4 illustrates how this works. Note that queueing in Click is implemented with an explicit element, called *Queue*.

Click runs as a dynamically loadable module inside a Linux 2.2 kernel. There is also a user-level implementation.

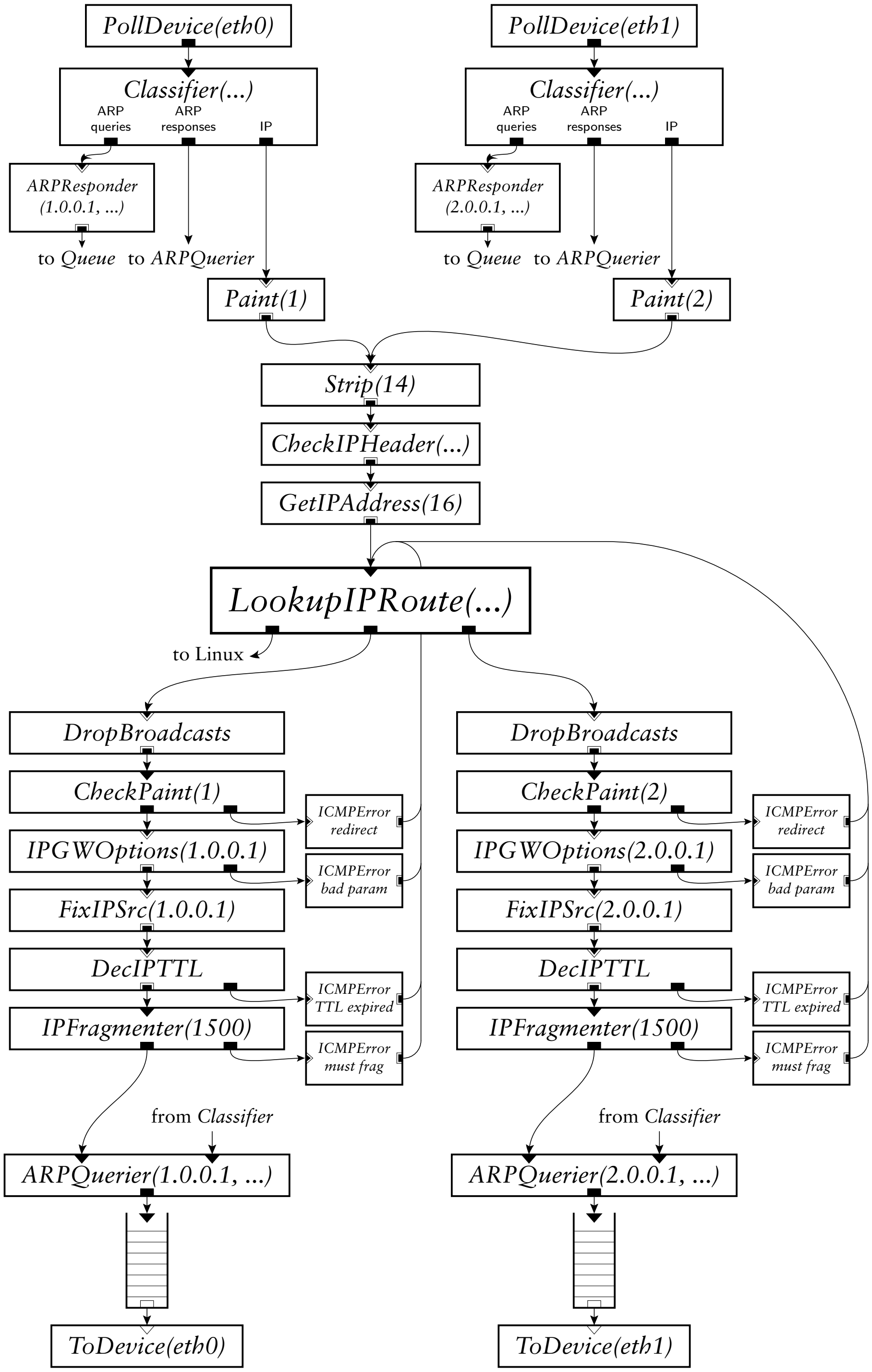


Figure 3: A Click IP router with two network interfaces.

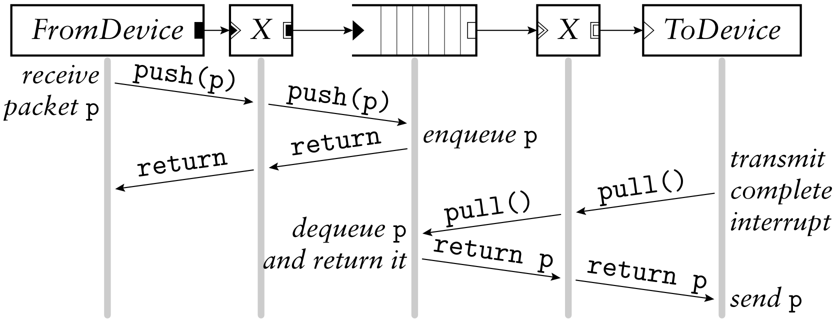


Figure 4: Functions called as a packet moves through a simple router. The central element is a *Queue*. During the push, control flow moves forward through the element graph starting at the receiving interface; during the pull, control flow moves *backward* through the graph, starting at the *transmitting* interface. The packet p always moves forward. The port colors indicate whether a connection is using push or pull processing: black means push and white means pull.

3 Language

Click is programmed using a simple, declarative configuration language. It specifies a data flow property—namely, how packets will flow between elements—and is therefore a dataflow language, albeit a very simple one. This section describes the language in some detail, particularly the properties that made the optimization tools of Section 4 possible.

The core language has two constructs: declaring elements and declaring connections. The ‘n :: C(s)’ statement declares an element named n with element class C and, optionally, configuration string s. The ‘a [p] -> [q] b’ statement creates a connection between two named elements (specifically, between a’s output port p and b’s input port q). There is some syntactic sugar to enhance readability; for example, ‘a -> b’ is equivalent to ‘a [0] -> [0] b’, and new elements can be declared inside a connection. Figure 5 shows the language in use; it describes a portion of the IP router in Figure 3.

Since the Click language is wholly declarative—it shows how elements should be connected, rather than describing procedurally how individual packets should be processed—a router configuration can be considered apart from its eventual execution environment. (In contrast, systems such as Berkeley *ns* [17] do not separate configuration from execution.) This facilitates the construction of tools, such as our optimizers, that process router configurations off line, as do these features:

- Any program in the configuration language corresponds to exactly one configuration graph, and a functionally equivalent program can be recovered from any configuration graph. Thus, tools can manipulate configuration graphs without losing important information when the graphs are translated back into programs. This requirement affected the design of higher-level language features like compound elements (Section 3.1).

- A configuration file can be unambiguously parsed without prior knowledge

```
strip :: Strip(14);
lookup_route :: LookupIPRoute(...);
from eth0 -> Paint(1)
        -> strip
        -> CheckIPHeader(...)
        -> GetIPAddress(16)
        -> lookup_route;
lookup_route[1] -> DropBroadcasts
        -> cp0 :: CheckPaint(1)
        -> gw0 :: IPGWOptions(1.0.0.1)
        -> FixIPSrc(1.0.0.1)
        -> dt0 :: DecIPTTL
        -> fr0 :: IPFragmenter(1500)
        -> to eth0;
```

Figure 5: A portion of a Click-language description of Figure 3.

of the element classes it uses. Thus, tools can meaningfully manipulate router configurations without understanding their components.

- Configuration files can be archives that contain a Click-language program plus arbitrary collections of data. Tools use archives to store extra information about the configurations they produce. In particular, tools that generate new C++ element classes store the resulting object files in an archive along with the configuration program. The system will automatically link against those object files when installing that configuration.

3.1 Compound elements

Users can extend Click's collection of elements by writing new element classes in C++ or by creating *compound elements* in the Click language. Compound elements are configuration fragments that can be named and created like single elements. They behave like inlined functions for configuration graphs, and provide an easy way to abstract common code into a library. They are also useful for other system components; for example, the pattern optimizer (Section 4.1) uses compound elements to specify patterns and their replacements.

Compound elements are just configuration fragments enclosed in braces. Connections between the compound and the rest of the router take place through two special elements, `input` and `output`; see Figure 6. Using a compound element is exactly equivalent to using its component elements, since the compound's boundary is compiled away by the language parser. This means that optimizations and analyses need no special processing to handle compound elements—as a first step, they can compile compounds away.

```
elementclass SFQ {
  h :: HashDemux(...);
  s :: RoundRobinSched;
  input -> h; s -> output;
  h[0] -> Queue -> [0]s;
  h[1] -> Queue -> [1]s;
}
```

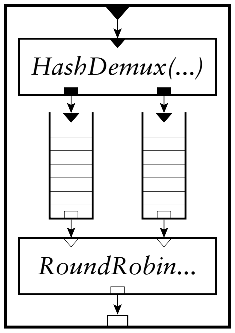


Figure 6: A compound element implementing a simple stochastic fair queue. Its language definition is on the left; a diagram of the compound is on the right.

3.2 Discussion

The configuration language specifies what elements should be created and how they are to be connected together, but it does not specify the semantics of the element classes themselves. In our system, different language processors have different ideas of the semantics of each element class. For example, the router itself defines each element class by its C++ implementation, but the tools described in the next section need much less information—for example, several tools only need to know which ports are push and which are pull. Rather than attempt to extract this information from the C++ implementations—a difficult, if not impossible, task—we supply several parallel definitions for each element class. For example, each element has a C++ definition and a push/pull definition; some elements have additional definitions as required by the tools. These additional definitions are textually embedded in the C++ definition.

In this approach, the user creates simple, high-level specifications for each element–property pair. These specifications are focused and easy to write, and it facilitates incremental specification—the behavior of an element is specified as tools are built. However, this forces the user to write many specifications for each element, and as new tools become available, old element classes may need to be updated with new specification information.

In contrast, one could write each element once in a language that facilitated analysis, then extract all necessary information from that single definition. For example, in the Ensemble system [14] for composing network protocol stacks, components are written in Objective Caml. However, Objective Caml's generality means that a complex theorem prover is required to analyze a configuration.

4 Language optimizations

This section describes particular tools that manipulate router configuration programs. Each tool reads a configuration, analyzes and/or manipulates that configuration, reports errors, and writes out the (possibly modified) configuration. The output of one tool is suitable as input for another, so tools can be composed in any user-specified order.

We present a tool for transforming a configuration based on user-specified patterns; two optimization tools that generate C++ source code; a tool that proves simple properties about a router, namely packet data alignment; and a tool that combines two or more router definitions into one "virtual router", enabling global optimizations and analyses on a collection of routers. We conclude with some minor tools and discussion.

We categorize each tool based on three properties:

1. **Language-only or language plus source code?** Some tools work entirely at the configuration language level, while others additionally generate new C++ source code.

2. **Element semantics.** Some tools work without understanding the semantics of individual elements. Others need to know some element semantics, ranging from simple (are the ports push or pull?) to more complex (how does this element affect packet data alignment?).

3. **Graph analyses and manipulations.** Some tools require only simple graph analyses and manipulations; others run different relaxation-based data flow algorithms on graphs; and others perform expensive graph operations like subgraph isomorphism calculations.

4.1 Pattern replacement

The pattern replacement tool, *click-xform*, is a generic search-and-replace tool for configuration graphs. It accepts a router configuration and a collection of patterns and replacements. It looks in the configuration for occurrences of each pattern, replacing each match it finds with the corresponding replacement. When there are no more occurrences of any pattern, it outputs the transformed configuration.

Click-xform patterns are quite general—in fact, they are as general as compound elements, for patterns and replacements are both written as compound elements. A pattern compound element matches a subset of a router configuration iff replacing that subset with an occurrence of the compound would result in an identical configuration (modulo element names). This definition is quite flexible—for example, the user can specify that the pattern must be isolated from the rest of the router, or that the pattern is connected to the router in a constrained way. Searching a configuration graph for an occurrence of a pattern is a variant of subgraph polymorphism, a well-known NP-complete problem. Luckily, the patterns and router configurations seen in practice are well-served by Ullmann's algorithm for subgraph polymorphism [20], and *click-xform*'s observed performance is good, taking about one minute on router graphs with thousands of elements (and much less on normal-sized routers).

The pattern replacement tool has uses for optimization and for generic router transformation. One simple optimization is replacing a slow element, or a collection

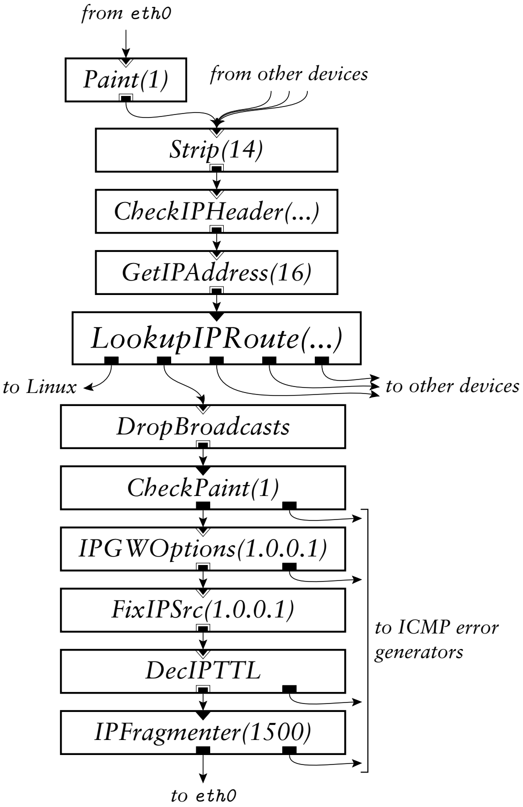


Figure 7: A portion of the IP router configuration (Figure 3), corresponding to the language fragment in Figure 5.

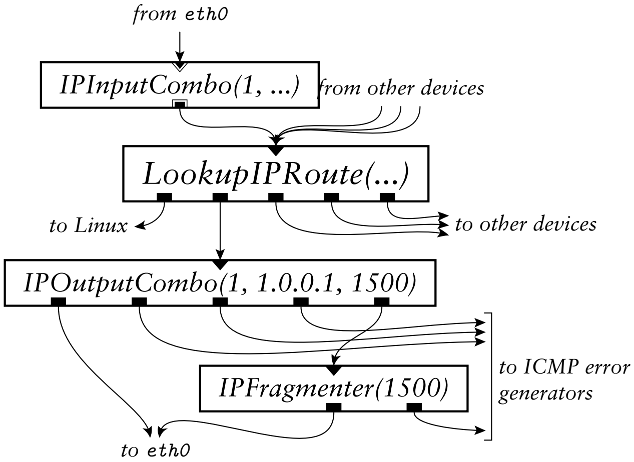
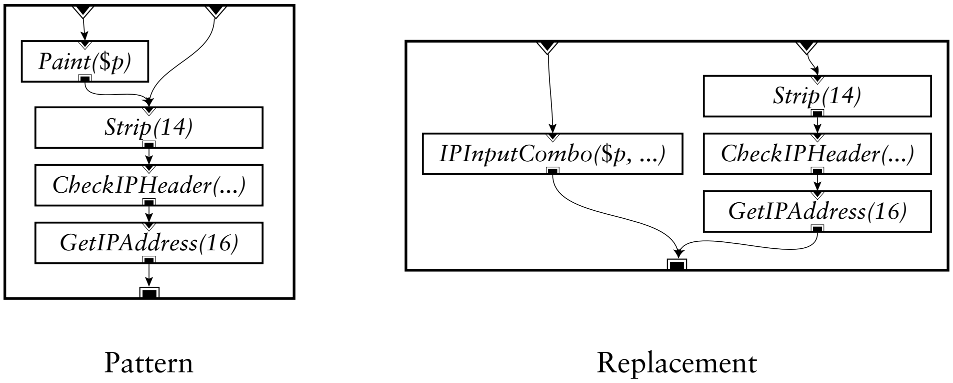


Figure 8: A router fragment equivalent to Figure 7 using faster "combo" elements.



Pattern

Replacement

Figure 9: A pattern/replacement pair suitable for the *click-xform* pattern replacement tool. This is one of three patterns that, together, can transform Figure 7 into Figure 8.

of slow elements, with a single more specialized and faster element. For example, we have designed elements that combine the work of common IP router elements; Figure 8 shows a router fragment equivalent to Figure 7, but using these combination elements. This optimization can be done by hand, but that is tedious, error-prone, and inflexible: the faster element is often more complex and less general than the slower elements. Given an appropriate set of patterns—including the pattern/replacement pair shown in Figure 9—*click-xform* can automatically transform the router fragment in Figure 7 into Figure 8, gaining speed without sacrificing modularity.

As an example of a generic transformation enabled by *click-xform*, a simple pair of patterns can add random early detection, or RED [9], to any router by inserting a *RED* element before every *Queue* in the configuration.

The pattern replacement tool works entirely at the language level and without any knowledge of element semantics. (Semantic knowledge is encoded in the patterns and replacements, but these are written by a user.) It performs extensive, and expensive, graph analysis and manipulation.

Pattern replacement has been implemented in several systems—for example, Scout [18] has a rule-based global optimizer, and instruction selectors like BURG [10] attack similar problems for tree structures. *Click-xform*'s pattern language is more powerful than any of these, however.

4.2 Fast classifiers

The fast classifier tool, *click-fastclassifier*, optimizes configurations that contain *Classifier* elements. *Classifier* is a generic packet classification element; it has one input port and many output ports, and sends each incoming packet to one of its outputs based on analysis of the packet's data. In particular, *Classifier* traverses a decision graph whose intermediate nodes check packet data against fixed values, and whose leaves represent particular output ports. The normal *Classifier* element builds this decision tree when the router is initialized and represents it in a data structure. *Click-fastclassifier* makes the configuration faster by compiling the decision tree into C++ code. Each different *Classifier* is compiled into a different C++ element class.

Click-fastclassifier also combines adjacent *Classifier* elements when possible, creating a single fast classifier that does the work of both. This reduces the number of elements a packet must travel through, and is particularly useful in the presence of compound elements, where a user may create adjacent *Classifier*s without realizing it (since the *Classifier*s are encapsulated inside compounds).

The fast classifier tool both changes the configuration program and compiles new element classes. It depends intimately on the semantics of *Classifier*, but requires little graph analysis or manipulation.

Similar optimization strategies have been developed for BPF [2], DPF [8], and other packet filters. The Click language provides a generic context for this filter optimization work, showing that it exemplifies a class of optimizations—where a faster element is swapped in for a slower one. For example, a tool that generated new element classes for some other element could share much of *click-fastclassifier*'s code.

4.3 Devirtualization

The element devirtualizer tool, *click-devirtualize*, optimizes configurations by removing virtual function calls from their elements' source code. Virtual function calls, also called dynamic dispatches, are made through a table of function pointers; on a Pentium III, a mis-predicted virtual function call takes dozens of cycles. (A correctly predicted call takes about 7 cycles, similar to direct function calls.) Packet handoff between elements is implemented in Click with virtual function calls, since element source code is written before the element's context is known. It would be possible to use a direct function call if the element author knew the next element's class, but Click was designed to avoid that kind of fixed dependency. However, once the configuration program is known, every virtual function call in the system could conceptually be removed.

Click-devirtualize automatically does exactly this. The tool reads a configuration program, then reads and partially parses the C++ source code for each element class used in that configuration. Then it generates new C++ element classes—one per element—where each virtual function call for packet handoff has been replaced with the right direct function call. For example, consider an element whose first output port is connected to the first input port of a *Counter* element. Then code like this, in the normal element class,

```
Element *next = output(0).element();
// call goes through virtual function table
next->push(output(0).port(), packet_ptr);
```

is transformed by *click-devirtualize* into code like this:

```
Counter *next = (Counter *)output(0).element();
// call is resolved at compile time
```

```
next->Counter::push(0, packet_ptr);
```

The next input port number has been inlined: the reference to '`output(0).port()`' has been changed to '`0`'. The actual transformation inlines other method calls as well, such as those that return how many inputs or outputs an element has.

While *click-devirtualize* can generate a new element class for every element, it usually does not, because even specialized elements can often share code. For example, all *Discard* elements can share code, since *Discard* throws away every packet it receives: there are no virtual function calls to devirtualize. Because of this, two elements can share code if they have the same class—say, *Counter*—and they are each connected to a single *Discard* element. This is because the two *Discard* elements share an element class, and therefore the `push` virtual function calls in the *Counter*s both resolve to the same static function (namely, `Discard::push`). Similarly, two elements with the same class, each connected to one of the two *Counter*s, can also share code, and so forth. Two elements *cannot* share code if any of the following properties is true:

1. The elements have different classes.

2. The elements have different numbers of input or output ports.

3. There exists an input or output port where that port is push on one element, but pull on the other.

4. There exists a pull input port, or a push output port, where the elements connected to that port cannot share code. (For example, the port is connected to a *Counter* on one element, but a *Strip* on the other.)

In our IP router configurations, such as Figure 3, analogous elements in different interface paths can always share code.

Virtual function calls do have some advantages. For example, infrequently executed paths may not be important enough to devirtualize, particularly if there are several such paths whose elements cannot share code; in this case, the i-cache cost of expanded source code may outweigh the performance savings of removing virtual functions. To address this, *click-devirtualize* can be told that certain elements should not be devirtualized.

Click-devirtualize can inline function calls as well as devirtualizing them. This can improve performance significantly, but it currently requires some hand intervention.

The devirtualization tool both changes the configuration program and compiles new element classes. It depends only on the push/pull properties of each element, not on more complex semantic properties; although it requires that every element's source code be available, it does not analyze or understand this source code in depth. It requires configuration graph analysis, both to discover the push/pull properties of the graph and to determine whether elements can share code. Both analyses are relatively simple data flow algorithms. It does not require substantial graph manipulation.

Devirtualization is a well-known technique in object-oriented programming languages such as Java. Mosberger et al. [16] demonstrate that *path inlining*, essentially devirtualization with inlining, is useful for decreasing protocol latency in a modular networking system (the *x*-kernel [11]), but they implement it by hand. To our knowledge, neither the *x*-kernel nor Scout [18] can implement devirtualization automatically.[1]

4.4 Packet data alignment

The packet data alignment tool, *click-align*, ensures that packet data is aligned correctly within a router. The Click packet abstraction follows Linux's example; it is basically a flat array of bytes. However, many elements require that the packet data is correctly aligned on a word boundary. For example, the IP header should be word-aligned or elements such as *CheckIPHeader* will fail. Other elements, such as *Classifier*, can adapt to any single alignment, but require that every packet has the same alignment. The *click-align* tool solves these problems by ensuring elements have the alignments they require, and telling elements what packet alignment they can expect.

Packet data alignment is a global property. It depends both on the initial alignment and on any modifications that happened to the packet on its path through the router. For example, the *Strip* element strips a specified number of bytes from the packet's header by bumping a pointer; a packet's alignment after passing through *Strip(1)* will be different than its alignment before. The alignment problem could be solved by inserting *Align* elements anywhere a particular alignment is requested, but this would be quite expensive—*Align* fixes alignment problems by making packet copies.

A better solution is to add *Align* elements only where they are needed. *Click-align* calculates the packet data alignment at every point in a configuration by performing a simple data flow analysis, resembling availability analysis [1], that takes alignment-modifying elements into account. It then inserts *Align* elements at every point where the existing alignment is incorrect and reruns the analysis. Finally, it removes redundant *Align* elements and adds an *AlignmentInfo* element, which informs every other element of the alignment that element can expect. Our IP router configuration requires no *Align* elements, but the tool is necessary anyway, as it provides the *Classifier* element with alignment information.

On the Intel x86 architecture, the *click-align* tool is optional—unaligned accesses are legal, and are not any slower than aligned accesses. (An unaligned access can cause two d-cache misses if it straddles two cache lines, but this is not a problem in practice, as the packet data is always in the cache.) On other architectures, however, configurations that have not been checked with *click-align* cause Click to crash.

The alignment tool works entirely at the language level. It does require knowledge of element semantics—specifically, the alignment-related semantics of each element. It relies on relatively simple data flow analysis and graph manipulations.

[1] Scout paths are specialized automatically, but this optimization largely consists of removing unneeded queues. Click avoids unneeded queues a priori.

4.5 Multiple routers

The *click-combine* tool can combine several router configurations into one larger configuration that gives a detailed picture of the overall network. For example, if the output interface of one router is connected to the input interface of another by a point-to-point link, then the combined configuration will have explicit connections between the relevant interfaces. This unified Click graph can be analyzed and manipulated using language techniques like the ones described above, ensuring properties of the network—for instance, that the frame formats at either end of each link are compatible—and optimizing away redundant computation performed by more than one router. The *click-uncombine* tool can then separate the combined configuration into its component router parts. The rest of this section describes three possible multiple-router optimizations: removing redundant IP fragmentation checks, removing redundant IP header checks, and ARP query optimization.

The *IPFragmenter* element in Figure 3 fragments the packet about to be transmitted into packets no larger than the link's MTU. However, if every path in the unified Click graph—and hence every network link—upstream of that element contains an *IPFragmenter* with a same or smaller MTU, then the packet already has the appropriate size, and no fragmentation is necessary. A simple availability analysis over the combined flow graph can determine this condition and remove redundant fragmenters.

Similarly, *CheckIPHeader* is not always necessary. This element only forwards a packet if the packet's length is reasonable, the checksum and certain other IP fields are valid, and the IP source address is a legal unicast address. Assuming that bit errors within the network are rare, and that the remaining ones can be caught at the endpoints, *CheckIPHeader* only needs to run once, at the entry to the network. Again, we can perform an availability analysis over the unified Click graph, and eliminate any instance of *CheckIPHeader* that is preceded on all paths by another instance of that element. (In this case, we must also ensure at the upstream instances are not invalidated by subsequent elements that may damage the IP header. One example of such an element is *RandomBitErrors*, which randomly flips bits on the packets it forwards.)

Finally, if two routers are connected by a point-to-point Ethernet link, there is no need to have a full ARP mechanism in place on that link. To perform this optimization, we may search the graph downstream of an *ARPQuerier* element: if ARP packets can reach at most one *ARPResponder*, and packets at that *ARPResponder* must have passed through that *ARPQuerier*, then both elements can be removed, and the *ARPQuerier* can be replaced by an *EtherEncap* element that prepends the appropriate Ethernet header. Furthermore, since the downstream interface will no longer receive ARP queries or responses, the associated *Classifier* can also be removed.

These optimizations are inherently dangerous. They produce correct results only if the user correctly and completely represents every router attached to a particular link as a Click configuration. Furthermore, the user must re-run the optimizations every time the devices on a link are changed. These requirements are only realistic between routers controlled by the same administrative entity, or where the routers could exchange information about their configurations automatically. A safer use for the *click-combine* and *click-uncombine* would be to check for properties like loop freedom. However, to demonstrate what optimizations are possible, we report performance results for a simple version of the ARP optimization in Section 5. (We have not yet fully implemented the standalone ARP optimization; instead, we used a set of patterns and the *click-xform* pattern replacer tool.)

Click-combine, *click-uncombine*, and the hypothetical multiple-router optimizations all work entirely at the language level. The *click-combine* and *click-uncombine* tools do not need to understand element semantics, but the optimizations do—to understand that *RandomBitErrors* can destroy a valid IP header, for example. *Click-combine* and *click-uncombine* perform extensive, but simple, graph manipulations and analyses. The analyses required for the optimization tools are more complex.

Multiple-router optimizations resemble some kinds of programming language optimizations, such as interprocedural optimizations.

4.6 Discussion

Besides the tools described above, we have also begun work on a suite of checking tools. These tools check configurations for correctness before they are installed—for example, checking the use of push and pull ports, checking that IP processing elements are preceded by an element that verifies the IP header's correctness, and so on. They work entirely at the language level, and rely on detailed element semantics and complex data flow analyses.

The router itself is a tool that manipulates router configurations, and, like the tools, it uses some programming language techniques. For example, it applies simple data flow algorithms to the configuration graph to determine push and pull processing, and to implement *flow-based router context* [13] (a way for one element to find another, distant element).

5 Performance results

This section presents performance measurements for unoptimized and optimized Click routers.

5.1 Experimental Platform

Our testing configuration has a total of 9 Intel PCs running Linux 2.2.14: four source hosts, the router being tested, and four destination hosts.

Our router hardware is a 700 MHz Intel Pentium III CPU, an Intel L440GX+ motherboard, and eight DEC 21140 Tulip [6] 100 Mbit PCI Ethernet controllers (on multi-port cards). The L440GX+ motherboard has two 32-bit 33 MHz PCI buses, and the Ethernet controllers were evenly divided between them. The Pentium III has a 16 KB L1 instruction cache, a 16 KB L1 data cache, and a 256 KB L2 unified cache.

The Click router uses the IP router configuration in Figure 3, modified to have a total of eight input and output interfaces rather than two. Unless otherwise specified, performance numbers given in this section and the next section refer to experiments using this router configuration.

The source and destination hosts are 200 MHz Pentium Pro CPUs, each with a DEC 21140 Ethernet controller. The source-to-router and router-to-destination links are point-to-point full-duplex 100 Mbit Ethernet. The source hosts generate UDP packets using a Click configuration. They produce packets at specified rates, and can generate up to 147,900 64-byte packets per second. The destination hosts count and discard the forwarded UDP packets using another Click configuration. The 64 bytes include a 14-byte Ethernet header, a 20-byte IP header, an 8-byte UDP header, 18 null bytes, and a 4-byte Ethernet CRC. When the 64-bit preamble and 96-bit inter-frame gap are added, a 100 Mbit Ethernet link can carry up to 148,800 such packets per second.

The base Click system uses augmented device drivers that support polling as well as interrupts. (Conventional Linux device drivers only support interrupts.) Interrupt overhead and device handling dominate the performance of a non-polling Click system, consuming over 80% of the time required to forward a packet and leading to receive livelock [15]. The polling system avoids much of this overhead by entirely eliminating interrupts and programmed I/O interaction with device hardware. (In the conventional system, some PIOs were only necessary to manage the device's interrupt generation logic. To remove the last PIO, we configured the Tulip to check the transmit DMA queue periodically for new packets. As a result, all communication between Click and the device hardware takes place indirectly through the shared DMA queues in main memory, in the style of the Orion [7] network adaptor.)

Polling Click runs as a single kernel thread executing a task queue. Most of the tasks on this queuecorrespond to the router configuration's *PollDevice* and *ToDevice* elements. A *PollDevice* task examines its device's receive DMA queue for newly arrived packets and pushes them through the configuration, while a *ToDevice* task examines its device's transmit DMA queue for empty slots which it tries to fill by pulling packets from its input.

5.2 Effects of language optimizations

Table 1 breaks down the costs of forwarding a packet on an unoptimized Click router. Costs are measured in nanoseconds by accumulating Pentium III cycle counters [12] before and after each block of code, and dividing the totals over a 10-second run by the total number of packets forwarded. **Polling packet** is the time *PollDevice* spends taking a packet from Tulip's receive DMA ring. **Refill receive DMA ring** is the time *PollDevice* spends replacing the DMA descriptor of the packet it just received with a new descriptor, so that the Tulip may receive a new packet. The Click IP forwarding path (see Figure 3) is broken down into a push path and a pull path. An input packet is pushed through the forwarding path by the *PollDevice* element until it reaches the *Queue* element before the appropriate transmitter. When the *ToDevice* element is

Task	**Time**(ns/packet)
Polling packet	528
Refill receive DMA ring	90
Push through Click forwarding path	1565
Pull from Click queue	103
Enqueue packet for transmit	161
Clean transmit DMA ring	351
Total	**2798**

Table 1: Microbenchmarks of tasks involved in the IP forwarding path of an unoptimized Click router.

ready to send, it pulls from *Queue*. **Enqueue packet for transmit** is the time *ToDevice* spends enqueuing a packet onto the Tulip's transmit DMA ring. **Clean transmit DMA ring** is the time *ToDevice* spends removing DMA descriptors of transmitted packets. The total cost of 2798 ns measured with the performance counters implies a maximum forwarding rate of about 357,000 packets per second, consistent with the observed maximum loss free forwarding rate (MLFFR) of 360,000 packets per second.

The language optimizations described in this paper reduce the cost of the push through the Click forwarding path, shown in the third row of Table 1. Figure 10 illustrates the effects of these language optimizations. The leftmost column represents the cost of pushing a packet through the Click forwarding path when no optimizations are performed. Applying all router-local optimizations—fast classifier, devirtualizer, and pattern replacement—to a router graph reduces the push cost from 1562 ns to 948 ns. Applying multiple-router optimization alone reduces the push cost from 1562 ns to 1276 ns. Applying both multiple-router and router-local optimizations reduces the cost from 1562 ns to 764 ns.

Of the three router-local optimizations, pattern replacement is the most effective. It reduces the cost of the push path by 33%. Although the devirtualizer provides a similar performance improvement, its optimization opportunities overlap with those of pattern replacement. As a result, applying both of these optimizations is not much more useful than applying either one alone.

Although the fast classifier optimization reduces the cost of the push path only slightly (to be precise, by 36 cycles), it improves the performance of the *Classifier* element by an average of 22%. Before applying the optimization, the average time spent in the *Classifier* element is 121 cycles per packet. After applying the fast classifier optimization, this number decreases to 94 cycles per packet.

Optimization opportunities that result from analysis of multiple routers are independent from those of pattern replacement and devirtualization. In our example, fast classifier no longer applies because the *Classifier* element is optimized away.

Forwarding a packet through Click incurs just four data cache misses (measured using Pentium III performance counters): one to load the receive DMA descriptor, two to read the packet's Ethernet and IP headers, and one to remove the packet from the transmit DMA queue after it has been sent. Click runs without incurring any other

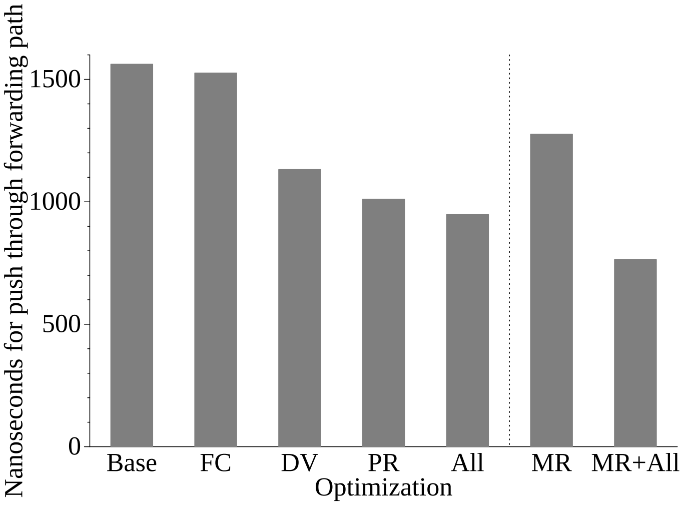


Figure 10: Effects of language optimizations on Click. **Base** denotes the time required to push a packet through the Click forwarding path when no optimizations are applied. The following three columns indicate the time when only the indicated optimization is performed: **FC** is "fast classifier", **DV** is "devirtualizer", and **PR** is "pattern replacement". **All** denotes all three optimizations applied together. **MR** denotes multiple-router optimization. **MR+All** denotes optimization across multiple routers, combined with all the other local optimizations.

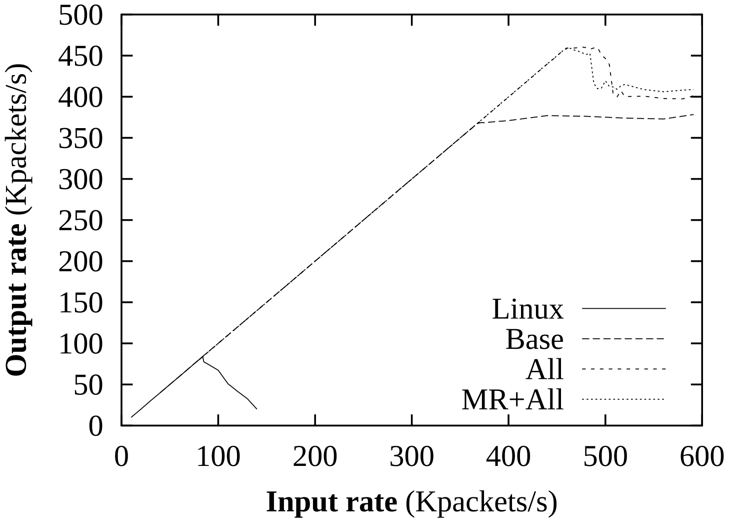


Figure 11: Forwarding rate as a function of input rate for 64-byte packets. An ideal router that forwarded every packet would appear as a straight line $y = x$. The **Linux** plot shows the performance of a standard Linux IP router. The **Base** plot shows an unoptimized polling Click. The **All** plot shows the effect of all three language optimizations: pattern replacement, fast classifier, and devirtualizer. The **MR+All** plot shows the effects of these optimizations in addition to multiple-router optimization.

data or instruction cache misses. With all three router-local optimizations turned on, just 988 instructions are retired during the forwarding of a packet. This implies that significantly more complex Click configurations could be supported without exhausting the Pentium III's 16K L1 instruction cache.

5.3 Forwarding rates

An unoptimized polling Click router has an MLFFR of 360,000 packets per second. It does not suffer from receive livelock under high input load. (Compare an ordinary, interrupt-driven Linux IP router, which has a MLFFR of 84,000 packets per second and exhibits receive livelock.) After applying language optimizations—pattern replacement, devirtualization, and fast classifier optimizations—Click has an MLFFR of 456,000 packets per second, a 6 to 1 improvement over the original version of the Click router [13].

5.4 PCI limitations

As shown in Figure 10, pushing a packet through the Click forwarding path when all optimizations are turned on (**MR+All**) requires fewer cycles than when multiple-router optimizations are not used (**All**). As a result, the per-packet forwarding cost—including the other tasks listed in Table 1—is 2.0 μs in the **MR+All** case, and 2.19 μs in the **All** case. Despite this fact, the MLFFR in the **MR+All** case is no higher than that in the **All** case. This suggests that packet processing in Click is not a bottleneck, and that the language optimizations have shifted the bottleneck to other components of the system.

The 2.19 μs per packet forwarding cost for the **All** case implies a theoretical MLFFR of 456,000 packets per second if Click were a bottleneck. An observed MLFFR of 456,000 packets per second, therefore, indicates that Click is indeed a bottleneck. However, as the input rate increases beyond 456,000 packets per second, the forwarding rate of the router first flattens, then drops, then flattens again. At an input rate of 500,000 packets per second, where the forwarding rate in the **All** case begins to drop, all eight Tulip cards indicate that packets are being dropped because the DMA engine cannot transfer data to main memory quickly enough, a sign that both PCI buses are congested.

In the **MR+All** case, the Tulips begin to report these errors immediately after they reach the observed MLFFR of 456,000 packets per second. This is reflected in Figure 11 by the fact that the **MR+All** curve drops immediately after reaching the MLFFR, and suggests that 456,000 64-byte packets per second is very close to the limit achievable with our dual PCI buses.

6 Related work

In this section, we compare our work with other research into software routers on commodity hardware, focusing on performance research and flexible routers.

Scout [18] is a new operating system designed for high performance networking. The basic abstraction of Scout is the path, which are somewhat similar to Click elements, although there are substantial differences. Like our Click language optimizations, Scout features several transformations on paths to improve performance. Optimizations implemented by hand for a Scout TCP forwarder [19] address similar problems to those we attacked for our IP router—for instance, the number of components on the forwarding path. The Scout IP router has reported performance roughly comparable to unmodified Linux [19].

The Router Plugins system [5] allows configurable packet processing by placing "gates" at fixed points in the NetBSD IP forwarding code; packets passing the gates are passed to "plugin" modules selected by a flow classifier. The plugins system is reported to forward about 34,100 packets per second on a 233 MHz Pentium II; the unmodified NetBSD code forwards 36,800.

ALTQ [3] provides configurable traffic management in FreeBSD. Unlike Click, ALTQ does not provide configurability beyond specification of one queuing policy per output interface.

Mogul and Ramakrishnan [15] describe the problem of receive livelock and analyze polling as a solution.

Click interacts with the Tulip [6] hardware using techniques similar to those used by Osiris [7]. Both eliminate programmed I/O interactions between device hardware and driver, so that the two communicate in a de-coupled fashion using DMA descriptor rings. As a router, Click uses network interfaces somewhat differently than most hosts. Click loads only packet headers into the CPU and caches; packet payload can remain in RAM only. Click also has more relaxed buffering requirements than hosts, since it doesn't need to move data into user space. These considerations make pure DMA more attractive for Click than the CPU-directed copying used by host adaptors like the Afterburner [4].

7 Conclusion

We have presented a collection of language-based tools for analyzing and optimizing Click router configurations. These tools apply common programming-language techniques to Click's simple and declarative configuration language, obtaining optimizations of the system as a whole. The uniform framework provided by the language makes it easy to design novel optimizations. For example, tools that combine multiple router configurations into a single configuration let us optimize routers using information about the entire network. Together with device driver and polling improvements, the result is a flexible IP router that can route 456,000 64-byte packets per second on a 700 MHz Pentium III.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *Proc. ACM SIGCOMM Conference (SIGCOMM '99)*, pages 123–134, August 1999.

[3] Kenjiro Cho. A framework for alternate queueing: towards traffic management by PC-UNIX based routers. In *Proc. USENIX 1998 Annual Technical Conference*, pages 247–258, June 1998.

[4] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network Magazine*, pages 36–43, July 1993.

[5] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. In *Proc. ACM SIGCOMM Conference (SIGCOMM '98)*, pages 229–240, October 1998.

[6] Digital Equipment Corporation. *DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller Hardware Reference Manual*, March 1998. `http://developer.intel.com/design/network/manuals`.

[7] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proc. ACM SIGCOMM Conference (SIGCOMM '94)*, pages 2–13, August 1994.

[8] Dawson Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. ACM SIGCOMM Conference (SIGCOMM '96)*, pages 53–59, August 1996.

[9] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Networking*, 1(4):397–413, August 1993.

[10] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG—fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[11] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: an architecture for implementing network protocols. *IEEE Trans. Software Engineering*, 17(1):64–76, January 1991.

[12] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3*, 1996. `http://developer.intel.com/design/pro/manuals`.

[13] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. Computer Systems*, 18(4), November 2000.

[14] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *Proc. of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–92, December 1999.

[15] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Computer Systems*, 15(3):217–252, August 1997.

[16] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of techniques to improve protocol processing latency. In *Proc. ACM SIGCOMM Conference (SIGCOMM '96)*, pages 73–84, August 1996.

[17] UCB/LBNL/VINT network simulator NS homepage. Available from http://www-mash.cs.berkeley.edu/ns/.

[18] Larry L. Peterson, Scott C. Karlin, and Kai Li. OS support for general-purpose routers. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 38–43. IEEE Computer Society Technical Committee on Operating Systems, March 1999.

[19] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Trans. Networking*, April 2000.

[20] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976.