Programming language optimizations for modular router configurations[‡]

Eddie Kohler[*] Robert Morris[†] Benjie Chen[†]

[*]ICSI Center for Internet Research [†]MIT Lab for Computer Science

kohler@icir.org, {rtm, benjie}@lcs.mit.edu

Abstract

Networking systems such as Ensemble, the *x*-kernel, Scout, and Click achieve flexibility by building routers and other packet processors from modular components. Unfortunately, component designs are often slower than purpose-built code, and routers in particular have stringent efficiency requirements. This paper addresses the efficiency problems of one component-based router, Click, through optimization tools inspired in part by compiler optimization passes. This pragmatic approach can result in significant performance improvements; for example, the combination of three optimizations reduces the amount of CPU time Click requires to process a packet in a simple IP router by 34%. We present several optimization tools, describe how those tools affected the design of Click itself, and present detailed evaluations of Click's performance with and without optimization.

1 Introduction

Modular components make systems more flexible and extensible. Different compositions of the same components can implement fundamentally different functionality. Furthermore, the use of fine-grained components with simple specifications can make a system easier to understand. Because of its requirements for flexibility and extensibility, and its difficulty, networking software has been a popular field for the application of component techniques [6, 8, 9, 11, 13, 15].

Networks get faster at an even greater rate than processors, however, making the efficiency of networking software ever more important. Even as component systems make networking software easier to program, component techniques introduce inefficiencies that monolithic software can avoid.

One way to avoid component overhead is to apply compiler optimization techniques at the component level. For example, object-oriented optimizations such as static class analysis can reduce the cost of communication between components. Dead code elimination, instruction or component selection, and inlining also have obvious applications.

Starting from this principle, we have developed several optimizers for Click, a system for building extensible routers from modular components [11]. The optimizers read Click router configurations on standard input, analyze and transform them in various ways, and write the optimized configurations to standard output. They are thus easily combined, much like compiler optimization passes, but unlike compiler optimization passes, they work at the level of router configurations. Our optimizations run quickly and make real routers significantly faster. For example, the combination of three optimization tools reduces the Click CPU time cost of an IP router by 34%, and raises its peak forwarding rate by 89,000 minimum-size packets per second, to 446,000. (On a newer PC, this peak forwarding rate jumps to 740,000 packets per second.) For high input rates, the performance of our optimized Click IP router is limited by our I/O system, not by the CPU time cost of Click.

The optimization tools form the main contribution of this work. They demonstrate that compiler optimizations can usefully be applied at the level of networking components. In particular, we hope to show that analogies with compiler optimizations and programming language techniques can usefully guide the design of pragmatic tools for optimizing modular routers, and, perhaps, component systems in general. We also discuss the original Click system's performance issues and relationships between the optimizers and Click's design, and provide a detailed analysis of the forwarding performance of an optimized Click IP router on several types of PC hardware.

In the rest of this paper, we discuss related work (Section 2), present aspects of Click relevant for performance

[‡]This version contains some corrections relative to the version published in ASPLOS X. Section 6.2 previously referred to "subgraph *poly*morphism", not "subgraph *iso*morphism".

(Section 3), describe our optimization approach in general (Section 5) and particular optimizer tools (Sections 4, 6, and 7), present performance results (Section 8), and conclude (Section 9).

2 Related Work

Previous work has described the Click modular router and the IP router configuration on which we base our evaluation [11].

The *x*-kernel [8] provides a framework for implementing and composing network protocols. Protocol nodes in the *x*-kernel resemble Click elements. Hand optimization of *x*-kernel configurations demonstrated that *path inlining*, which combines the effect of our devirtualizer (Section 6.1) with inlining, can significantly decrease protocol latency [12]. Automatic configuration optimization is not supported.

Scout [13], a successor to the *x*-kernel, was designed for routing and high performance networking, rather than protocol composition. Scout comes with a simple rule-based optimizer similar to our *click-xform* tool (Section 6.2). However, their optimizer transforms paths, or linear chains of "elements"; *click-xform* transforms subgraphs, which is much more powerful. Optimizations implemented by hand for a Scout TCP forwarder [16] address similar problems to those we attacked in our IP router—for instance, the number of components on the forwarding path.

Dynamic modular networking systems, such as System V STREAMS [15] and FreeBSD's Netgraph [6], focus on the dynamic construction and manipulation of configurations. A complete configuration cannot be easily extracted from these systems, so optimizing a configuration must proceed piecemeal. We know of no automatic optimizers for these systems.

Previous work applying compiler optimizations to components and systems tends to be more integrated with traditional compiler technology. The Ensemble [9] and Esterel [4] projects, for example, use language processors that understand the entirety of the programming languages used to write their systems' components. In contrast, our optimizers understand components at a high level; some do not handle C++, the implementation language, at all. The Ensemble/Esterel approach affords potentially greater opportunities for optimization, but it also leads to far more complex language processors.

In particular, Ensemble is a component-based network protocol architecture designed especially for group membership and communication protocols. Unlike Click, the *x*-kernel, Scout, STREAMS, and Netgraph, which all use conventional systems programming languages like C or C++, Ensemble components are written in OCaml, a functional programming language. The Ensemble authors translate their OCaml components into formal statements in the Nuprl theorem proving system. From there, they can extract optimized versions of each node and compose them according to various combination predicates. Nuprl can theoretically perform many kinds of optimizations, but the process is not fully automatic. Human interaction with the Nuprl system is required to extract useful specifications from Ensemble components. This may take anything from a half-hour to an entire afternoon to develop, and requires input from both the component programmer and a Nuprl expert [9]. The optimizations implemented by Ensemble resemble a combination of our *click-devirtualize* and *click-xform*. Forwarding performance was not reported.

Knit [14], a component framework for C programs, also applies programming language techniques to systems components. Its implementation strategy, where Knit modifies object files produced by an unmodified C compiler, resembles our approach more than the integrated approach of systems like Ensemble. The Knit "flattening" optimization is like an version of our devirtualizer that additionally inlines code.

3 Click and Its Performance

Click is a component framework for PC router construction. The components are fine-grained packet processing modules called *elements*. Each element has a *class*; element class definitions are written in C++. Router configuration designers then combine elements into routers using a simple declarative language of our design. This makes Click routers flexible, extensible, and relatively easy to construct. Figure 1 shows a standard-compliant Click IP router with two network interfaces. Elements appear as boxes; arrows between boxes are *connections*, which determine how packets travel from element to element. As indicated by this diagram, Click configurations can be thought of as graphs with elements at the vertices. For more details, see [5, 11].

Component designs are often slower than purpose-built code. The cost of inter-component communication is pure overhead, and reusable components may require slower, more general algorithms. Performance is a secondary goal for Click, since the fastest core routers will never run on PCs. However, even gigabit Ethernet taxes high-end PC processors and buses, and will for years to come. There are no spare cycles; slow software means dropped packets. Therefore, Click was designed to be as efficient as possible, within the parameters of a flexible, extensible component framework.

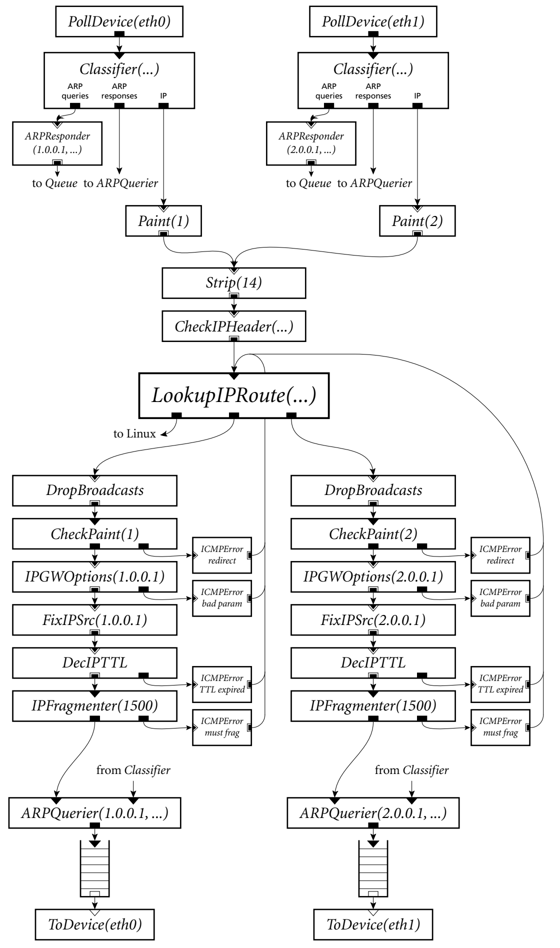

FIGURE 1—A Click IP router with two network interfaces.

Among the relevant design choices: C++, our implementation language, has low inherent overhead and happily coexists with operating system kernels. The Click packet abstraction is a thin veneer over the Linux kernel's `sk_buff`; extensive use of inline functions makes it as efficient as `sk_buff` while providing a friendlier interface. To avoid overhead, elements perform only rudimentary input checking. For example, they often assume that received packets have the expected protocol, so protocol dispatch must be made explicit in router configurations. Click replaces the host operating system's interrupt-driven network stack with polling device drivers and a constantly-active kernel thread. This important change eliminates receive livelock [10], where receive interrupt processing occupies all CPU resources and drives the forwarding rate to zero, and raises a Click IP router's peak forwarding rate to over four times that of unmodified Linux.

Nevertheless, sources of overhead remain.

• **Virtual functions and packet transfer**. Packets are transferred between elements via dynamic dispatches, or virtual function calls in C++ terminology. This is much cheaper than alternatives like implicit queues between components, but it still has inefficiencies. We investigated the cost of virtual function calls relative to conventional function calls on a Pentium III processor. The Pentium caches the targets of indirect branch instructions; when correctly predicted, a virtual function call takes about 7 cycles, comparable to a conventional function call. Incorrectly predicted calls, however, take dozens of cycles. A Click IP router's forwarding path takes 1160 cycles on this processor, making the cost of misprediction significant in percentage terms.

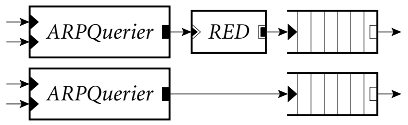

FIGURE 2—A configuration fragment that stresses the branch predictor.

Unfortunately, the branch predictor performs poorly on Click routers. For example, two elements with the same class may connect to elements with different classes, as in Figure 2. Packet transfers from the two *ARPQuerier*s share one call site, since the two elements have the same class; however, the elements transfer packets to different targets, so if packets alternate between the *ARPQuerier*s, the branch predictor is always wrong.[1]

Of course, even well-predicted virtual function calls are more expensive than no function calls at all. Click's fine-grained components are easy to work with, but lead to routers with many elements on the forwarding path—sixteen, in the case of our standards-compliant IP router. At a conservative seven cycles per packet transfer, 9% of this router's forwarding path cost is due to function call overhead.

• **General-purpose elements**. Click elements should be as general-purpose as possible. This makes it easier to reuse elements, design router configurations, and analyze configurations written by others. It also tends to make the elements themselves less efficient.

For example, many packet classification tasks in Click use programmable generic classifiers called *Classifier*, *IPFilter*, and *IPClassifier*. These elements compile textual filter specifications, such as "*src 10.0.0.2 && tcp src port 25*", into decision tree structures traversed on each packet. Almost every configuration we've written involves one or more of these elements, and initially, they were by far the slowest elements in our configurations. We sped up their

[1]Furthermore, simpler Click elements often use syntactic sugar called `simple_action` that can halve their code size, but confuses the predictor.

```
void Classifier::push(int, Packet *p) {
  const unsigned char *packet_data = p->data() - _align_offset;
  Expr *ex = &_exprs[0]; int pos = 0; // ...
  do {
    if ((*((unsigned *)(packet_data + ex[pos].offset)) & ex[pos].mask.u) == ex[pos].value.u)
      pos = ex[pos].yes;
    else
      pos = ex[pos].no;
  } while (pos > 0);
  checked_push_output(-pos, p);
}
```

a. *Classifier* inner loop

```
inline void FastClassifier_a_ac::length_unchecked_push(Packet *p) {
  const unsigned *data = (const unsigned *)(p->data() - 0);
 step_0:
  if ((data[3] & 65535U) == 8U) {
    output(0).push(p);
    return;
  }
  output(1).push(p);
  return;
}
```

b. *click-fastclassifier* output

FIGURE 3—Classification code with and without *click-fastclassifier*.

inner loops by restricting decision tree operations, and implemented an extensive set of decision tree optimizations, similar to BPF+'s [3], to optimize them further. Still, a special-purpose classification element built for a single task can beat any generic classifier. This is the essential insight behind compilers' dynamic code generation [7] and program specialization/partial evaluation techniques.

Users could address both of these sources of overhead within the Click framework. For example, they could write big, coarse-grained elements to reduce inter-element packet transfer, or write specialized elements rather than general ones. These solutions come at the cost of flexibility and extensibility and are therefore unacceptable. There would be no problem, however, if *programs* implemented the solutions instead of users. That idea motivates this work.

4 CLICK-FASTCLASSIFIER

The *click-fastclassifier* tool addresses the *Classifier* inefficiencies described above, and provides a convenient introduction to our optimization methodology.

Again, the problem is that Click's classifiers traverse a decision tree structure laid out in memory. Compare this with the approach of compiling decision trees into code with inlined constants. This always improves upon the original classifiers' data-cache usage, since there is no tree to access. The compiled code might be larger than the original code for large decision trees; still, i-cache usage and cycle counts will usually be better in the compiled version. To implement this optimization, users would create an element class for each classifier in their configuration by translating decision trees—perhaps even those generated by Click itself—into `if` statements.

The *click-fastclassifier* optimization tool automates exactly this process. In particular, it:

- Reads a Click configuration file from standard input.
- Searches the configuration for classification elements (*Classifier*, *IPFilter*, and *IPClassifier*), and combines adjacent *Classifier*s to improve optimization possibilities.
- Extracts those elements into a "harness" configuration and runs Click on the harness. Click checks the classifiers for syntax errors, creates their decision trees, then outputs those trees in human-readable form; the optimizer parses this output. Since *click-fastclassifier* uses Click to extract decision trees, classifier syntax changes need be implemented exactly once. The "harness" configuration, which contains only the classifiers and generated boilerplate, avoids possible side effects from running Click on the input configuration.
- Generates C++ source code for new element classes, one per decision tree. (Classifiers with identical decision trees use the same class.) The code consists of Click boilerplate plus a packet-handling function consisting of the decision tree translated into C++. Figure 3 shows *Classifier*'s packet-handling function and the equivalent function generated by *click-fastclassifier* for a simple classifier ("*Classifier(12/0800, –)*", which compares the twelfth and thirteenth bytes of the packet data against the hex value 0800).
- Changes each classifier element in the router configuration to use its corresponding generated class. For example, an element declaration like "*c :: Classifier(12/0800, –)*" might change to "*c :: FastClassifier@@c*".
- Writes a combination of the new router configuration and C++ source code to standard output. When the user installs this configuration, Click will first compile the source code and dynamically link with the result. This makes the new element classes accessible to the configuration.

Thus, *click-fastclassifier* transparently optimizes every classifier element in a router configuration. The original, simpler configuration need not be modified.

The speedup attributable to *click-fastclassifier* depends both on the classifiers' decision trees and on the packets passing through the classifiers. The best case is a very large tree of which most packets traverse a large fraction before being emitted. We implemented a 17-rule firewall from

Building Internet Firewalls [18, pp 691–2] in *IPFilter*, then measured *IPFilter*'s CPU cost for a packet matching the next-to-last rule (DNS-5). Without *click-fastclassifier* (and with Section 8's evaluation setup), this took 388 nanoseconds, or 23% of the total time it takes a packet to pass through the default Click IP router (excluding devices). With *click-fastclassifier*, this dropped by more than half, to 188 ns.

5 The Tool Approach

In general, Click optimization tools are programs like *click-fastclassifier* that read router configurations on standard input, analyze and transform the configurations, and output the results on standard output. All the optimizers we've built have important properties in common, and the optimizers have affected the design of Click itself.

5.1 Optimizers and Click

Optimizers don't link with element class definitions. They couldn't; elements meant for the Linux kernel require symbols and features not available at user level, for instance. Thus, optimizers do not run configurations as programs, but treat them more as graphs. A library provides an extensive set of graph manipulations—adding and removing elements and so forth.

Graph operations aren't relevant in Click itself, where configurations don't change after they are installed. To add an element to a Click router, the user must install an entirely new configuration, although this can be done in such a way that important state is transferred into the new router. In retrospect, this single decision—that router configurations should be static—allowed optimizers to exist. In some other component-based networking systems, system state builds up dynamically, through "add" and "remove" operations; a current configuration is difficult to extract from such a system, let alone install as a unit. Allowing for dynamic configuration changes would also greatly reduce available optimization opportunities; *click-fastclassifier*, for example, assumes that classifier configurations remain the same throughout a router's lifetime.

5.2 The Click language

The existence of optimizers has influenced the little language used for writing router configurations. We could have thought of this language as a script for controlling the router. Instead, we think of it as abstractly specifying a router configuration—as static and declarative, rather than dynamic and imperative. In this view, the language's sole function is to describe the collection of elements in the router and the connections between them. Clearly, such a language is more easily parsed outside the context of a router than any Tcl-like script alternative.

Optimizers cannot link with actual element definitions, so they don't necessarily know the element classes available to a router. Therefore, we changed the language so that programs can be parsed correctly without knowing which names correspond to element classes.

The optimizers have somewhat inhibited language evolution. Syntax changes and extensions are rare, for example. Extensions require updates to two parsers. More fundamentally, optimizers expect to be able to arbitrarily transform configuration graphs and generate Click-language files corresponding exactly to the results. Language extensions would have to keep this in mind. Because of these factors, we have ended up changing the Click language only to improve *compound elements*, its abstraction facility.[2]

Optimizers inspired the archive feature, where a configuration may consist of multiple files bundled into a single archive. Several tools use this feature to attach source and/or object code specialized for a single configuration. Click compiles and loads the object code before parsing the configuration itself.

An alternative design might have had optimizers call Click to parse a configuration. An option would cause Click to emit the resulting parse tree without initializing the resulting router. Unfortunately, Click and the optimizers have different parsing requirements, and supporting both in a single parser might not be significantly simpler than keeping two parsers. For example, the Click parser complains about unknown element classes, automatically compiles away compound element abstractions, aborts early when certain errors occur, and keeps only general information about the locations of errors in the source file. These choices simplify and speed up the Click parser, which must live in the Linux kernel, but are inappropriate for optimizers.

5.3 Elements and specifications

Optimizers cannot easily understand or analyze element implementations. Element classes are written in C++, whose imperative features and weak type system make it harder to analyze than languages like OCaml or Esterel. Therefore, we developed mechanisms to embed simple specifications for element properties directly in the C++ source. Scripts extract these specifications from the source and write them, in structured form, into files read by the optimizers. Click uses the specifications directly.

For example, each element must inform Click whether its input and output ports support "push" or "pull" packet

[2]Compound elements let users create libraries of common configuration fragments. Some optimizers also depend on them.

transfer.[3] Originally, this took place in unstructured code. However, our more advanced optimizers need this information; *click-devirtualize* (Section 6.1) must compile different code depending on whether an element is using push or pull. Therefore, we replaced the unstructured code with a simple, textual *processing code* encapsulating the same information. Element classes specify their processing code in their class definitions, using a function like this:

```
const char *processing() const { return "a/ah"; }
```

("`a/ah`" says that the input port and the first output port may be used as either push or pull, but the second output port is always push.) Click calls this (virtual) function to fetch the processing code, while tool support scripts search the source for definitions of `processing` and extract the corresponding codes. Other functions used in the same way include `flow_code` and `class_name`.

This approach maintains a common understanding between tools and Click, since they use the same specification. It also limits the implementation's flexibility. An element has exactly one processing code, for example; it cannot change its code depending on its configuration. We have not found this onerous.

Some specifications pose a more difficult problem than processing codes. The *click-align* tool, for example (Section 7.1), must know how particular elements will change the data alignment of passing packets. We have not figured out how to specify this information textually. Instead, *click-align* contains explicit code for the more commonly relevant element classes. This solution is unsatisfactory. Alignment specifications can get out of date with the element code itself, for example. At least the specifications could be embedded in the element code as comments.

Despite this, we have found external specifications like processing codes and alignment specifications extremely convenient in practice. External specifications can be shared between optimizers or targeted at individual ones. Compared with general approaches like Esterel or Nuprl that analyze source code directly, external specifications are more limited, but also easier to write and maintain. Furthermore, our optimizers seem more efficient and easier to write and use than general theorem provers.

5.4 Analogies

We think of Click optimizers as compiler optimization passes working on a high-level machine language whose "instructions" are element classes. Most of the optimizers correspond to well-known compiler or programming language optimization techniques: *click-fastclassifier* to dynamic code generation, *click-devirtualize* to static class analysis, *click-xform* to instruction selection or peephole optimization, *click-undead* to dead code elimination. This useful analogy inspired the construction of our first optimizer.

Like compiler optimization passes, or Unix filters, optimizers are easy to combine in different orders.

6 Optimization Tools

We have written three optimizers besides *click-fastclassifier*: *click-devirtualize*, *click-xform*, and *click-undead*. Section 8 evaluates their effectiveness on a simple IP router configuration.

6.1 *Click-devirtualize*

The *click-devirtualize* tool addresses virtual function call overhead by changing packet-transfer virtual function calls into conventional function calls. Users write element classes not knowing the context in which they will be used. Assume instead that every *RED* element was immediately followed by a *Queue*. Then the virtual function call by which *RED* transfers packets downstream could be replaced with a conventional function call to *Queue*'s packet handling function. This transformation obviously limits the flexibility of the *RED* element, and is therefore inappropriate for hand implementation. *Click-devirtualize* simply runs the optimization automatically.

Click-devirtualize reads a router configuration, then reads and partially parses the C++ source code for each element class used in that configuration. It generates new C++ element classes, roughly one per element, that replace each virtual function call for packet transfer with the correct direct function call. For example, consider an element whose first output port connects to the first input port of a *Counter* element. Then *click-devirtualize* would transform 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);
```

into code like this:

```
Counter *next = (Counter *)output(0).element();
// call is resolved at compile time
next->Counter::push(0, packet_ptr);
```

Note that '`output(0).port()`' was changed to '`0`', the actual port number. *Click-devirtualize* inlines several other method calls as well.

While *click-devirtualize* can generate a new element class for every element, it usually does not. For exam-

[3] Push and pull are different ways to pass packets between elements. See [11] for more information.

ple, *Discard* elements are dead ends—they do not transfer packets elsewhere—so all *Discard* elements can share code.[4] Therefore, two elements can share code if they have the same class (*Counter*, say) and each connect to a *Discard*. The two *Discard*s share code, so the packet-transfer virtual function calls in the *Counter*s resolve to the same function (namely, `Discard::push`). Similarly, two elements with the same class, each connected to one of the two *Counter*s, can 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. (Again, push and pull are different mechanisms for packet transfer.)
4. There exists a pull input port, or a push output port, where the elements connected to that port cannot share code, or the two connections terminate at different port numbers.

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

Since code expansion may make complete devirtualization impractical, *click-devirtualize* can be told that certain elements should not be devirtualized. *Click-devirtualize* should be the last optimizer applied in any chain, since it cements the order of elements in the configuration graph.

6.2 *Click-xform*

Click-xform reads a router configuration and an arbitrary collection of pattern and replacement subgraphs. It checks the configuration for occurrences of each pattern and replaces each occurrence with the corresponding replacement. When there are no more occurrences of any pattern, it emits the transformed configuration. Pattern and replacement subgraphs are router configuration fragments written as compound elements in the Click language. A pattern matches a subset of the configuration graph if the subset contains corresponding elements that are connected the same way. Furthermore, connections into or out of the subset must occur only in places allowed by the pattern. Element configuration strings must also match, except that patterns may contain wildcards that stand for any configuration argument.

Although *click-xform* is a general-purpose configuration transformer, we designed it for a more specific task:

[4]Strictly speaking, only *Discard*s with push input ports can share code.

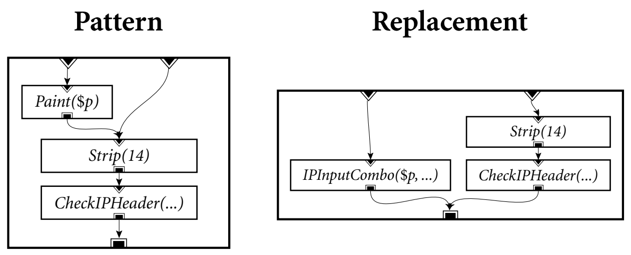

FIGURE 4—A pattern–replacement pair suitable for *click-xform*.

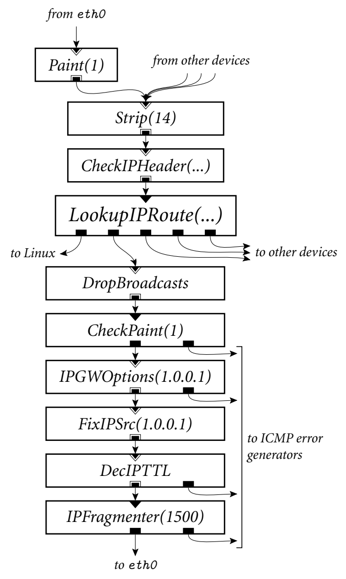

FIGURE 5—A portion of the IP router configuration.

replacing collections of general-purpose elements with optimized combination elements. This optimization both lowers virtual function costs by reducing the number of elements in a forwarding path, and reduces the overhead of general-purpose code.

We discourage Click programmers from using these combination elements directly, since they are relatively inflexible and have complex specifications. Instead, combination element programmers should write *click-xform* patterns that replace general-purpose element collections with the corresponding combination elements. Router designers use general-purpose elements, keeping their configurations easy to read and modify, but pass their configurations through *click-xform* before installation. Using the programmer-supplied patterns, *click-xform* will automatically change the configuration to use combination elements wherever that makes sense. This process somewhat resembles instruction selection or peephole optimization.

Figure 4 shows an example pattern–replacement pair. The *IPInputCombo* element combines the functions of a *Paint–Strip–CheckIPHeader* sequence. Applying that pattern–replacement pair, plus two others, can reduce the

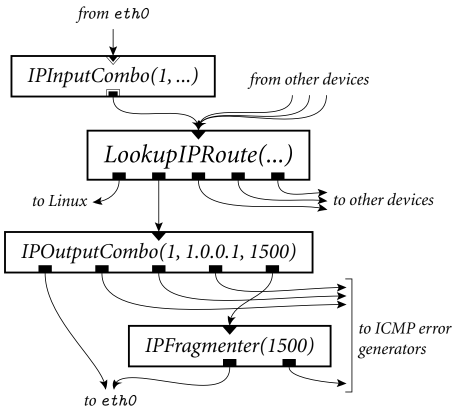

FIGURE 6—A router fragment equivalent to Figure 5 using faster "combo" elements.

number of elements on an IP forwarding path from ten to three; see Figure 5 and Figure 6.

Searching a graph for an occurrence of a pattern is a variant of subgraph isomorphism, a well-known NP-complete problem. *Click-xform* implements Ullman's subgraph isomorphism algorithm [17], which works well for the patterns and configurations seen in practice. For example, *click-xform* takes about one minute to run several hundred replacements on a router graph with thousands of elements, and much less time for normal-sized routers.

Click-xform, and the other optimizers, compile away compound element abstractions before analyzing router configurations. This gives the optimizers a further advantage over manual optimization. Users would have to remove compound element abstractions by hand, complicating the configuration, to expose all optimization opportunities.

6.3 *Click-undead*

The *click-undead* optimizer performs the equivalent of dead code elimination on router configurations. For example, *click-undead* removes *StaticSwitch* elements and their unused branches; *StaticSwitch* routes all packets along one of several output paths. Generally, *click-undead* is effective only in the presence of compound element abstractions, which are the most likely source of dead code in Click configurations. For example, a compound element might use *StaticSwitch* to route packets along one of several possible paths depending on some configuration argument. We don't discuss *click-undead* in the evaluation section, since none of the elements in our IP router are dead code.

7 Other Tools

Our success with optimizers encouraged us to build tools that address other Click architectural issues, or that play with novel configuration analyses. The most useful of these, *click-align*, allows Click to run on non-x86 architectures without complicating the packet data model. *Click-combine* and *click-uncombine* construct configurations that represent the processing of multiple routers in a network; this enables unusual router optimizations. Tools not described further include *click-check*, which checks configurations for errors; *click-flatten*, which compiles away compound element abstractions; *click-mkmindriver*, which creates a minimum Click containing only the elements needed for a given configuration; and *click-pretty*, which pretty-prints configuration files as HTML.

7.1 *Click-align*

Click packet data is stored in a flat array of bytes, but for speed and convenience, many elements load from this array a machine word at a time. On the Intel x86 architecture, these loads need not be aligned on word boundaries; unaligned accesses are legal, and in our benchmarks, just as fast as aligned accesses.[5] On architectures such as ARM, however, unaligned accesses crash the machine.

Possible solutions include using special instructions to load words from packet data, checking packet data alignment at the beginning of every relevant element, and enforcing known alignments when packets are created or read from devices. (Linux uses a combination of special instructions and well-known alignments set by devices.) Unfortunately, these possibilities would either slow down the x86 case, complicate Click's packet data model, or prevent complex configurations.

Click instead asks the user to ensure that elements receive packets with the correct alignment. This requires two new element classes: *Align* aligns packet data on a given boundary using a data copy, and *AlignmentInfo* informs elements what packet data alignments they can expect.

Inserting these elements by hand would be tedious and error-prone, of course. The *click-align* language tool therefore automates the process. It reads a configuration on standard input, calculates the configuration's expected and required packet data alignments, and inserts *Align* elements wherever the expected and required alignments are in conflict. The algorithm for calculating alignments was patterned after data-flow analyses in the compiler literature; several heuristics minimize the number of inserted *Align*s. Finally, *click-align* removes redundant *Align*s and adds an *AlignmentInfo* element.

As mentioned above, the specifications for elements' expected and required alignments are built in to the *click-*

[5]An unaligned load can cause two d-cache misses, but this is not a problem in practice, since packet data is always in the cache.

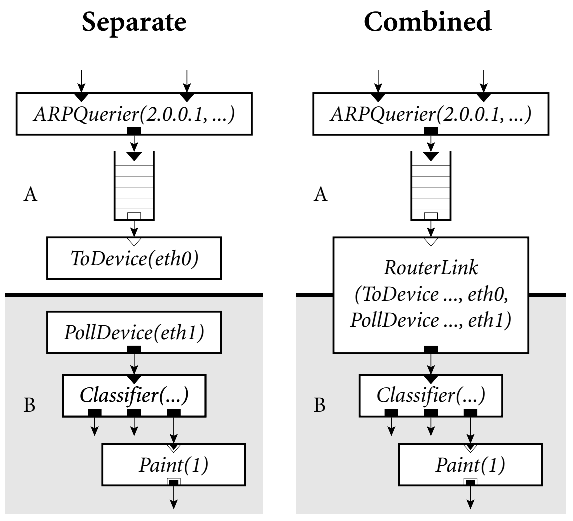

FIGURE 7—Portions of two routers, and the corresponding combined configuration generated by *click-combine*.

align tool. For example, the tool knows that *CheckIPHeader* expects input packets to be word-aligned.

7.2 *Click-combine* and *click-uncombine*

Network architects care about how individual routers behave, and how multiple routers on their network might interact. To facilitate simultaneous analysis of a multiple-router network, the *click-combine* tool creates "router" configurations that encapsulate the behavior of, and connections between, multiple routers. Such combined configurations may be analyzed and manipulated, either to ensure properties of the network—for instance, that the frame formats at either end of each link are compatible—or to optimize away redundant computation performed by more than one router. The *click-uncombine* tool separates a combined configuration into its component router parts. These tools are speculative and experimental; we present them to demonstrate how far the optimizer idea can be pushed.

Operationally, *click-combine* encapsulates router configurations inside compound elements, then links those compound elements together via *RouterLink* elements. The user supplies several router configurations and inter-configuration links, such as "router A's *ToDevice(eth0)* element connects to router B's *PollDevice(eth1)* element"; *click-combine* emits the combined configuration. Figure 7 demonstrates this in action.

The best use for combined configurations is probably to check router networks for properties like loop freedom. Optimizations are also possible, though dangerous. For example, the combined configuration in Figure 7 exposes the point-to-point nature of the link between routers A and B. There is therefore no need for an ARP mechanism on that link (unless and until the configuration changes). We wrote a set of *click-xform* patterns and replacements that remove ARP from these kinds of links. A tool chain like `click-combine ... | click-xform ... | click-uncombine ...` combines router configurations as appropriate to the state of the network, removes ARP where possible, then extracts the possibly-modified router configuration from the combination. We report performance results for this optimization below.

8 EVALUATION

We now turn to performance measurements for unoptimized and optimized Click IP routers. Our reference configuration is the Click IP router shown in Figure 1. We first present performance in terms of CPU cycles required to process a packet, then in terms of overall packet forwarding rates. The optimizers described above can reduce the CPU time spent per packet in the Click forwarding path by 34%, and increase its peak forwarding rate by 89,000 packets per second. We present a detailed analysis of the limiting factors for a Click router's performance, and demonstrate that our optimizations allow Click to forward close to the maximum allowed by our hardware. Finally, we analyze how the effectiveness of the optimizations changes with newer hardware.

8.1 Testing configuration

Our testing configuration consists of nine Intel PCs running a modified version of Linux 2.2.16. One PC is the router host, four are source hosts, and four are destination hosts. The router host has eight 100 Mbit/s Ethernet controllers connected, by point-to-point links, to the source and destination hosts. During a test, each source generates an even flow of UDP packets addressed to a corresponding destination; the router is expected to get them there.

The router host has a 700 MHz Intel Pentium III CPU and an Intel L440GX+ motherboard. Its eight DEC 21140 Tulip 100 Mbit/s PCI Ethernet controllers [2] are on multi-port cards split across the motherboard's two independent PCI buses. The Pentium III has a 16 KB level-1 instruction cache, a 16 KB level-1 data cache, and a 256 KB unified level-2 cache. The source and destination hosts have 733 MHz Pentium III CPUs and 200 MHz Pentium Pro CPUs, respectively. Each host has one DEC 21140 Ethernet controller. The source-to-router and router-to-destination links are point-to-point full-duplex 100 Mbit/s Ethernet.

The source hosts generate UDP 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. Each 64-byte UDP packet includes

Task	CPU time (ns/packet)
Receiving device interactions	701
Click forwarding path	1657
Transmitting device interactions	547
Total	**2905**

FIGURE 8—CPU cost breakdown for an unoptimized Click IP router.

Ethernet, IP, and UDP headers as well as 14 bytes of data and the 4-byte Ethernet CRC. When the 64-bit preamble and 96-bit inter-frame gap are added, a 100 Mbit/s Ethernet link can carry up to 148,800 such packets per second.

Since we are interested in the effects of our optimizations, we compare solely against unoptimized Click. This also avoids unfair comparisons against non-polling systems.

8.2 CPU time

The most pertinent measure of the cost of a Click router configuration is the CPU time required to forward a packet. Each packet incurs additional costs, such as PCI contention and bandwidth and network device processing time, but those costs are similar for any configuration using the same devices. Our optimizations focus on the forwarding path, not device interactions, so CPU time is the correct cost metric.

Figure 8 breaks down the CPU time required to forward a packet on an unoptimized Click router. Costs are measured in nanoseconds by accumulating Pentium III cycle counters [1] before and after each block of code, and dividing the totals over a 10-second run by the total number of packets forwarded. We break CPU cost into three categories: receiving device interactions, such as pulling a packet from the receive DMA ring; Click's forwarding path; and transmitting device interactions, such as enqueuing a packet onto the Tulip's transmit DMA ring. These measurements are different than the true values, since using Pentium performance counters has significant cost, and since the overhead due to Click's task queue, and due to other processes on the system, is not included. However, the total cost of 2905 ns measured with performance counters implies a maximum forwarding rate of about 344,000 packets per second, consistent with the observed maximum forwarding rate of 357,000 packets per second. Although device interactions are expensive, Click's forwarding path takes the majority of time.

Our language optimizations reduce the CPU time spent by the Click forwarding path by up to 36%, and the total

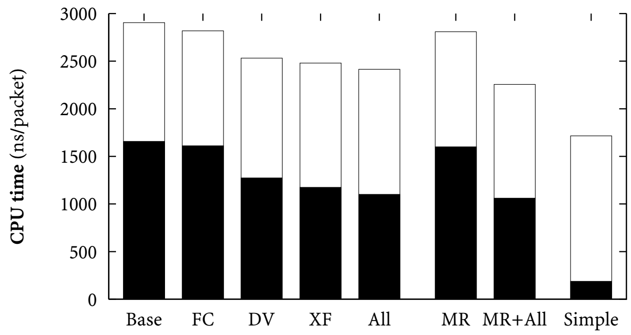

FIGURE 9—Effect of language optimizations on CPU time cost. Black bars show Click forwarding path cost, white bars show total cost including device drivers. "Base" is an unoptimized IP router. "FC" is Base plus *click-fastclassifier*, "DV" is Base plus *click-devirtualize*, and "XF" is Base plus combination elements installed with *click-xform*. "All" denotes all three optimizations applied together. "MR" denotes ARP elimination, our sample multiple-router optimization; "MR+All" denotes all four optimizations applied together. Finally, "Simple" is the minimal configuration, consisting only of device handling and a single packet queue.

CPU time per packet by up to 22%. Figure 9 illustrates the effects of the language optimizations individually and in combination. The leftmost column represents the cost of pushing a packet through an unoptimized Click IP router. Applying all the optimizations described in Section 6—*click-xform* with the combination elements, *click-fastclassifier*, and *click-devirtualize*—reduces the cost of the Click forwarding path by 34%, from 1657 ns to 1101 ns. Adding ARP elimination, the optimization enabled by Section 7.2's multiple-router configurations, reduces the cost further to 1061 ns.

Of the three router-local optimizations, *click-xform* is the most effective. *Click-devirtualize* provides a similar performance improvement, but its optimization opportunities overlap with those of *click-xform*. As a result, applying both of these optimizations is not much more useful than applying either one alone. *Click-fastclassifier* is not that effective here, reducing CPU time by just 3%. This is because the single classifier in the configuration has a very simple decision tree.

Forwarding a packet through Click incurs just four 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 the device marks it as sent. Each cache miss incurs a fetch from main memory, which takes about 112 ns. Click runs without incurring any other data or instruction cache misses. With all three optimizers turned on, just 988 instructions are retired during the forwarding of a packet. This implies that significantly more complex Click configurations could

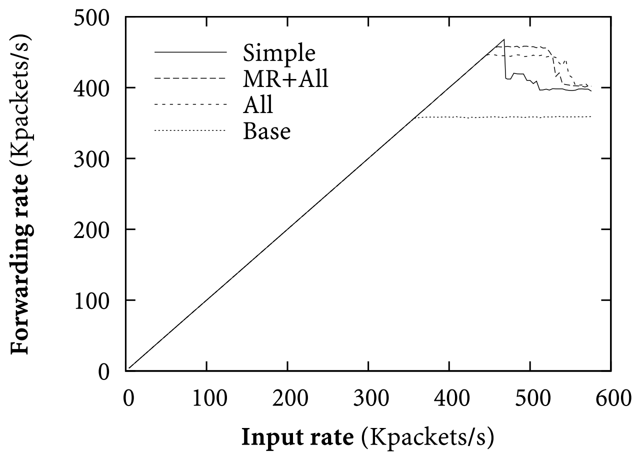

FIGURE 10—Effect of language optimizations on forwarding rate for 64-byte packets. An ideal router that forwarded every packet would appear as a straight line $y = x$.

be supported without exhausting the Pentium III's 16 KB L1 instruction cache.

8.3 Forwarding rates

Figure 10 graphs forwarding rate versus input rate for variously optimized IP routers. The CPU time savings enabled by language optimizations translate into higher peak forwarding rates. The maximum loss-free forwarding rate (MLFFR) for an unoptimized IP router is 357,000 64-byte packets per second. (Minimum-size packets stress the CPU more than larger packets, and we are mostly concerned with CPU time here.) The optimized IP router ("All") has a significantly higher MLFFR, 446,000 packets per second; "MR+All" raises that further, to 457,000 packets per second.

Unlike the unoptimized IP router, the optimized configurations are unable to sustain their peak forwarding rates, dropping to approximately 400,000 packets per second as the input rate increases. Interestingly, the same pattern applies to "Simple", the simplest possible Click configuration. Furthermore, the MLFFR of "Simple" is not much higher than that of the optimized IP configurations, although its total CPU time cost is 25% lower. These factors suggest that the optimized IP routers and "Simple" are no longer solely limited by CPU time cost.

8.4 PCI limitations

To investigate the factor limiting forwarding performance for each router configuration, we examined where our system dropped packets. Code inspection, Tulip documentation, and experiments showed that each packet has one of four possible outcomes. It may be dropped on the receiving Tulip card because the Tulip's internal FIFO is full ("FIFO overflow"), or because the Tulip was not able to fetch a ready DMA descriptor after two tries ("missed frame"); it may be dropped at the Click *Queue* when packets are arriving faster than they can be sent ("*Queue* drop"); and if it survives those obstacles, it is sent ("packet sent"). If a packet is to be dropped, it is best to drop it at the receiving card's FIFO, as this early drop point avoids wasting resources—in particular, it avoids all PCI bus and memory operations.

Figure 11 shows the outcome rates for varying input rates and for three router configurations, "Simple", "Base", and "MR+All". The solid line in each graph corresponds to the forwarding rate. The other lines cumulatively add other drop outcomes to the forwarding rate. The sum of all outcomes is the input rate—that is, the straight line $y = x$.

The baseline IP router configuration is clearly CPU-limited. All of its input packets are either forwarded or dropped as missed frames. Missed frame drops occur when a Tulip card finds that the next DMA descriptor is not free twice in a row. This indicates that the CPU is emptying and refilling the relevant receive DMA ring slower than the Tulip card is filling it—that is, the CPU is overloaded.

The "Simple" configuration is not CPU-limited. None of the packets dropped by "Simple" are missed frames; they are either FIFO overflows or *Queue* drops. Both of these outcomes indicate that the PCI bus or memory system is overloaded. FIFO overflows occur when a receiving Tulip card gets packets faster than it can request DMA descriptors and write packets to memory. *Queue* drops occur when packets are added to a *Queue* faster than *ToDevice* removes them. Instrumenting the *ToDevice*s revealed that these elements were often idle—that is, they chose not to pull packets—because their devices' transmit DMA queues were full. Thus, the CPU wanted to send packets faster than the transmitting Tulip cards could process them. These tasks—reading packets from memory, writing packets to memory, and requesting DMA descriptors—are limited not by the CPU, but by the PCI bus or the memory system. Our analysis is not detailed enough to determine the bottleneck resource precisely.

The "All" and "MR+All" plots in Figure 10 are probably determined by the following factors. They initially flatten out because the CPU isn't fast enough; in this zone, all dropped packets are due to missed frames. As the input rate rises above that point, more and more PCI bandwidth is consumed by the receiving Tulips checking (unsuccessfully) for a free DMA descriptor for each incoming packet (these are "missed frames"). Above a certain input rate the failed descriptor checks saturate the PCI bus. Each failed descriptor check uses up PCI bandwidth that another Tulip

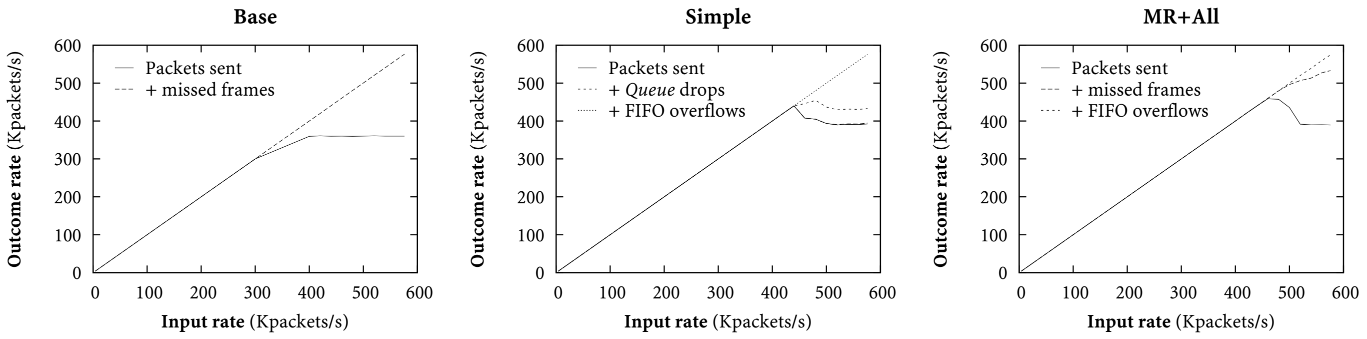

FIGURE 11—Cumulative outcome rates as a function of input rate for three Click router configurations.

could have used to receive or send packet data. Thus, once the PCI bus is saturated, increases in input rate cause decreases in forwarding rate. For a while these failed descriptor checks do not result in Tulip FIFO overflows, since the Tulip flushes the failed frame when the descriptor check fails. As the input rate increases, however, packets arrive faster than the Tulip checks for descriptors; at that point the Tulip discards the excess packets (as FIFO overflows) without any impact on the PCI bus. Thus input rates above about 550,000 packets per second do not cause decreases in forwarding rate.

8.5 Hardware evolution

In order to illustrate interactions between the optimizations and changing hardware capabilities, we measured the performance of Click on three additional platforms. Platform P1 consists of an 800 MHz Pentium III with 32-bit/33 MHz PCI, P2 is the same machine with 64-bit/66-MHz PCI, and P3 is a 1.6 GHz AMD Athlon MP with 64-bit/66 MHz PCI. P0 is the 700 MHz platform discussed in the previous sections. In all platforms but P0, the Ethernet hardware is the Intel Pro/1000 F gigabit Ethernet card. This card has a 64-bit/66 MHz PCI interface, but will operate at 32-bit/33 MHz when used in the slower bus. The router has two interfaces, each with a full-duplex link to a host. The two hosts both generate and count packets; each of them generates half of any given load. Each host is capable of generating a million 64-byte packets per second.

Figure 12 shows how much the optimizations improve performance on each platform, and Figure 13 shows the forwarding rates for the three platforms. P1 is almost identical to the router hardware discussed in the previous section; P1 probably performs somewhat less well because the Pro/1000 card requires the CPU to use programmed I/O instructions for each batch of packets sent or received, while the Tulip does not. The graph for P2 demonstrates that performance for "Simple" was limited by the PCI bus in P1, while performance for other Click configurations

	MLFFR (packets/s)		
Platform	All	Base	**Ratio**
P0	446,000	357,000	1.25
P1	430,000	350,000	1.23
P2	450,000	330,000	1.36
P3	740,000	640,000	1.16

FIGURE 12—Effect of "All" optimizations on MLFFR for each hardware platform.

was not. P3's CPU is about twice as fast as that of P2; this lets P3 forward about 1.9 times as fast as P2 for "Base," and about 1.6 times as fast for "All." Our optimizations seem effective on all platforms.

9 CONCLUSION

This paper has described optimization tools based on compiler optimizations and programming language techniques for Click, a component-based networking system. Three of our optimizations collectively boost the performance of a Click IP router to close to the maximum performance allowed by our hardware. The optimizers are easy to use, pleasant to write, improve system performance, and extend system functionality; the *click-align* tool, for example, allows Click to support non-x86 architectures. We believe this technique, compiler-like tools for configuration transformation, is valuable enough to influence the way systems are built.

Complete source code for Click and the optimization tools is available on-line at `http://www.pdos.lcs.mit.edu/click/`.

ACKNOWLEDGMENTS

We are grateful to Frans Kaashoek and to Mazu Networks, Inc. for their support of this work, to the anonymous reviewers and the Utah Flux group for helpful comments, to David Karger for pointers into the subgraph isomorphism literature, and to Click users for bug reports and

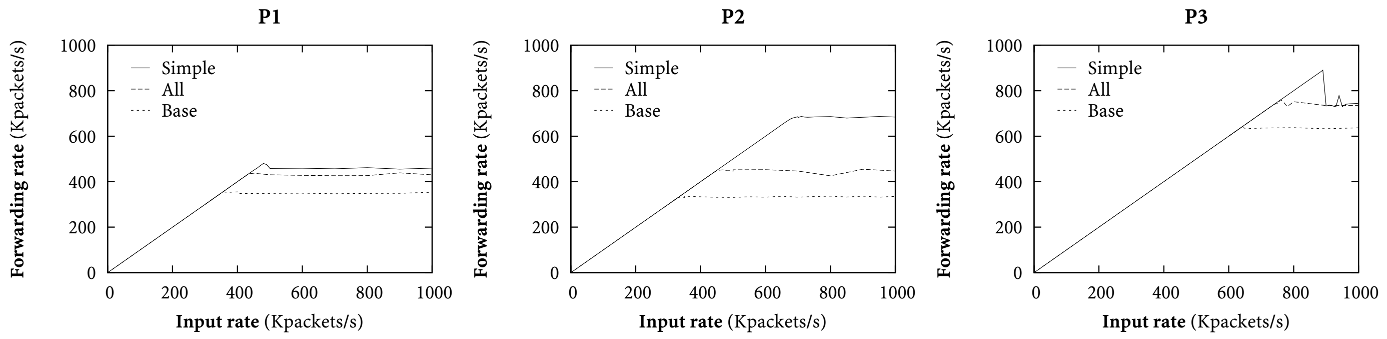

FIGURE 13—Forwarding rates for platforms **P1**, **P2**, and **P3**.

patches.

Much of this research took place while all three authors were at MIT LCS. It was supported there 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.

REFERENCES

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

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

[3] 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.

[4] Claude Castelluccia, Walid Dabbous, and Sean O'Malley. Generating efficient protocol code from an abstract specification. In *Proc. ACM SIGCOMM Conference (SIGCOMM '96)*, pages 60–71, August 1996.

[5] Click Project. The Click modular router (Web site). `http://www.pdos.lcs.mit.edu/click/`.

[6] Julian Elischer and Archie Cobbs. Netgraph. `ftp://ftp.whistle.com/pub/archie/netgraph/index.html`.

[7] 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.

[8] 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.

[9] 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. 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–92, December 1999.

[10] 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.

[11] Robert Morris, Eddie Kohler, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. Computer Systems*, 18(3):263–297, August 2000.

[12] 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.

[13] 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.

[14] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. 4th ACM Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 347–360, October 2000.

[15] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

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

[17] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. Assoc. Comput. Mach.*, 23:31–42, 1976.

[18] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls, Second Edition*. O'Reilly and Associates, Sebastopol, California, 2000.