We evaluate Streamline in terms of performance of low-level primitives and high-level applications (7.2), reconfigurability of real host-based and embedded networking tasks (7.3) and optimization speed and quality (7.4).
Evaluation

<table>
<thead>
<tr>
<th>No.</th>
<th>Claim</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>optimization execution time</td>
<td>&lt; 250 ms</td>
</tr>
<tr>
<td>2.</td>
<td>higher throughput with cache-sized buffers</td>
<td>3x</td>
</tr>
<tr>
<td>3.</td>
<td>higher throughput with fragmentation-avoiding buffers</td>
<td>1.2x</td>
</tr>
<tr>
<td>4.</td>
<td>higher throughput of pipes with large shared buffers</td>
<td>4x</td>
</tr>
<tr>
<td>5.</td>
<td>higher throughput of copy with splicing</td>
<td>3-10x</td>
</tr>
<tr>
<td>6.</td>
<td>higher throughput of pipeline with signal avoidance</td>
<td>1.5-100x</td>
</tr>
<tr>
<td>7.</td>
<td>higher throughput of pipeline with signal batching</td>
<td>3x</td>
</tr>
<tr>
<td>8.</td>
<td>higher throughput of legacy I/O applications:</td>
<td></td>
</tr>
<tr>
<td>a.</td>
<td>the Bind DNS server</td>
<td>1.4x</td>
</tr>
<tr>
<td>b.</td>
<td>the Mplayer video player</td>
<td>3x</td>
</tr>
<tr>
<td>c.</td>
<td>the Tcpdump traffic monitor</td>
<td>3-10x</td>
</tr>
<tr>
<td>9.</td>
<td>efficient use of kernel and device offload</td>
<td>device dep.</td>
</tr>
</tbody>
</table>

Table 7.1: List of Quantitative Claims Made in Introduction

7.1 Overview

This chapter verifies the quantitative claims made in the introduction of this thesis. Specifically, it evaluates the items listed in Table 7.1. Of these, points 2, 3, 5, 6 and 7 have already been evaluated in the technical chapters. Table 7.2 refers to the experiments. We split the evaluation of the remainder in three themes. We evaluate efficiency and effectiveness of the solver algorithm in Section 7.4. This includes point 1. We compare the performance of system interfaces and applications on standard Linux and Streamline head-to-head in Section 7.2, which covers points 4 and 8. Section 7.3 presents a set of native Streamline applications to confirm point 9. It shows two host-based applications that benefit from kernel offload and four embedded applications that scale to network linerate by offloading to special purpose processors.

7.2 Performance

It is harder to demonstrate utility of a new I/O architecture by improving existing application throughput than by handcrafting programs to make use of all unique features. Besides evaluating individual contributions to show their merit in vitro, pipes to show real Unix primitive improvement, and native applications to demonstrate novel features, we measure performance of several mature legacy applications, to demonstrate that Streamline is useful also for applications not written for it. We evaluate end-to-end performance at two levels. The first, Unix primitives, displays the raw gains at the buffer inter-
face level. The second, application benchmarks, shows achievable gains for applications that use legacy interfaces.

We compare a Streamline-enabled version of Linux 2.6.24.2 head-to-head with a stock version in terms of application throughput. All tests were run on an HP 6710b machine with Intel Core 2 Duo T7300 processor, 4MB L2 cache and 2 GB RAM running in 32-bit mode. We ran the experiments on a single core to minimize scheduler influence and show the full cost of task switching.

### 7.2.1 Unix Pipes

Figure 7.1 shows throughput of straightforward copying (a `write` followed by a `read`) through a Unix pipe for three IPC implementations: a standard Linux pipe, a producer/consumer pair of threads that directly access shared memory and streamline buffers of varying size. In this test, we do not use Streamline’s `peek` optimization and thus copy the same amount of data as the other applications. Any performance improvement comes from a reduction in context switching. The threaded application shows an upper bound on achievable performance, because it requires no switching to kernelspace at all and it implements a multi-packet ring. Similar to Streamline rings, its throughput is dependent on buffer size, but we only show the best case here for clarity (1MB). That configuration outperforms Linux’s implementation of Unix pipes by a factor 5 for large blocks and 12 for minimal blocks. In between are 4 differently sized Streamline Slow Reader Preference DBufs. We see that the fastest implementation is neither the largest (64MB), nor the smallest (64KB), but an intermediate (1MB). This outperforms Linux by a factor 4 for large and 9 for small packets and is therefore only between 20 and 33% slower than the optimal case.

The actual factor of throughput increase depends on physical cache size, producer-consumer distance and whether the application buffer is cached,
but the ratios are static; we previously observed similar results on different hardware [dBB08a]. In this earlier execution of the same experiment on older hardware with a smaller cache and uncached application buffer we observed increases of 2.5x and 2x for large blocks. In those tests, the optimal buffer size was smaller, but again comparable to local the L2 cache (256KB). As cache sizes keep increasing, so do obtainable throughput increases.

Figure 7.2 explains why the highest throughput is achieved with a medium-sized buffer. Initially, performance grows with the buffer as the number of necessary context switches drops when calls are less likely to block. ulti-
mately, however, page-faults affect performance as the TLB and d-Cache start suffering capacity misses. These are more expensive than switches, therefore maximum throughput is obtained when the working-set just fits in the L2 cache.

7.2.2 Legacy Applications

The micro benchmarks demonstrate that significant I/O overhead can be saved by optimizing buffer parameters and employing copy avoidance. We now investigate to what extent these savings translate to real application domains. For this purpose we ran the applications introduced in Section 2.1 on top of both the Linux and Streamline versions of sockets and libpcap: bind identifies per-call overhead, mplayer per-byte overhead and tcpdump parallelization cost. The presented results are the median of 51 2-second CPU utilization measurements.

**DNS Server**  The Bind named daemon replies to DNS requests. We ran a version 9.4.1 daemon in non-recursive mode and sent it a steady stream of 10 thousand requests per second. DNS messages are small: requests were below 55B and replies below 130, including IP headers. For this reason, application processing easily dominates total overhead. Figure 7.3 shows the result of sending 10 thousand requests per second to our daemon. We show total CPU utilization for Linux and Streamline, whereby for Streamline we vary both buffer size and event batch threshold (we plot Linux for multiple values
Evaluation

for clarity only, it does not batch). We present the median of 103 measurements, with upper and lower quartiles within 20% of the presented results.

The figure shows that the daemon performs between 1.1x and 1.5x as well on top of Streamline as on top of Linux. As expected, this factor grows with the amount of batching (throttling). Buffer size affects performance less. This is a result of NIC hardware requirements that restrict buffer scalability freedom: the reception DBuf must be at least 1MB and the transmission queue at most 1MB.

**Video Client** To demonstrate savings for memory-bound applications we now present results obtained with streaming a high definition (SXGA), high data rate (100Mbit) MPEG4 stream to a popular video client: mplayer 1.0 RC2. We disabled software decoding of the compressed stream, as the (software) decoder would render the application computation-bound. Our results thus compare to machines with hardware video decoders. Figure 7.4 summarizes the results obtained with 1MB reception and transmission buffers (again, chosen because of NIC requirements). Here, too, Streamline is more efficient than Linux, between 2- and 3-fold, depending on event threshold. For small batching factors Streamline is as fast as Linux or even slower. Cost drops linearly with context switch reduction until we reach a 2x speed-up at 256 blocks per call.

**Traffic Monitor** Figure 7.5 shows throughput of Tcpdump 3.9.8, a popular traffic analyzer. We compare standard tcpdump, which uses Linux Socket Filters, to a version that talks to our pcap interface. To investigate scalability with parallel data access, we capture a moderate datastream: 200 Mbit of full-
sized 1500B packets per second, generated with IPerf 2.0.2. The IPerf server requires 50% CPU time. With a single listener, Streamline uses up hardly any extra resources, while standard Linux requires 5% CPU time (10% of the application cost).

Savings decrease as we run applications in parallel. When capturing full frames (‘sl 1500B’) with 10 instances, Streamline causes a 13% CPU overhead, slightly above a single Linux instance, whereas Linux more than doubles overhead, drops packets and saturates the CPU. Whereas standard tcpdump overhead increases linearly with the number of packets, our version incurs no significant overhead for up to 9 applications. This wide disparity comes from the fact that Streamline coalesces most processing in the kernel and shares the same buffer with all clients. With more than 10 applications thrashing occurs. By inspecting the number of voluntary and forced context switches independently we learned that, although involuntary switching increases for each extra application, thrashing does not occur until the number of voluntary switches starts decreasing. This is a sign that applications do not get the chance to handle all data when it comes in, but need an extra time-slot and the beginning of a snowball effect: thrashing.

Although we expected standard tcpdump to be memory bound, Figure 7.5 shows that the minimal and maximal capture length versions have roughly the same overhead. Thus, tcpdump is not directly memory bound. Indeed, 100 Mbit of data may be copied tens of times before a modern memory bus is saturated. The real bottleneck is that tcpdump switches for each packet. Standard tcpdump switches 19 times as much as a single Streamline tcpdump instance (77000 vs 44000). Even for 10 parallel applications, our version switches only a fifth the amount of a single standard tcpdump (16000).

Running ten instances of tcpdump is not a common operation, but the results are representative for any system configuration where multiple appli-
Evaluation

cations access the same data, for instance for network intrusion detection and auditing or for group communication such as multiway pipes.

7.3 Example Applications

In legacy applications, Streamline can only optimize the operations behind the system interface. This section presents applications developed directly on top of Streamline. We first present three case studies of complex high throughput tasks and show how Streamline maps these onto heterogeneous processors (various types of Intel IXP network processors) and then review a handful of other applications in less detail. Contrary to the legacy applications, these examples have no direct competitors in existing software. Furthermore, the applications often run slower on the (slightly outdated) programmable peripherals than on modern CPUs. The purpose of these demonstrations is not to compare maximally obtainable throughput, but to show that Streamline can seamlessly incorporate heterogeneous resources when available. If a programmable NIC is available, offloading some processing will benefit the system by freeing up CPU resources, even if it cannot reach the same level of performance.

7.3.1 Embedded Applications

The various revisions of Streamline have also been used to program heterogeneous platforms. Manycore CPUs are expected to have a degree of performance asymmetry [LBKH07] and full heterogeneity [ABC+06a]. In embedded operation, this design has been tried before. Network processors (NPUs), for instance, are processors for high-throughput networking applications that combine slow, general-purpose cores for control operations with fast, restricted-use cores for data processing. Over the last five years, we have ported Streamline to three variants of the popular Intel IXP line of network processors. This section presents work involving groups of contributors. It first introduces the IXP design and its variants. Then, it shows four applications: a packet-level network monitor on first generation hardware (IXP1200), an application-level intrusion prevention system on second generation hardware (IXP2400), and a cryptographic token-based switch and programmable router both on the fastest hardware (IXDP2850).

Network Processors  The IXP is an asymmetric multicore processor built from a single ARM (StrongARM or xScale) CPU running Linux and a number of RISC processors known as µEngines, or µEs. These run no operating sys-
tem whatsoever. Instead, each supports a number of *threads* in hardware that have their own program counters and register sets. As a result of hardware multiplexing, context switches between threads take zero cycles. \( \mu \)Engines are not generic load-store machines. Instead, each has an “instruction store” memory region of limited size (from 1 to 32KB).

We have ported Streamline to three configurations: a Radisys ENP 2506 programmable network card based on the IXP1200 NPU, an IXP-2400 PCI-X plug-in board and a IXDP-2850 dual processor machine that is only accessi-

ble over the network. All have (between 2 and 10) 1-Gbit network ports. The IXP1200 has 6 \( \mu \)Engines with four hardware threads per core and a 1KB in-

struction store running at 233 MHz, the IXP2400 has 8 \( \mu \)Engines, eight threads per core, 4KB stores and a 600 MHz operating frequency, the IXP2800 has

16 \( \mu \)Engines, eight threads per core, 8KB stores and a 1.4 GHz frequency. The IXP2800 also embeds cryptographic logic units. Figure 7.6 presents the Radisys ENP 2506 network card as example configuration. For input, the board is equipped with two 1Gbps Ethernet ports (1). The card embeds an IXP1200 network processor with 8 MB of SRAM and 256 MB of SDRAM (2) and is plugged into a 1.2 GHz Pentium III over a 32/66 PCI bus (3).

**Software Stack** On the IXP1200 we implemented 2 spaces: one kernel space on the control CPU and one space spanning the \( \mu \)Es. The reason for not giving each each engine its own space is that the vendor-supplied code loader can only set or replace the code for all \( \mu \)Engines at once. The total system layout is similar to that of the VERA router [KP02]: this board is attached to a regular “host” computer system that has kernel and userlevel spaces. Like the VERA router [KP02] developers, we noticed that the control CPU is not
Evaluation

nearly powerful enough for datapath network processing. We therefore restrict the CPU to control plane operation. It talks to host streamline over a PCI-X channel and carries out instantiation and buffer allocation requests on behalf of the UEs. Communication with UEs is through shared-memory.

Because the IXP control processor runs Linux, most of the time porting Streamline to the IXPs was spent writing \( \mu \)Engine code loading wrappers and channels across the PCI-X bus. This last effort led to the replication buffer implementations introduced in Section 4.5.5. Implementation details are discussed in more detail elsewhere [NCdBB04, dBSvR +06]. Contrary to the IXP1200, the two newer systems have not been integrated with a host computer system. Their applications are meant to be isolated from a host.

Nic-Fix: High-Speed Network Monitoring

The need for affordable network monitors is growing, e.g., for security, traffic engineering, SLA monitoring, charging, and other purposes. NIC-FIX [NCdBB04], is an implementation of Streamline:FFPF [BdBC +04] on cheap Intel IXP1200 network processors.

Design System design is ‘bottom-up’ in that packets are handled at the lowest processing level as much as possible and few packets percolate to higher levels. Moreover, higher levels only take action when prompted to do so by the lower layers. This is a well-known approach, e.g., in router design [SKPG01]. NIC-FIX forms the lowest two levels of a four level Streamline hierarchy. It enables transparent offloading of packet processing to the network card when possible, reducing strain on the memory and peripheral buses when packets do not have to travel up to the host.

The basic structure of the programs running on the \( \mu \)Engines is as follows. For each \( \mu \)Engine, NIC-FIX provides boilerplate code in \( \mu \)Engine C, a subset of the C language. This boilerplate code contains a standard main() processing loop and a slot for the application-specific filter function. In principle, users may define the entire body of their filter function directly in \( \mu \)Engine C. However, in practice this is complex. That is why NIC-FIX also exports a higher-level approach, whereby filters are written in FPL (Section 6.5.1) which is precompiled to \( \mu \)Engine C and then IXP \( \mu \)Engine object code. This scheme closely resembles that of hardware plugins [JJH02]).

When a packet arrives at a network port its dedicated receiver \( \mu \)Engine copies the packet to the next available slot in the only DBuf. The remaining \( \mu \)Engines each execute a single filter. All four threads on a \( \mu \)Engine execute the same filter, but on different packets. If the packet matches the filter, the \( \mu \)Engine places a reference in its private IBuf. Otherwise, it is ignored.
Figure 7.7: NIC-FIX architecture: device memory is mapped to the host.

The DBuf resides in the NIC’s SDRAM and consists of static slots sufficiently large for any Ethernet frame. The IBufs are stored in smaller, faster SRAM. Additionally, each filter has a private storage space (MBuf) for data that persists between invocations. As Figure 7.7 shows, all three NIC-FIX buffers are memory mapped at the host processor, where a kernel pipeline continues processing. Buffers use FRP synchronization, so that no cross-space signaling is needed, but packets may go lost.

**Results**  An important constraint for monitors is the cycle budget. At 1 Gbps and 100 byte packets, the budget for four threads processing four different packets is almost 4000 cycles. As an indication of what this means, Table 7.3 shows the overhead of some operations. Note that these results include all boilerplate (e.g., transfers from memory into read registers and masking).

To evaluate NIC-FIX, we execute the three FPL2 filters shown in Figure 7.8 on various packet sizes and measure throughput. Only A is a ‘traditional’ fil-
Evaluation

ter. The other two gather information about traffic, either about the activity in every flow (assuming the hash is unique), or about the occurrence of a specific byte. Note that the hash function used in B utilizes dedicated hardware support. The results are shown in Figure 7.9. We implemented three variations of filter C. In C1 the loop does not iterate over the full packet, just over 35 bytes (creating constant overhead). In C2, we iterate over the full size, but each iteration reads a new quadword (8B) rather than a byte. C3 is Figure 7.8 without modifications.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[0] = HASH(26,12,256)</td>
<td>200 cycles</td>
</tr>
<tr>
<td>R[0] = PKT.B[0]</td>
<td>110 cycles</td>
</tr>
<tr>
<td>R[0] = PKT.W[0]</td>
<td>120 cycles</td>
</tr>
</tbody>
</table>

Table 7.3: Approximate overhead of some operators

Above a packet size of 500 bytes, NIC-FIX can process packets at line rate for A, B and C1. For smaller packets, filters C1 – 3 are not able to process the packets within the cycle budget. Up to roughly 165,000 PPS C1 still achieves throughput of well above 900 Mbps. Beyond that, the constant overhead cannot be sustained. C2 and C3 require more cycles for large packets and, hence, level off sooner. This suggests that simple prefilters that do not access every byte in the payload are to be preferred. This is fine, as the system was intended precisely for that purpose.

Just as for the C filters, throughput also drops for the simple filters A and B when processing smaller packets. However, these drops occur for a different reason, namely because the receiving µEngine simply cannot keep up.

SafeCard: Zero-Copy Intrusion Prevention

SafeCard [dBsvR+06] is a network intrusion prevention system for edge hosts that combines full packet payload scanning, application-layer protocol filtering (which requires traversing the entire protocol stack) and flow-based behavioral detection. Network intruders are increasingly capable of circumventing traditional Intrusion Detection Systems (IDS). Evasion and insertion techniques blind the IDS by spoofing the datastream, while polymorphism cloaks malicious code to slip past the filter engine [PN98, HPK01]. Besides hiding the attack, however, attackers employ another weapon to thwart network defense systems: raw speed [SSW02]. Less sophisticated attacks traveling over Gigabit links may be as difficult to stop as more complex attacks spreading more slowly. This leads to an interesting dilemma. On the one hand, systems that handle evasion and polymorphism are either too slow
Example Applications

(A) filter packets:

```cpp
IF (PKT.IP_PROTO == PROTO_UDP
   && PKT.IP_DEST == X && PKT.UDP_DPORT == Y)
   THEN RETURN 1;
ELSE RETURN 0;
FI
```

(B) count TCP flow activity:

```cpp
// count number of packets in every flow,
// by keeping counters in hash table (of size 1024)
IF (PKT.IP_PROTO == PROTO_TCP) THEN
   i = Hash(26,12,1024); // hash over TCP flow fields
   // increment the pkt counter at this position
   MEMORY[i]++;
FI
```

(C) count all occurrences of a character in a UDP packet:

```cpp
IF (PKT.IP_PROTO == PROTO_UDP ) THEN
   j = PKT.IP_PKT_SIZE; // saved pkt size in register
   k = 0
   FOR (i = 0; i < j; ++i)
      IF (PKT.B[i] == 65) THEN // look for char 'A'
         k++;
      // increment counter in register
   FI
   RETURN;
   MEMORY[0] = k; // save to shared memory
FI
```

Figure 7.8: Example Filters in Pseudocode (almost FPL2)

for in-line deployment (and are often host-based) or not sufficiently accurate (e.g. [JNS05]). On the other hand, fast in-line solutions are not able to detect and stop sophisticated attacks (e.g., [S. 04]). In this project we have built a network card that can be deployed in the datastream as an Intrusion Prevention System (IPS) at the edge of the network and that handles many forms of attack at Gigabit rates.

SafeCard provides a single IPS solution that considers many levels of abstraction in communication: packets, streams, higher-level protocol units, and aggregates (e.g., flow statistics). We selected state-of-the-art methods for the most challenging abstractions (streams and application data units) and demonstrate for the first time the feasibility of a full IPS on a network card containing advanced detection methods for all levels of abstraction in digital communication. To support in-depth analysis in higher-level protocol layers and achieve performance at Gigabit rates without swamping the host processor, we offload all tasks to a smart NIC. Additionally, physically removing safety measures from the user's machine has the advantage that they can-
Evaluation

Figure 7.9: Throughput for different NIC-FIX filters

![Throughput Graph](image)

Figure 7.10: SafeCard application pipeline

![Pipeline Diagram](image)

not be tampered with, which from a security viewpoint may be preferred by administrators. As in the case of NIC-FIX [BH05], SafeCard uses a slightly outdated and therefore cheap processor.

**Design** SafeCard combines 4 stages of defense: header-based filtering, payload inspection, flow-based statistical processing and application level protocol reconstruction. Supporting these methods are 3 additional stages: packet receive, packet transmit, and TCP stream reassembly. The full 7-stage pipeline is shown in Figure 7.10. Besides combining many levels of abstraction in our IPS, we also make contributions to individual components; we explain those shortly, but first give a brief overview of the entire pipeline. Stage 1 receives packets. In stage 2, we filter packets based on header fields (e.g., protocol, ports). Stage 3 is responsible for reconstructing and sanitizing packets. Stage 4 performs payload inspection. Stage 5 performs flow-based statistical processing. Stage 6 performs application level protocol reconstruction. Stage 7 transmits packets.

---

1 In terms of manufacturing costs, not necessarily in retail prices
TCP streams. In stage 4, we match the streams against Snort-like patterns. Unmatched traffic is inspected further in stage 5 by an innovative protocol-specific detection method capable of stopping polymorphic buffer overflow attacks. This method is superior to pattern-matching for the detection of exploits in known protocols. Against other types of malicious traffic, such as Trojans, it is ineffective, however. The two methods therefore complement each other: an indication of the strength of our sieve-based approach. Stage 6 further expands this idea by taking into account behavioral aspects of traffic. It generates alerts when it encounters anomalies in flow aggregates (e.g., unusual amounts of traffic) and subsequently drops the streams. Stage 7, at last, transmits the traffic if it is considered clean.

Each filter can drop what it perceives as malicious data. Only safe traffic reaches the last stage, where it is transmitted to its destination. The first practical stage, header-based filtering, is implemented using FPL \cite{CGXB07b}. Its functionality is run-of-the-mill and roughly equivalent to pcap.

We have developed a version of TCP reassembly that is both efficient and secure. Recreating a continuous stream of data from packets is expensive because in the common case it incurs a copy of the full payload. TCP is especially difficult to reconstruct, as it allows data to overlap and has many variants. These features have been frequently misused to evade IDSs. We reassemble in-place, i.e. in zero-copy fashion, and take a conservative view of traffic by dropping overlapping data. In terms of performance, we win by reducing memory-access costs. In the common case, when packets do not overlap and arrive in-order, our method removes the cost of copying payload completely. Instead, we incur a cost for bookkeeping of the start and length of each TCP segment. Due to the (growing) inequality between memory and CPU speed this cost is substantially smaller.

Pattern matching in SafeCard is implemented using regular expression matching, because static string matching (as for example provided by hardware CAMs) is too limited for detecting most intrusion attempts. Our engine, ruler \cite{HvRB07} (a contribution of Kees van Reeuwijk), is innovative in that it matches packets against the whole set of regular expressions in parallel, which allows it to sustain high traffic rates. Matched substrings can be accepted or rejected entirely, or rewritten to an altered output packet. Regular expression matching has been used in IDSs before, but only in special cases. Traditionally, the cost of matching scales linearly with the number of signatures. Because there are thousands of known signatures, scanning at high-speed is infeasible. Instead, string-matching algorithms whose runtime complexity remains constant regardless of patternset-size have to be used in the common case. Ruler can completely replace string-matching because it is a generalization of and therefore provably as efficient as Aho-Corasick (AC),
a popular constant-time pattern-matching algorithm that is employed for instance in Snort.

Unfortunately, future worms are expected to be increasingly polymorphic. Snort-like pattern matching is not suitable for stopping such attacks. Rather, we use protocol-specific detection methods requiring up to application layer communication. We protect hosts from buffer overflow attacks by tracing the address that causes the control flow diversion to a specific (higher-level) protocol field and capturing characteristics (such as the length of the field) that are subsequently used as an attack signature. This Prospector module is the work of Asia Slowinska [SB07].

Flow-based detection complements payload-scanning and (header-based) protocol reconstruction as the three detection vectors are orthogonal. As a demonstrator, SafeCard incorporates a filter that limits per-service traffic spikes to protect servers against flash mobs. The algorithm is admittedly naive, and serves mostly as a placeholder.

**Implementation** SafeCard is implemented as a single Streamline pipeline that can run in software, but (to protect at full Gigabit speed with minimal overhead) also runs as a combined control CPU/µEngine pipeline on an Intel IXP2400 smart NIC. Tomas Hruby developed most IXP specific code. Figure 7.11 shows this configuration. There, the computationally intensive TCP reassembly and pattern matching filters are offloaded to specialized processors. Application-layer and behavioral detection execute as software filters, because they are too complex to write in µEngine C and computationally less demanding. We offer a full network IPS implemented as a pipeline on a single network card. Each stage in the pipeline drops traffic that it perceives as malicious. Thus, the compound system works as a sieve, applying orthogonal detection vectors to maximize detection rate.

**Results** Each filter in the pipeline can prove to be the bottleneck. Some operations are obviously more expensive than others, such as pattern-matching, but this heuristic is of limited value when functions are implemented on different hardware resources. Indeed, as we discussed before, the Ruler engine can be scaled across multiple fast µEngines, while flow-detection must compete with Prospector for cycles on the XScale.

We can get an indication of the per-stage processing overhead by running the µEngines in single-thread mode and measuring the cycle count in isolation. Table 7.4 shows the cost in cycles per protocol data unit (PDU, e.g., IP packet, TCP segment) with minimal payload and the additional cost per byte of payload for each hardware accelerated filter. Figure 7.12 shows on the left
the maximal sustained rate of the filters as obtained from these numbers. At 600MHz, we can see that all filters can process common-case traffic at a Gigabit except Ruler. A single Ruler instance can process only 170 Mbit. The 5 combined engines thus top at 850Mbit, which we’ve plotted in the figure as 5x Ruler. Merging Reception and Transmission would give us the additional engine we need for full Gigabit processing.

A single threaded cycle count represents the worst-case scenario for overall throughput in TCP processing, because a single thread spends much of its time waiting for memory. Since multi-threading enables latency hiding throughput will improve dramatically.

Independent of maximal obtainable throughput is the question how indirect stream reassembly measures up to regular copy-based reassembly. For this reason we have compared them head-to-head. As we have no copy-based method available on the $\mu$Engines we ran this comparison in a host based Streamline function. The two functions share the majority of code, only differing in their actual data bookkeeping methods. Figure 7.12(right) shows that indirect reassembly easily outperforms copy-based reassembly. Only for the smallest packets can the computational overhead be seen.

<table>
<thead>
<tr>
<th>Description</th>
<th>PDU</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reception</td>
<td>313</td>
<td>1.5</td>
</tr>
<tr>
<td>TCP reassembly</td>
<td>1178</td>
<td>0</td>
</tr>
<tr>
<td>Ruler</td>
<td>628</td>
<td>26</td>
</tr>
<tr>
<td>Transmission</td>
<td>740</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.4: Single threaded cycle counts of individual filters
The third row in Table 7.4 shows the overhead in cycles of Ruler. As expected, costs scale linearly with the amount of data; the cost per PDU is negligible. The function is computation-bound: fetching 64 bytes from memory costs some 120 cycles, but processing these costs an order of magnitude more. For this reason multi-threading is turned off.

We have to benchmark Prospector on the XScale, because it has not been ported to the $\mu$Engines. Figure 7.13(left) compares throughput of Prospector to that of a payload-scanning function (we used Aho-Corasick). We show two versions of Prospector: the basic algorithm that needs to touch all header data, and an optimized version that skips past unimportant data (called Prospector+). The latter relies on HTTP requests being TCP segment-aligned. This is not in any specification, but we expect it is always the case in practice.

Each method processes 4 requests. These are from left to right in the figure: a benign HTTP GET request that is easily classified, a malicious GET request that must be scanned completely, and two POST requests of differing lengths. In the malicious GET case all bytes have to be touched. Since AC is faster here than both versions of Prospector we can see that under equal memory-overhead we suffer additional computational overhead.

However, all three other examples show that if you do not have to touch all bytes—the common case—protocol-deconstruction is more efficient than scanning. Looking at the right-most figure, the longest POST request, we can see that the gap quickly grows as the payload grows. The benign GET learns us additionally that skipping remaining headers when a classification has been made can result in a dramatic (here 2-fold) increase in worst-case performance. Note that none of these example requests carry a message body. This would also be skipped by Prospector, of course. Even without message bodies, performance is continuously above 18,000 requests per second, making the function viable for in-line protection of many common services.
Our final experiment evaluates the pipeline in hardware. We do not include results for the filters on the XScale again, because their throughput is not measurable in bitrate and we have already computed an upper bound. For this test we connected our board to a mirror image of communication between three computers. By using mirroring we were able to test peak throughput without interfering with the active TCP control flow. The traffic was generated using ab, a benchmarking tool for Apache. When ran against two servers at the same time our maximally obtainable rate was 940Mbits. The results are shown in Figure 7.13(right).

From the Figure we can see that with 6 \( \mu \)Engines we can process all traffic. To free up the sixth \( \mu \)Engine we had to remove the transmission unit temporarily. The presented numbers are worst-case estimations as a result of crude dropped traffic statistics. Actual performance could be up to 20\% higher.

**Cryptographic Token-based Switch**

Colleagues have used Streamline to demonstrate the practicality of using cryptographic tokens as a basis for switching. They embed cryptographic tokens in the link-layer frame or network-layer packet and make scheduling decisions based on this (unforgeable) token. Token generation and verification are computationally expensive tasks, especially when carried out hundreds of thousands of times per second. This application reuses the Streamline implementation of SafeCard. It required no significant architectural changes. With consent from the authors, we copy two short paragraphs from the main publication [CGXB07a], one that explains the design and another that summarizes results.
Figure 7.14: Token Based Switch using a dual IXP2850.

**Design**

“The TBS application consists of two distinct software modules: the token builder and the token switch. In our prototype, the modules are implemented on a single hardware development system (IXDP2850) although in reality they are likely to be situated in different locations. Therefore, our implementation consists of a demo system as shown in Figure 7.14. The token builder application module is implemented on two \( \mu \)Engines in the Ingress NPU, while the token switch module is implemented on two \( \mu \)Engines in the Egress NPU. Although the mapping can be easily scaled up to more \( \mu \)Engines, we use only two \( \mu \)Engines per application module because they provide sufficient performance already. As we will see in [the next paragraph], the bottleneck is the limited number of crypto units [on the IXDP2850].”

**Results**

The performance [...] is shown in Figures 7.15 and 7.16. It has two charts: (1) ‘data received’ which represents the received rate in the IXDP2850 box and (2) ‘successfully switched data’ which denotes the rate that the TBS could handle properly using just a single thread for processing. The ‘data received’ chart is low for small packets because of the Gigabit PCI card limitation used in the Rembrandt6 PC for traffic generation. So, for small packets it reflects the limitations of the traffic generator rather than those of the TBS. The second chart, ‘Successfully switched data’, is lower than the first one for high speeds because here we are
using a single threaded implementation. The multithreaded version coincides exactly with the ‘data received’ chart and is therefore not visible. While we cannot predict the real performance for speeds above 1 Gbps without performing measurements with a high-speed traffic generator, we estimated the outcome by using the Intel’s cycle accurate IXP simulator running in debug mode.
7.3.2 Other Applications

We have engineered more applications directly on top of Streamline pipelines. We now present a small set of representative examples.

**Kernel Snort**  Snort is a network intrusion detection system (IDS). It is widely used: according to the developers it is even “the most widely deployed IDS/IPS technology worldwide” [sno]. Snort detects intrusions by matching arriving network data against content rules. These may contain byte signatures, but also higher level protocol-specific information. Snort iteratively applies a set of rules against data until it matches or all rules are exhausted. Rule matching mainly consists of memory comparisons and is therefore data intensive. Normally, snort receives live data in userspace through the libpcap interface and performs all processing there. Intrusion detection is a classic fit for the “push processing down” heuristic of Streamline:FFPF, because it works as a very tight sieve: many safe packets can be discarded early, relatively few packets need considerable inspection. Intrusion detection (ID) is difficult in most libpcap-based implementations such as standard (or, “vanilla”) Snort [Roe99]. Even though most packets do not contain a signature, many packets still have to be copied to userspace, because only limited rule matching can be offloaded to the kernel with BPF. Streamline avoids these copies and the corresponding context switches by running the pattern matching algorithm in the kernel. Initial tests showed about a 50% decrease in CPU utilization for the same datarate.

**Weblog Generator**  For a research project in distributed wikis, we were asked to build a weblog generator that could analyze Wikipedia traffic at server linerate: at least 4000 HTTP requests per second. Because no software could be installed on the Wikipedia servers themselves, the log generator had to sniff a raw data stream and reconstruct all TCP connections and HTTP requests on a stand-alone host. The reference Streamline package comes with a filter, httpwiki, that reconstructs requests in a pipeline and communicates periodically with a controlling application through shared memory. The pipeline is offloaded to kernelspace to minimize task switching. It writes its results to a DBuf shared with the userspace application. This periodically reads the log and writes it to disk. Its interval is set to 10 seconds: a 40.000x reduction in task switches compared to a libpcap-based approach (and that is only if each HTTP request fits in a single network packet). Throughput can probably be increased further by offloading disk output to the kernel as well, as part of the pipeline. Throttling proved sufficient, however, to reach the set goal of 4000 req/s. Due to unresolvable privacy concerns, the Wikipedia ad-
Administrators finally decided against live operation in their Amsterdam data center, unfortunately.

**IP Flow Record (IPFIX) Generator**  IPFIX is an IETF standard for Internet protocol flow record exchange, based on an earlier Cisco de facto standard. It is recorded in Request For Comments (RFC) 3917. The Streamline reference system contains a filter that translates IP packet streams into flow records. Contrary to common implementations, the `ipfix` filter is not tied to a specific data source or application interface and can therefore generate statistics for a range of active sources and pipelines. Its overhead is always limited to only the cost of record generation itself, because it is always either specified as part of an existing application or else uses prefix sharing (Section 6.4) to attach to the existing streams.

**User Programmable Virtualized Networks**  The last application uses Streamline to build reconfigurable (or, programmable) networks. Such user programmable virtualized networks (UPVN [MSGL06]) span a control network across a network of machines running Streamline (and potentially other reconfigurable systems, such as Click routers [KMC+00]). UPVN allows users to directly modify behavior of nodes in the network and therefore of the network itself. As Figure 7.17 shows, the control interface is connected to a multitouch table interface for simple analysis and reconfiguration. Through this interface, users can route traffic between nodes and configure node behavior. The left-hand screen shows the entire network of hosts and links, where the user has just zoomed in to reveal the configuration of a single node. The
right-hand figure shows how with simple gestures, users can review network state and issue change instructions. Locally, on Streamline nodes, these requests map on the Streamline control system interface (Section 6.4). The dots in the image represent live Streamline filters. In UPVN, each Streamline pipeline is part of a larger distributed application. The software is developed by external users with only minor assistance from us. No serious changes to the Streamline architecture were needed.

7.4 Optimization

The presented applications used the push processing down heuristic of Streamline:FFPF or were manually configured. In Section 6.3.2, we explained that heuristics are of diminishing value as the application domain expands. Besides the proven heuristic-based algorithm, Streamline therefore embeds an optimization algorithm that is based on mathematical programming: slquant. Section 6.3.3 introduced the formalized data model and the next section presented a step-by-step transformation process from application and system descriptions to this mathematical integer program. The practicality of the solution rest on its efficiency and quality. This section evaluates optimization performance for applications and systems ranging from trivial (single filter and space) to fairly complex (ten filters and spaces) and demonstrates that for packet filters the solver arrives at the same strategy as the heuristic-based method without having been given hardcoded rules and that for other applications it generates common sense solutions on par with the choice of domain experts.

Efficiency

To be cost effective, optimization calculation time has to be low compared to execution time (Section 2.2.2). For long running networking tasks, cost is easily amortized. We therefore consider any humanly imperceptible runtime acceptable, which gives an upper bound of above 100ms. The experiments will show that complicated programs exceed this boundary on highly parallel systems, but that on current hardware (up to 10 spaces) and for average sized applications (up to 10 filters) runtimes between 1 and 50 milliseconds are more typical.

Figure 7.18 shows the results of calculating configurations for linear pipeline applications with the GNU Linear Programming Kit (GLPK) 4.29. We ran the solver for pipelines of increasing length and for homogeneous platforms with increasing numbers of processors (simulated). The applications themselves are unimportant; all filters are the same and known to exist in all spaces
(the only filter aspect that affects optimization performance). All tests are executed on the same hardware as the performance experiments in Section 7.2. We present the median of 11 runs. For all but the smallest (sub-millisecond) runs, upper and lower quartiles are with 10%. They never exceed 50%. The left-hand figure shows the variable space of generated models measured in
number of arcs. The right-hand figure shows execution time for the same models. In both cases the x-axis shows the number of vertices in the pipeline and each line corresponds to a number of spaces. As could be expected, variable space grows linearly with the length of the pipeline. Execution time, on the other hand, grows superlinearly. We see that our goal of 100ms execution time is feasible for all pipelines when system parallelism is fairly low: up to 10 spaces. As parallelism grows, either application length must be reduced or calculation bound scaled up. Figure 7.19 presents the same data, but now illustrates scalability with system parallelism (i.e., number of spaces) by replacing pipeline growthrate with system growthrate on the x-axis. Here, the left-hand figure clearly illustrates that state space grows superlinearly with system parallelism. As a result, execution time grows more rapidly even than with pipeline length.

To estimate to what extent scalability depends on application shape, we also executed this test for another type of application. Figure 7.20 presents problem size and execution time results for split-join applications. These consist of a single source node, a single sink and up to eight intermediate filters in parallel configuration (Figure 6.4 presents an example). As in the case of the pipeline, the actual application is not relevant to the optimization performance results. In extended pipeline notation, the request is as follows:

\[
\text{filter | filter + filter + \ldots + filter | filter}
\]

The figure shows that state space again grows linearly with application size.
The growthrate is steeper, however, due to parallel graphs having more streams per filter. Execution time also shows the signs of superlinear growth.

In short, the optimal solver in Streamline is capable of generating solutions for most common problems (10 or fewer filters) on current hardware (10 or fewer spaces) well within the budget of 100 milliseconds. In the current model variable space grows rapidly with system parallelism, however. The current measurements present an upper, but not necessarily a lower bound on performance. In general, integer programs are more expensive to calculate than (continuous) linear programs (Appendix B), but network flow programs have a special structure that make them amenable to as fast or faster optimization than linear problems. Our current solver, GLPK, does not expose a dedicated network simplex method. We therefore use an integer solver. To demonstrate that algorithm choice determines execution time to a large extent, we plot performance of two equivalent integer solvers, both part of GLPK, in Figure 7.21. We have seen the left-hand figure before; the right-hand shows performance for the same test with another solver. This one first calculates the linear relaxation using the simplex method (a common approach), while the faster approach may not (it chooses from a set of approaches at runtime). Results are up to two orders of magnitude apart for large requests. The solver can inflate cost in yet another way. Before it can start with optimization, it must first find a solution in the feasible region. Especially if this region is small compared to total search space, this step can take much longer than the actual optimization. If Streamline can present a (potentially bad) solution from the start, the optimizer can skip this step. Runtime is most problematic on parallel systems. Here, scalability can be further improved through model refinement. At least on homogeneous systems, many solutions are equivalent. Currently, the solver wastes time comparing all permutations of the same logical configuration; a next step in this direction is to automatically identify and extract equivalent solutions to limit growth without impairing optimization quality. Limiting state space is not difficult in itself — as the use of heuristics showed. The difficulty lies in extracting solutions without accidentally pruning feasible solutions prematurely. This is still an open issue.

Quality

The aim for the solver is to arrive at the same configurations as domain experts. No single example can prove that slquant generates good applications for all domains. We present two case studies that we can directly compare to known good solutions to set a lower threshold on practicality. The first compares the generic optimizer to the narrow heuristics of Streamline:FFPF to show that it reaches the same conclusions for this domain. The second com-
Figure 7.21: Performance Comparison of Two Equivalent Solvers

Figure 7.22: Selected Configuration for a Packet Filter

pares the optimizer to the hand-crafted SafeCard configuration that Streamline:FFPF could not map, because it is not a push processing down application. If slquant can generate a mapping comparable to the manual selection, its cost function likely works as intended. Besides these two previously introduced applications, we have examined other similar pipelines. In all cases, the generated configurations corresponded to the ones shown here, for the
same reasons (that we explain in the examples).

**FFPF** To demonstrate that the approach subsumes ‘push processing down’, we compare the choices of slquant and Streamline:FFPF for a straightforward packet filter application. Figure 7.22 shows the path slquant generates through the variable graph for a simple 3-stage pipeline

```
rx | bpf | user
```
mapped onto a dual-core machine that has BPF available in the kernel. The figure shows all possible filter and stream implementations. Filters and streams are represented by arcs; spaces by bounding boxes. Filter arcs are marked as dotted. The selected path is printed in bold. In the shown configuration (that is copied verbatim from the output of slquant, except for the dotted lines that are normally colored blue), the first two filters are co-located in the kernel on the same core. In other words, the packet filter is offloaded to the lowest available space. The third filter is not, because it can only be implemented in userspace. It is executed in a user process on the same core. We have also requested optimization of the same application on a system without a BPF implementation in the kernel. Then, the last two filters are necessarily co-located in userspace. According to the solver, that option generated almost 50% higher total cost, which is why the solver chooses the shown configuration over that alternative when kernel BPF is available. The actual differences in number of cycles are likely to be lower, but as we explained in Section 6.3.2, the measurement is sufficiently close, since it generates the correct result.

The example confirms that slquant pushes filters to their sources without being given a specific heuristic. It pushes processing ‘down’ in packet filtering, because that strategy minimizes total data movement cost. In model G₄ and up, memory loads and stores are accounted independently from filter processing. If all else is equal, moving a filter that drops blocks before a channel crossing will reduce the flow over the channel and therefore the cost attributed to the store and load arcs surrounding that channel.

**SafeCard** By relying strictly on “push processing down” heuristics, Streamline:FFPF limits itself to packet filter applications. Slquant, on the contrary, maps arbitrary network flow applications. To verify that it indeed arrives at reasonable mappings for non-filtering applications, we make it calculate configurations for SafeCard. Contrary to real SafeCard, the configuration of which is fixed, slquant adapts the configuration based on expected throughput. To show the effect of input on the model, we run the solver at two throughput levels. Figure 7.23 shows the output of slquant for the SafeCard pipeline at
16,000 500B packets per second (64 Mbps). The individual arcs and vertices are too small in this compact output to read, but we can clearly discern the bounded boxes around the two employed spaces. To present understandable results, we have asked smallquant to only render the spaces, vertices and arcs that are used in the final configuration. With 7 spaces on the IXP2400, the result is too large and cluttered to interpret if all elements are plotted.

Figure 7.23 shows that all filters with a μEngine implementations are mapped together onto a single engine. The other two filters are mapped on a single kernelspace on the xScale CPU. Figure 7.24 shows the same pipeline, but now modeled with 132,000 500B packets per second (528 Mbps). Here, the ruler filter has been taken out of the first μEngine and instead executes on two private μengines. The example shows that the solver tries to co-locate filters as much as possible. It only brings additional resources online when the data flow requires it. Section 6.3.3 explained that smallquant does not explicitly encode this rule, but that it derives from the cost of data store and load operations. Each space has to read data from a DBuf and possibly IBuf. Each additional space therefore increases total cost. Only when no alternatives with N-1 or fewer spaces exist will smallquant select a solution with a N spaces (all else being equal: on a heterogeneous system relative filter cost can change the overall picture). The number of employed spaces therefore grows with the data rate.

7.5 Summary

We have evaluated Streamline on three features: performance, reconfigurability and optimization effectiveness. By adapting to system configuration details, Streamline increases Unix pipe IPC throughput by factors (4-9x). At the application level this advantage translates into throughput increases ranging from 40% for small packets up to 10x for shared data access. In the first case, only task-switch cost is saved, since data movement is minimal. In the
second case, both switch and copy costs are markedly lower. These numbers can be considered a worst case bound, as the selected legacy applications are relatively impervious to optimization.

To judge reconfigurability, we ported Streamline to a particularly heterogeneous environment, where we used it execute complex high throughput I/O applications. In all cases, observed throughput managed to reach the physical limits of the system. In other words, Streamline did not impose any artificial bottlenecks on processing. Finally, we measured optimization effectiveness. Streamline can generate configurations for practical problems (<250 ms for up to 10 filters and 10 spaces), but variable space grows quickly with application and system complexity. The generated configurations are practical: for packet filters, the solver independently arrived at the same strategy that we hardcoded in a heuristic in Streamline:FFPF. For SafeCard, it settled on the same strategy that we selected by hand.