In reconfigurable systems, a control layer manages the composition of applications from atomic elements. The primary design feature of application-tailored I/O is that this complex task is not deferred to users, but performed automatically and optimally when an application is started. This chapter discovers optimization opportunities in Streamline (6.2), selects a solver algorithm and generates a problem model (6.3), and integrates the solver into Streamline so that it controls all data- and transport plane construction (6.4). Fine-grained access control ensures safe operation even in shared environments such as the kernel (6.5).
6.1 Overview

Contrary to compile-time solutions, system level optimization can maximize application performance in the face of growing hardware diversity. Because system-level automated (or, self-) optimization is rare in practice, we approach it conservatively by attaching to the Streamline data- and transport planes a control system that is both demonstrably utilitarian and fully comprehensible. In application tailoring, we distinguish between optimization itself, automation of this process and execution of the chosen solution in a real system.

**Optimization (6.2)** is the basis for high throughput on ranges of hard- and software. Streamline applications are pipelines with competing candidate streams and filters. Streamline enables selection through filter name reification, buffer specialization and - stacking and graph substitution.

**Automation (6.3)** supplants tedious manual system administration. Streamline computes a network flow program that incorporates all application choice and relevant system features (e.g., cache layout) and that evaluates all possible configurations in tens of milliseconds to demonstrate practicality of online quantitative optimization.

**Execution (6.4)** builds an application that implements the selected configuration. Streamline extends basic resource discovery and pipeline construction with transactions for fault tolerance, graph sharing for increased performance and just-in-time compilation and loading of filters for embedded operation and further increased performance.

**Fine-grained Access Control (6.5)** safely grants access to kernel and devices to untrusted users. Streamline reuses Unix file permission where possible and supplements these with safe languages and mandatory stream sanitization. Besides Unix-type isolation, fine grained permissions enable trading security for throughput on a case-by-case basis.

6.2 Optimization

Optimization implies choice. When delegating optimization responsibility to an external party, and particularly when that party is a dumb machine, the option space and fitness evaluation have to be rendered explicit. The next section describes generic automation concerns about knowledge representation and search algorithms. This section answers the question *what* choice
a controller has in Streamline to optimize application construction, i.e., to perform I/O application tailoring. It demarcates the option space by enumerating all degrees of freedom in the computation and communication layers. Besides naming the vectors informally, it also specifies for each vector an algorithm for fitness evaluation. Following the runtime system architecture, vectors are organized in two sets covering computational and communication choice. Optimization of control elements themselves is absent. Such self-adaptation is complicated, renders the system difficult to comprehend and does not immediately contribute to the goal of increasing I/O throughput.

6.2.1 Computation

In application tailoring, computation optimization is the task of selecting the best fit implementation of a filter: the implementation that maximizes a metric, such as throughput in bytes per second, selected by the application. If we say that the fitness value is to be maximized – which is certainly true for throughput – the task is to find all possible combinations of executable filter implementation and computation space and select the highest ranking pair.

Reification and reflection Reification is the action of binding an abstract notion to a concrete resource. Name based lookup marries a well understood (though not necessarily formally defined) moniker, such as “MPEG decode” or “grep”, to an executable implementation. Reification enables open, evolving, composite systems, because it enables a system to modify its components at will during execution. Reification is closely related to reflection. In the context of computing, both were introduced together by Smith [Smi82]. Here, reification concerns the action of binding proper, while reflection means the ability of a software system to perform introspection on its own structure. An intuitive and popular definition is given in the same work [Smi82]:

“An entity’s integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter.”

Jointly, the concepts enable self-adaptive systems that modify their own structure. Terminology is handled loosely in the literature; we will use the term reflective programming for this approach to adaptive systems. The power of reflective programming is widely acknowledged, judging by number of supporting programming environments, which encompass functional (Lisp) and logical (Prolog) languages, but also object oriented languages (Java, C#, Javascript,
Smalltalk). Even a system language like C supports reification through function pointers and reflective programming through dynamically shared libraries: the dlsym lookup method resolves a human readable name to a pointer.

The reflective step of finding a concrete implementation that fits an abstract specification can take many forms in principle, but is almost invariably implemented as a simple name to object code resolution. This is the form path lookup takes in operating system shells and it is the form chosen for Streamline. Name based resolution is an intuitive method of constructing executables from task-oriented declarative descriptions. It separates the concerns of what to do from how to do it, and as a result the high level interface can be seen as a form of declarative programming.

Selection Filter selection in Streamline resembles executable search in common computing shells. Shells perform a greedy search through an ordered list of directories: the search path. A program name is matched against the name of each executable file in a directory until a match occurs or the search space is exhausted. The greedy path lookup algorithm ensures that at most a single answer is given, even if multiple candidate solutions are available. System administrators rely on the greedy nature of path-based search to support multiple variants of the same applications. They can prioritize implementations based on the order of directories in the search path. Path-based search makes a selection among competing implementations of a given well defined operation (say, ‘cc’ for the local C compiler). Saying that path lookup performs an optimization is a step too far, however. Implicitly, an administrator may order the search path to give preference to one C compiler (say, the Intel CC) over another (GCC) because he or she knows this to execute faster or generate faster executables, but no trace of this rational decision is visible at runtime. Instead, it is hardcoded as a rule directly in the search algorithm. As system configurations or application demand changes, hardcoded rules of thumb can become invalid. Because in this form information is not reevaluated, it is possible – for complex systems with many rules even probable – that suboptimal choices are made. For this reason, we are wary of early solution space reduction and heuristics.

Best Fit Numerical Optimization Contrary to path lookup, filter selection in Streamline is based on quantitative optimization. An alphanumerical filter moniker, or name, corresponds to an abstract filter specification that may have multiple passive executable filter implementations, one of which is selected to become an active executing filter instance. Relative fitness of implementations depends on the optimization objective. In application tai-
lored I/O we emphasize throughput, but aim to support arbitrary objectives and sets thereof. For this reason, metadata is not strictly defined. Filter developers can attach arbitrary pairs of labels and integer values. Examples of labels are cost per filter invocation and cost per byte, but also false positive ratio and working set size. Furthermore, all examples can be refined into worst, average and best case estimates or probability distributions.

Uncertainty A known problem in optimization is that numerical approximations are often unsubstantiated and, hence, arbitrary. Application tailoring avoids the practical consequences at present, because the spread between candidate solutions is so great. Optimization is concerned with incorporating special purpose resources that perform factors or orders better than general purpose resources. The FPGA implementation of an NFA (non deterministic finite automaton) based pattern matching engine is not a few percent more efficient than its serialized CPU counterpart, it is orders of magnitude better [MNB07]. The goal is to encode the knowledge of an expert in the field, so that Streamline will reach the same conclusions without direct human involvement. If this knowledge has to be based on heuristics and estimates because no perfect benchmark can be found, so be it. That is not to say that any estimate is equally valid. Our task is to present a metadata model that is rich enough to express all known information, yet flexible enough not to demand detailed data if none exists. Again, we defer the general discussion of how to reason with sparse, stochastic and questionable data to the next section on automation and here limit ourselves to the data model for filter optimization. Each filter implementation and each space type can attach labels with numbers to encode quantitative metadata of its own choosing. For instance, a filter can express average cycle cost as key/value pair (cycles, 5000). We expect that implementations of the same filter type will generally select the same set of appropriate metrics. A pattern matching engine, to stay with that example, can encode experimentally observed throughput for a set of block sizes and pattern hitrates. Or, it can model overhead as a combination of per invocation and per byte overhead. The second method is more expressive, but for technical reasons not always viable.

In path lookup, all executables target the same homogeneous computing environment, whereas Streamline filters span a set of potentially heterogeneous spaces. A quantitative comparison between candidate implementations depends not just on the fitness of the filters, but also on that of their underlying spaces. The filter programming API is simple enough to allow the same implementation to be compiled to multiple target spaces, such as userspace PowerPC Solaris and kernel AMD64 Linux. Streamline must nor-
Control

malize the values to calculate a meaningful product of implementation and space fitness. If one is undefined, a hardcoded default average (0.5) is substituted. This way, implementations can signal both that they are “faster” or “slower” than default solutions.

**Template Substitution**  In the related work on composite systems (described in Section 2.3) and in Streamline itself, we see a strong bias towards protocol processing as application domain, in networking [Rit84] and multimedia [MMO+94]. Protocols are uniquely suited to reflective programming because they unambiguously define the implementation of a moniker. Moving beyond TCP, AES and MPEG (to name common examples) protocol processing and generic block operations (counting, limiting, spreading) requires a more powerful reification interface. We have experimented with more expressive reconfiguration scripts in Betagis [dBBB05], which performs reflective programming in Prolog to build Unix shell scripts for specific (often one-off) system configuration tasks. Tasks are either atomic, in which case solutions are snippets of shell code, or composite, in which case they take the form of templates with free variables. The Prolog interpreter constructs scripts by recursively substituting free variables in templates with other templates or lines of shell code. Streamline exposes a more limited programming environment than scripts, but templates perform a similar function.

Templates expand the optimization option space through indirection. The simplest templates, *aliases*, identify filter implementations as specializations of generic tasks. More powerful transformations are transformations from one arbitrary graph onto another. Each network interface in Streamline exports its own unique data transmission filter. It would be cumbersome to have to manually adapt requests to each specific type of hardware. Instead, filters such as `skb_transmit` also registers an alias to the more generic `tx` type. The TCP protocol filter `tcp` returns all TCP segments, which builds on independent tasks, such as checksumming and reassembly. A high-end NIC performs common network tasks on-board. It exposes all these features as filters, but also registers a template that encapsulates common network reception handling. Another example, Boolean selection, was shown to be implemented with template rewriting in Section 3.3.2. Streamline does not have a default set of templates. As is common for all element types, templates are registered in a repository by external developers. Templates are registered as a pair of graphs, one encoding the request to match and one its replacement. For instance, all network traffic is received from the Netfilter subsystem of Linux as follows:

```
traffic := netfilter_in + netfilter_out
```
DNS traffic is filtered by defining the template

\[
\text{dnstraffic} := \text{traffic} \mid \text{ipv4} \mid \text{tcp} + \text{udp} \mid \text{dns}
\]

At runtime, a request is expanded into the set of all equivalent requests through recursive application of all matching rewrite rules on each expansion (“forward chaining”, Section B.1). Template metaprogramming enables iterative concretization of abstract tasks, or “stepwise refinement” [Wir71].

Formally, template matching is a search for graph isomorphism between a template and all possible subgraphs of the request. This problem is NP-Complete [GJ90]. Even for fairly small requests, search space can be large: in the order of the number of arcs in the request \( N \) times the number of arcs in the template \( M \), for all templates \( P \) or \( \mathcal{O}(NMP) \). To considerably reduce search complexity, Streamline only allows templates that have a single source. It matches each source node to each vertex in the request. On a match, it performs a parallel depth-first search through the two graphs until they diverge or have fully matched.

**Algorithm Summary**  
In short, filter implementations are ranked by the product of the normalized implementation and space fitness values for a chosen metric. If a filter or space lacks quantitative data, a default average value is substituted. All else being equal, the highest ranking pair is selected. Among substituted equivalent graphs the graph with the highest aggregate fitness is selected, where graph fitness is calculated as the minimum cut of the graph. For both vertices and graphs, selection is random among equivalent top ranking solutions.

### 6.2.2 Communication

Streams, like filters, are amenable to optimization. The multihop path across spaces to connect each filter to the next can be optimized by making intelligent routing decisions and individual streams can be specialized (Section 4.4).

**Channel Selection**  
A stream is implemented as a stack of buffers on top of a communication channel. Both are amenable to optimization. In the current version of Streamline channel type selection is hardcoded, because the optimal channel is always known for each space crossing and invariant to all other variables. In other words, in practice the channel solution space is free of choice. Between userspace and kernel, shared memory communication is always preferred over copying, for instance. Especially on distributed
installations, where Streamline is currently limited to UDP channels, using different protocols for different communication patterns and media (DCCP over WAN, multicast over Myrinet [VLB96], etc.) [AHB05] would be a logical extension. The Network Inference Engine [dB04] tailors network protocol stacks in this manner. At present, however, such distributed operation is not the focus of Streamline.

Communication cost for a graph $G$ — where cost is defined as latency, bandwidth or any other metric — is minimized when the aggregate cost of all edges in the graph is minimized. For a pair of filters, transport cost is minimized by selecting the cheapest edge between them. When vertices are connected by two or more paths, optimization becomes a problem of interdependent variables. Total network cost is given by the sum of all paths. It is minimized if the most significant contributors to cost are optimized at the cost of less critical paths. In the next section we will see a more efficient equivalent alternative algorithm, but for now we perform brute force calculation of all possible combinations of paths to select – all else being equal – the optimal network.

**Buffer Selection** Streams are implemented by a network of buffers. The network implementation is hidden from users. Contrary to filters, optimization is not based on reification. Streams are functionally uniform, but specializable on secondary traits, such as whether and how discrete block boundaries are marked. For each stream, data characteristics can be inferred from filter metadata. For each buffer implementation, preconditions on data input are hardcoded. At runtime, the runtime system can automatically infer which buffers are capable of implementing a stream, for the most part. A few conditions unfortunately require manual verification; these are by default not part of the application tailoring process. Table 6.1 organizes the stream implementations from Section 4.5 into a few choices, underlines the default selection and states necessary preconditions for deviation as well as the optimization goals that will see advantage or disadvantage. T&L is short hand for ‘throughput and latency’, space is short for memory utilization.

Two specializations go beyond binary option selection and are configurable in a spectrum: buffer size (if static) and throttle depth. By default, size is governed by a channel hint that depends on the underlying memory size. For the most common channel, a mapped region in a cache coherent shared memory system, it is 50% of the fastest cache shared by both filters. Throttling defaults to a depth of 32 if enabled, as discussed in Chapter 5.

The optimal stack of buffer implementations can be automatically extracted for each given goal by using the information in the rightmost three
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Choice</th>
<th>Precondition</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>( N, 512\text{KB} )</td>
<td></td>
<td>Throughput</td>
<td>Latency</td>
</tr>
<tr>
<td>Event Batch</td>
<td>( N, 32\times )</td>
<td></td>
<td>T&amp;L, Space</td>
<td></td>
</tr>
<tr>
<td>Scalable</td>
<td>Scalable or fixed</td>
<td>Slotted layout</td>
<td>T&amp;L</td>
<td>Seek Time</td>
</tr>
<tr>
<td>Alloc</td>
<td>Dynamic or not</td>
<td>Single space</td>
<td>Space</td>
<td>T&amp;L</td>
</tr>
<tr>
<td>Layout</td>
<td>Byte or block</td>
<td>Block stream</td>
<td>T&amp;L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Marker or slotted</td>
<td>Record</td>
<td>T&amp;L</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>Align or not</td>
<td>Manual</td>
<td>T&amp;L</td>
<td></td>
</tr>
<tr>
<td>Sync</td>
<td>Block or not</td>
<td>Manual</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lossy or not</td>
<td>Dataloss OK</td>
<td>T&amp;L</td>
<td></td>
</tr>
<tr>
<td>NUMA</td>
<td>Caching or not</td>
<td>Channel spec.</td>
<td>T&amp;L</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>Burst or not</td>
<td>Channel spec.</td>
<td>T&amp;L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prefetch or not</td>
<td>Channel spec.</td>
<td>T&amp;L</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Specialization Optimization.

columns of Table 6.1. Application developers choose the goals by which graphs must be judged (e.g., throughput). Label matching prunes the candidate stack to contain only implementations that are listed as advantageous to the chosen goal. Precondition checking removes incompatible choices, such as a pointer-based dynamic buffer for a stream that traverses multiple spaces. Lossy communication is the clearest example of this. Implementations that require manual checking are marked in the table. For instance, Streamline's socket interface uses the alignment implementation is used for IP header alignment, but this cannot be employed for arbitrary native applications. Automatic precondition verification works as follows. All preconditions operate on one of three properties: data type, application- or system characteristics. The first is defined by filter developers in the form of a weak type, as first mentioned in Section 3.3.2. Relevant application properties are whether all adjacent filters are co-located in the same space or not. Hardware properties are extracted from the underlying channel implementation, as the size example showed.

**Algorithm Summary** Communication optimization combines path routing, channel- and buffer selection. For neighboring spaces, the optimal direct channel implementation is hardcoded. For distant spaces, a shortest multihop path is calculated. In principle, a multihop path may exist that implements a cheaper path than a given single hop option. For simplicity – and because it does not occur in practice – this case is not evaluated. Specialized
buffers reduce communication cost over the default store and forward network. For each channel, initially all buffers are selected that have registered as advantageous to the optimization goal. This set is known through label matching. Then, the candidate set is pruned through precondition testing.

### 6.2.3 Contention

The term “all else being equal” repeatedly appears when discussing optimization. Each time, it signals a potential conflict due to interdependent choices. Contention occurs when multiple filter implementations match the same name, in which case quantitative ranking must unambiguously guide selection. It is also seen between filter and channel optimization. Selection of an implementation binds an endpoint of one or more streams. For instance, a decryption operation can be offloaded to a specialized co-processor to reduce computational cost, but at the cost of requiring data movement from a CPU to the co-processor and back, increasing transport overhead. Global optimization must quantitatively trade off computation versus communication efficiency in each such situation. Consequent selection of optimal solutions even in entangled option spaces is a topic in itself; it is handled in the next section.

### 6.3 Automation

Optimizing large sets of variables for each application is tedious, at best. When variables are correlated, computational complexity quickly grows to exceed human capacity. Streamline offloads the search for a good configuration to the computer system, because that is much better at performing large calculations. This section explains this automation step of the optimization problem. It estimates solution space dimensions, then selects an algorithm, presents a formal data model and shows how the intuitive models we used so far can be translated into this model through a series of graph transformations.

#### 6.3.1 Complexity

Problem size is a function of the number of variables, the number of choices for each variable and the characteristics of interdependency. Briefly, Streamline has these variables:

1. filter location (space)
2. stream implementation (channel, buffer type)
Section 6.2.3 argued that these are not independent. A system with $N$ spaces and $M$ filter names will have between $M$ and $NM$ filter implementations. Because many spaces share a space type (and thus implementation) the upper bound is the more reasonable estimate. Then, the system can construct up to $(NM)^2$ unique streams. These numbers represent worst case bounds: when all options are independent and therefore implement a feasible solution. As a result of buffer stacking (Section 6.2.2), for $B$ buffer implementations, each stream can be implemented by many buffer combinations. If all implementations can be freely combined, this gives the powerset $\mathcal{P}(B)$. Worst case, each stream then has $2^B$ potential implementations. Total number of possible streams, then, grows to $2^B(NM)^2$ unique stream implementations. Because pipelines combine selection of multiple elements, combinatorial complexity grows rapidly. A pipeline of $M$ filters can implement each filter in $N$ ways and has $M-1$ streams between these implementations. For this, the state space is $N^M$ possible combinations of filter implementations and $2^{B(M-1)}$ possible stream implementations. To estimate real cost, let us calculate the problem size for a common application.

**Example** For a modest pipeline of ten filters overlaid over three spaces and only a single buffer implementation, all streams are fixed by the selection of filters. If each filter can be executed in each space, then we have 10 sets of 3 variables. From each set one must be chosen exclusively. Total variable space then is 30 – by all accounts a small number. The number of combination of choices that we can make is $3 \times 3 \times \ldots \times 3 = 3^{10} = 59,049$. Even for this small example, the problem is already infeasible to compute by hand. What is more, the number grows quickly as the number of spaces, buffers and templates grows and as cost factors are modeled increasingly accurately. The number of spaces can grow into the tens, even hundreds, on manycore environments.

On a 16 thread Intel Nehalem-EX with a kernel and userspace on each hyperthread, the solution space inflates to a staggering $32^{10} \approx 1 \times 10^{15}$ unique solutions. A pipeline has one fewer streams than filters. With 8 buffer implementations, this pipeline can be implemented with $2^{8^9} \approx 5 \times 10^{21}$ unique sets of channels and buffers. Templates further inflate these numbers. The number of templates that apply to each request is generally small. Commonly, a singular shorthand, such as "rx", is rewritten into a couple of competing composite implementations. This practice increases total search space by about an order of magnitude. Last, to improve correspondence with reality, model fidelity is increased by estimating actual data access and computation cost. A key feature of Streamline is that filters can share data and use indirect buffers.
to avoid repeated data storage. For globally efficient buffering, some filters must make the locally suboptimal decision to store data, so that others benefit from not having to store. Which filter is most suitable depends on the total network of read and write accesses. On parallel systems, the co-scheduling of filters to share caches also affects throughput; which filters to keep near each other is again a global problem.

Resource Bounds Because CPUs have a fixed (cycle) budget, choices cannot be fully independent. The instantiation of a filter in one space reduces remaining capacity of all spaces on that CPU. In other words, filters compete for space. An extreme situation that we encountered in practice (Section 7.3.1) concerns tiny processors that can only load a single filter onto a processor at once. Interdependence can be modeled as a constraint that invalidates all infeasible solutions without having to exhaustively compare all feasible ones, because constraints do not affect solution fitness at all.

6.3.2 Algorithm

The operating cost of an algorithm is directly related to the amount of search space structure that it can exploit. The more constrained the space, the fewer coordinates need to be evaluated to converge. Not all options are viable or even feasible, as the example of resource bounds showed. A solver that can minimize evaluation of infeasible solutions will generally outperform one that does not. Because system-level automation is fairly rare, we have paid special attention to an evaluation of search strategies: appendix B reviews a wide array of candidate algorithms and classifies them by problem solving quality and performance. This section introduces the two complementary approaches that we chose for application tailored I/O and argues both their strengths and weaknesses.

The existence of a considerably smaller feasible region than the whole space, coupled with a simple repetitive structure, indicates that more directed search strategies than Monte Carlo can be employed in application tailoring. Approximate methods and machine learning are too complex and slow. Instead, Streamline uses faster, tractable algorithms. For the well known application domain of Streamline:FFPF, it uses greedy, heuristic-based search. For others, we have built a more experimental optimal solver based on math programming. The first method is undoubtedly more easy to explain than the second, and has proven to give good results in many applications. On the other hand, its results are less robust than those of math programs that calculate from basic cost factors. We first review the greedy approach and note its advantages and weaknesses. Then, we argue why math programs are su-
perior in principle, but have scalability concerns in practice. The following two sections are devoted to a thorough explanation of the more robust, less straightforward, approach.

**Heuristics** Streamline:FFPF favors heuristics for application tailoring. The main heuristic is to “push processing down” [BdBC04]. This rule pushes filters away from the CPU onto peripheral devices as much as possible. It also pushes them towards peripheral input sources; even in the absence of smart peripheral hardware, this ensures that most processing (in this domain: filtering) takes place in the kernel – saving a copy to userspace for traffic that is discarded. The rule of thumb has proven to generate efficient configurations for the constrained packet filtering application domain. The approach is also easy to verify. In line with the heuristic, we originally modeled the graph of spaces as a vertical stack with application on top, a single kernel space below and various levels of peripheral hardware below that, as shown in Figure 6.1.

Not surprisingly, when we expanded the application domain, we found that the heuristic is very task dependent. Its success derives largely from the insight that as packet filters filter data streams, maximal efficiency is obtained when pruning takes place as close to data input as possible. For a network packet filter, the input is always the reception path of a NIC. It is not difficult to see that for other applications, the strategy will be less successful. Consider, for instance, a four stage pipeline where the first (input) and third filters can be implemented in the kernel, but the second and fourth only have an implementation in userspace. Push processing down will send data across a space boundary three times, even if the implementation of the second vertex is identical in kernel and userspace – in which case the solution is clearly suboptimal.

**Limitations** For general purpose optimization, heuristics-based search is not a robust strategy. With a large and a priori unknown set of applications and systems, constructing fundamental rules of thumb is precarious. Push processing down already had to be extended with additional rules to apply Streamline:FFPF to non-filtering networking tasks. [HdBBO5]. To avoid ping-pong behavior, we added a constraining rule that said that graphs may develop only in a single direction: up or down the stack. This rule, in turn, is ambiguous for requests that have multiple sources or sinks. A third rule fixed this issue (because it is far from succinct, we refrain from explaining it here). The set of rules will likely balloon as the number of applications, hardware configurations and application domains expands.

In short, by making a selection for each independent filter based on heuris-
tics, the greedy approach causes premature optimization. Within a well-defined application domain where heuristics are known safe, this is not a problem. For a general purpose I/O system, however, specific heuristics with broad applicability are hard to find. Incorrect hints give suboptimal solutions.

**Optimal Search** Structured optimal search in a single problem space has no unforeseen outcomes and is therefore most robust to changes in application and system domain. For applications outside packet filtering, Streamline supplements the heuristic search of Streamline:FFPF with a more robust, but computationally more complex, optimal search approach. It avoids logical programming for the reason given in Appendix B.1: efficiency. Instead, the solver, *slquant*, is based on mathematical programming. The approach is faster, because it exploits the convex problem structure to direct search to the global optimum. It is robust, because it does not discard options based on potentially invalid assumptions. Finally, it is understandable, because the algorithm is simple: it lacks 'magic' parameters that have to be tweaked and the relation between input and solution can be verified by hand. To further limit time complexity, the large obvious network model is encoded as a network flow component.

Integer programming, while faster than logic programming, is a generic search strategy and therefore not particularly well suited to any specific problem (Appendix B). We are unable to select a more specialized algorithm, because—to the best of our knowledge—the problem does not perfectly fit a more constrained problem. Superficially, we are dealing with a combinatorial knapsack problem, as we must identify whether a given amount of processing can be achieved given a set capacity limit. Although knapsack prob-
problems are NP-hard, there are known polynomial time bounded approximation algorithms [KPP04]. Introduction of multiple (types of) resources and constraints on combinations pushes our problem beyond the pure knapsack problem, however, rendering all bounded approaches inappropriate. If we forgo the budget (knapsack) constraints, the problem resembles another NP-hard problem with a polynomial time bounded approximation algorithm [KT02]: informally, the metric labeling problem assigns a label to each vertex in a graph from a fixed set of labels, where additional constraints on label assignment are specified by a separate graph that connects labels themselves. Matchings of label to resource are only allowed when neighbors in the destination graph have labels that are also neighbors in the label graph. As said, this problem does not consider budgetary constraints. Since both presented problems are NP-hard and neither is sufficiently generic to express all our constraints, it is likely that the more generic matching problem in Streamline is NP-hard. We forgo a proof.

**Pragmatic Optimal Search** A subtle point in goal selection concerns whether optimality is sought, or whether a configuration is required to reach a certain sufficient level of operation. The goal of automation in Streamline is to optimize applications as well as a domain expert would, only now consistently at top ability and considerably faster. To select a configuration without human involvement, a total order of all candidates has to exist. Because solution quality is a combination of individual filter and stream choices, the numerical ranking is a combination (or, trade-off) of individual aptitudes. These basic numbers are hardcoded in Streamline; they are meant to agree with the estimates an expert normally uses. When we say that we calculate an optimal mapping, then, we do not mean that Streamline always arrives at the best implementation of an application: only that the solution is optimal given the model. To what extent this choice corresponds to real world optimality depends on correspondence of the model with reality.

Lack of hard data for filter parameters makes quantitative optimization a known hard problem. We briefly discussed this issue in Section 2.2.2, where we argued for a simple model to minimize potential confusion. In Streamline, not only is the application model simple, it does not have to be of high fidelity. To reach optimization quality comparable to domain experts, it has no need for more information than a human administrator has. Since human experts do not have accurate filter cycle cost at the tip of their fingers, neither does Streamline. In many cases, broad comparative information suffices. Knowing that a co-processor is faster than the CPU at cryptographic operations is enough information to consider off-loading crypto filters, for instance.
6.3.3 Data Model

The intuitive application and system models must be brought into an integrated canonical integer form to make use of math programming. We assume that mathematical programming is efficient enough for loadtime execution if the state space is sufficiently constrained. We experimentally verify this point in the evaluation section. Model building is more art than science, but structured approaches do exist. We follow the classic approach of structured synthesis [Sch81]. Here, an optimization problem is formulated as built from four datasets. The first, variables, implement choices, such as which channel to select. The second, parameters, numerically encode properties, such as a channel's capacity. Constraints limit the solution space; they are expressed in linear (for a linear program) relationships of variables and parameters. Finally, an objective also expresses a relationship of variables and parameters, but instead of an (in)equality, it is a function to be minimized or maximized (i.e., optimized). Goal is to find the set of variable values that achieves this. This section first introduces each dataset in detail and then summarizes the integrated model. We do not explain the concept of mathematical programming here; the approach is introduced independently from application tailoring (with simpler examples) in Appendix B.

Variables Variables encapsulate choice. They are the only means for the automation algorithm to maximize objective fitness. Each optimizable property must be expressed in an individual variable. Application tailoring is primarily concerned with finding implementations for filters and streams. Each of these request networks, therefore, maps onto one or more variables. In practice, this means that each filter and stream will be implemented by a set of options from which one exclusive choice has to be made.

Bufferstacks Buffers can be combined to form an execution stack. To curtail state space, and because selection is independent from the rest of the problem, buffer stack selection is not integrated in the main problem model. Modeling all possible combinations of buffers would increase the number of variables for each arc by approximately an order of magnitude. Instead, it is more efficient to precompute optimal stacks for each space crossing once and to insert the result at the different applicable occurrences (options) in the global request. At present, optimization of stacks is a manual task.

Parameters Parameters encapsulate numerical properties of options; they are the basis for quantitative selection between option sets, or solutions.
Common examples are latency expressed in cycles or nanoseconds, throughput expressed in bytes per second and memory utilization in bytes. In application tailoring, the main metric is processor cycles (per second, per byte, etcetera).

**Cycles** Streamline tries to minimize total CPU cycle count for a given data rate. In practice, cycle cost can differ between CPUs in a heterogeneous system. The current model does not explicitly take that into account, but a weighting function could easily be applied if needed. Cost is measured by attributing a cost function to each filter, each data store and each load.

The incurred cost at each filter and stream depends on the data rate observed by a filter. Streamline uses a simple recursive graph walk from each source downstream to calculate rate at each filter. The user specifies the expected data rate at each source in blocks per second and average bytes per block. With this information, data rate at each non-source filter is calculated as the combined rate of its inputs. Filter cost depends on rate, but it is not a completely trivial mapping. Filters do not spend the same amount of processing on all blocks, or on all bytes in a block: for instance, an IP port filter only touches a few bytes, regardless of packet size; a sampler only touches one in every so many packets. Additionally, some filters drop, add or edit data; we summarized the various filter classes in Section 5.3. This behavior is represented in the model by having filters apply scaling factors to the block-rate and byte-factor: decrease it through filtering and increase it through data creation. To differentiate between modification (deletion plus addition) and simple forwarding, the two scale factors have to be encoded independently. The model can only truthfully encode filter behavior if it is consistent; we review solutions to variance shortly, but first introduce this simpler model.

To keep the system practical, the individual per-filter cost functions must be simple and parameters easy to estimate. Filter cost is expressed as a combination of per-invocation and per-byte cost. Each filter implementation has this combination of parameters. Data load and store cost is purely per-byte. The final filter model, then, consists of two four-tuples \( I_f \) and \( B_f \). The first encodes the per-invocation cost and scaling factors: cost in cycles, ratio of blocks on which actual time is spent processing, ratio of blocks forwarded and ratio of blocks introduced at this node. The second encodes the cost per byte and the ratios of bytes read, forwarded and created per block. Filter developers need not write these models directly: most fall into one of the standard categories that we defined through classification in Section 5.3. These can derive their cost function from a small set of standard options. Table 6.2 revisits the classification and fills in the model data, where letters represent
choice.

Memory access cost is calculated as the byterate times a per-byte hardware specific estimate. Spaces on the same core can share data from a fast cache to reduce access cost over default DRAM access. With multilayer caches, data can even be retrieved fast between cores. The optimization model takes these access latencies into account. Here, memory access cost is calculated using the distance between neighboring filters in the pipeline. This results in a cache-aware placement strategy that favors allocating filters with minimal memory access distance. Figure 6.2 shows the multicore CPU introduced in detail in Figure A.3 with two container tasks. Filters are located on neighboring cores to be able to access data from the fastest shared cache.
Effects of Variance in State and Input  For certain tasks, the input pattern is also an important factor in efficiency. Depending on the arriving data, a network intrusion scanner may have to process only one byte or an entire block. The two have completely different cost factors. This input diversity is not encoded in the model itself, but the effect of variance can be measured by running the solver multiple times for different parameter sets. For the example, the bytes read and blocks forwarded fields of $B_f$ and $I_f$, respectively, would be altered. Such stochastic programming gives stability information of a given configuration to changes in hardware and software (Appendix B.2.1).

Constraints  In math programming, the feasible region of the solution space is constrained through equations of parameter-weighted variables that must hold for a solution to be admissible. Such numerical constraints implement hyperplanes that split the solution space in a feasible and non-feasible part (technically speaking, constraints can be expressed in the objective function through the construct of Lagrange multipliers, but we are interested in their function, not implementation).

Connectivity  For math solvers to return feasible solutions, all constraints must be explicit – even the most elementary ones. The principal constraint on application tailoring is that a solution cannot be feasible unless each filter has exactly one implementation. Connectivity constraints form standard network flow programs that the network simplex method can solve efficiently. A network flow program (NFP) is a highly constrained integer program where each set of equations $\alpha_1 x_{i1} + \alpha_2 x_{i2} + ... + \alpha_n x_{in} = b_i$ and each $\alpha_j$ is one of $-1, 0, 1$ (basic connectivity modeling is explained in more detail in Appendix B). Together, the set of equations forms $A x = b$. $A$ is the incidence matrix of the flow network: each row $m$ represents a vertex and each column $n$ an arc. The value of element $A_{mn}$ is $-1$ if the arc originates at the vertex, 0 if there is no relation, and 1 if it terminates there (looping arcs are not allowed). $B$ encapsulates constraints: if $b_i$ is 0, input and output for vertex $i$ must be equal. This is the common conservation of flow constraint that ensures ‘goods’ flow through the network. If a single good arrives at a vertex, it must ‘choose’ a single outgoing link to continue on. This arrangement is the basis for the transshipment problem. In application tailoring, data is not a physical good: it can be duplicated, split or modified at will. In the pipeline

```
rx | ip | tcp | http + ftpdata
```
data is duplicated after the tcp filter, for instance. We extend the model to support these features in the next section. Without sources, no goods or data
enter the network. Sources and sinks are created by making $b_i$ non-zero. A negative $b_i$ allows a negative sum for a row: more outgoing than incoming arcs. In other words, it models sprouting of flow out of nowhere. A positive value absorbs value and thus implements a sink. In application tailored I/O, these connectivity matrices encode candidate mappings from filters to spaces and from channel to pair of spaces. Like capacity and cost, they are a parameter set. Because connectivity can only be an integer in the range $[-1, 1]$, the whole problem falls in the category of integer problems. These are more expensive to solve than (continuous) linear programs, unless wholly of the constrained network flow type. The next constraint will show that application tailoring is a general integer programming problem.

**Capacity** In application tailoring, the principal constraint models hardware capacity. If cost is expressed in terms of cycles, then resource capacity is expressed as cycles per time interval. In Streamline, resources are identified by spaces, which means that each space should have a capacity attached. The constraint then becomes that the total set of filters in a space plus the total set of streams incident on the space may together consume fewer than or equal the number of cycles given by its capacity, where capacity is expressed as parameter. One complication is that spaces in Streamline do not represent individual resources. Each CPU can have an arbitrary number of kernel and user spaces. Multiple spaces consume resources from a single resource budget.

**Objective** Constraints on the search space are encoded as equations with strict bounds on variables. One such equation is not bounded, however; instead, the goal is to maximize (or minimize, depending on goal) this objective function by selecting an optimal set of variables. For a vector of $N$ variables $x_i$, the objective function expresses an equation of variables multiplied by parameters, for example

$$\text{min}(c_1x_1 + c_2x_2 + \ldots + c_Nx_N)$$

Commonly, this is expressed as $\text{min}(cx)$. With input rate estimated, the objective in application tailored I/O becomes to minimize the cost for this given flow network. In other words, this is a minimum cost flow problem. To find the maximal flow that a given platform can process, we perform a form of gradient descent search for different input source ranges: a min-cost-max-flow approximation.

By default, flow represents throughput, but parameters can be adapted to quantify another goal. For some common system operation goals, notably
latency and power minimization, the metric does not have to be changed. Here, only the objective function changes from minimization of total cost to minimization of the longest path or of the number of applied resources. Unfortunately, it seems unlikely that power minimization can be expressed as a mathematical program, at least using the current notation. We state without proof that minimizing the number of fixed-capacity resources is a bin-packing problem, which is NP-hard by itself. Since cost is attributed to columns in matrix $A$, we could aggregate cost per resource by adding a column for each resource; however, we cannot express a linear objective function that counts the number of columns used: cost is always calculated through an algebraic expression (normally, summation). A less satisfactory, but practical, solution to the power minimization objective is to express constraints as before and perform a satisfiability search: vary the number of resources and calculate whether a given application be run given that set of resources. To efficiently find the solution with the fewest number required (for symmetrical resources), bisection methods can be applied, which bound search complexity to $\log_2(n)$. To avoid using many resources inefficiently, the number of occupied resources should be minimized even when power conservation is not the main aim, all else being equal. As the last statement implies, this aim is lexicographically ordered (Appendix B.2) behind the main goal and can therefore not be trivially written as a linear combination of objectives. Linearization is possible in principle [Ise82], but in Streamline, we can avoid complicating the objective function because we are certain that the objective is always minimized as a result of total cost minimization: data spread increases memory contention; the optimal solution from a set of otherwise equivalent candidates is the one that uses the fewest resources.

**Summary**  
We arrived at the following model for application tailoring:

**Variables** select one from a set of arcs, where each set represents all implementation options for a stream or filter. There are as many variables as there is choice. A lower bound is given by the sum of all filters and streams in the request graph. Together, all variables form the vector $x$. For reasons that will become clear in the next section, all variables are represented as arcs in the linear model, including those that represent vertices in the request model.

**Parameters** encode numeric data of variables. In application tailoring, connectivity, cost and capacity are the main parameters. Each choice has a cost associated. Cost is attributed to spaces in a space by arc matrix $c_e$. 
Each resource, in turn, has a capacity, which is hardcoded in the vector $r$.

**Constraints** limit the feasible region. The connectivity constraint states that for all vertices $i$ and arcs $j$ the equality $A[i,j] \times x[j] = b[i]$ must hold. A secondary constraint is that each element lives in one space and that each space exists on one resource. The capacity constraint states that the capacity of a resource may never be exceeded. These constraints are integrated by adding a row to connectivity matrix $A$ for each resource and attributing cost for each column (arc) to the matching row.

**Objective** is to minimize global cost. The second objective, minimization of number of applied resources, does not have to be expressed independently. This gives the single objective function

$$\min(\sum_{j}^{} c_e[i] \times x[i])$$

### 6.3.4 Model Construction

Application tailoring maps a request digraph (a pipeline) onto a system digraph of streams and channels. The native models cannot be solved by a math program solver; the two must be integrated, so that all optimization opportunities identified in the previous section (6.2) are variables of a single integer program. We now present a five phase process that transforms the combination of request digraph $G_r$ and system digraph $G_s$ step-by-step into a linear (integer) form suitable for an external solver. Table 6.3 summarizes the transformation process. Figure 6.3 graphically depicts each stage; the labels are explained in the appropriate paragraphs. The process has an inverse – this is necessary to be able to transform the otherwise meaningless optimum of the canonical problem into a practical application configuration for Streamline.

**$G_r$ and $G_s$: Intuitive Models** The input to the process consists of two digraphs. $G_r$ is a combination of labelled vertices $f_x$ representing filter names and unlabeled arcs $s_y$ representing streams. In system digraph $G_s$, labeled vertices $SP_z$ model spaces and labeled arcs $C_{iw}$ represent channels.

**$G_1$: Coalesced Linear Model** In a linear model, arcs represent choice. So far we equated these with streams, but in application tailored I/O, choice is not limited to streams alone. Commonly, a math program $Ax = b$ has a single
### Table 6.3: Overview of Transformation Phases

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>G₁</td>
<td>Linearize</td>
<td>models all choice as arcs, including filter selection.</td>
</tr>
<tr>
<td>G₂</td>
<td>Substitute</td>
<td>expands choice to all the equivalent solutions.</td>
</tr>
<tr>
<td>G₃</td>
<td>Merge</td>
<td>Merges input graphs.</td>
</tr>
<tr>
<td>G₄</td>
<td>Add Cost</td>
<td>models each stream as as set of read and write arcs.</td>
</tr>
<tr>
<td>G₅</td>
<td>Add IBufs</td>
<td>adds parallel arcs (choice) for different buffer types.</td>
</tr>
</tbody>
</table>

### Figure 6.3: Consecutively Refined Model Stages

Objective $cx$ that is to be minimized or maximized. The next step, therefore, is to transform a graph consisting of filter vertices and stream arcs into one that coalesces both object types onto vertices. In this coalesced linear model, $G₁$, vertices have no practical meaning, while arcs represent both filters and streams. By coalescing, we relate the optimizable properties of two distinct object types to one another. The mapping is only feasible if both use the same currency, which is true in application tailored I/O, where both are expressed...
in cycles.

\textbf{G2: Substituted Request Model} The template substitution mechanism described in Section 6.2.1 expands the request \( G_r \) to a set of equivalent requests \( G_r^+ \). The next step is to transform this disjoint set into a single problem. The only valid solutions are those that exclusively choose one of the parallel paths as a whole. Figure 6.4 depicts parallel equivalent paths combined in a single choice graph through a common input and output. The global input vertex is connected to the local input node of each subgraph. This, in turn, connects to all sources in the graph. Similarly, all sinks are connected to a local output vertex and all these vertices are connected to a single global output vertex.

\textit{Choice} Modeling choice splits appears straightforward. The problem arises when we observe that splits in the merged graph represent not only choices, but that some graphs contain task splits, themselves. A split can-
not coincide with a choice, because then split arcs cannot be discerned from choice arcs. Instead, choice is modeled by setting capacities on arcs and enforcing a flow conservation constraint: the inflow and outflow at each vertex must be the same. Split and choice networks can then be discerned by nesting one in the other. Figure 6.5 shows an example, where a single vertex, $F$, can be implemented as a single instance $F_i$ or as a pair of data parallel instances $F_{j1}$ and $F_{j2}$. The constraint on flow conservation ensures that each choice $f_{j1}$ has to maintain the same amount of flow to be a feasible solution. As a result, the 1-way and 2-way options subgraphs have to have different amounts of flow internally. Note that the specific numbers are immaterial: all that is important is that they add up to the same sum.

**Sources and Sinks** Subgraphs can embed sources or sinks and have cross- and backlinks. These effects complicate the algorithm to compute gains. In the case of a sink, the capacity of the sink is lost: it is not recovered when the subgraph terminates at a join. Similarly, a source in the subgraph increases flow: higher output will leave at the join than entered at the split. Backlinks either transfer data to another subgraph (crosslinks) or cause a directed cycle (Section 3.3.1). Neither can extend out of the choice subgraph, otherwise it would not implement a distinct choice. Crosslinks leaks capacity from one subgraph to another otherwise disjoint subgraph. Backlinks redistribute capacity within the choice graph. Figure 6.6 revisits the graph with backlink from Section 3.3.1 to show the first case; Figure 6.7 shows the second case.

**G₃: Merged Request and System Model** To create a network flow program that models all choice as a single connectivity graph, the input graphs are integrated. We call the resulting graph $G₃$. Each filter $f_x$ in $G_2$ in is replaced by a set of candidate filter implementations $x$-$SP_i$, one for each space $i$ that implements the filter. If $G rv$ has an arc from a filter $f_1$ to another $f_2$, then each $f_1$-$SP_i$ in $G_2$ is connected to each $f_2$-$SP_j$ (the vertex sets form a bipartite subgraph).
Without precautions, this transformation causes a state explosion, because in the merged graph the number of edges between each pair of filters is approximated by the product of the number of candidate filter implementations (since each combination of implementations is a solution). A small pipeline with ten filters and eight potential implementations per filter (each in a separate space) will have $10 \times 8 = 80$ filter arcs, but $9 \times 64 = 576$ stream arcs: a much larger number. The total number of arcs rises quickly due to the quadratic term. In the evaluation section we will observe how critical this state explosion is to efficient calculation.

**G₄: Cost Model**  The primary constraint on resource use is that each resource has a capacity that cannot be overrun (for most metrics, a cycle budget). This constraint is modeled through an inequality that says that all variables for all spaces of a given resource consume less than or equal to the fixed budget of that resource.

$G₄$ models the data store and load operations independently around filter execution to be able to attribute these costs to specific resources. Because Streamline manages all data access, it can infer which filters communicate a lot and should be interconnected cheaply. To minimize communication overhead (e.g., avoid ping-pong transport between tasks), communication lines must always be kept short, all else being equal. To account all data access costs, each filter implementation $f₁\cdot SPₐ$ is modeled not by one, but by three arcs $f₁\cdot SPₐ \rightarrow w$, $f₁\cdot SPₐ \rightarrow c$ and $f₁\cdot SPₐ \rightarrow r$, where $w$ stands for write, $c$ for connect and $r$ for read. This refinement is also present in temporal communication graphs [Lo92]. Worst case, it doubles the number of arcs in the graph. This happens in the absence of choice. Because parallel options cause a quadratic growth in arcs, however, the linear extra cost of the read and write arcs are immaterial for most graphs. Figure 6.8 shows the separate read, write and interconnect phases and the difference in growthrates as the number of
spaces grows.

**G₅: Buffer Variation Model**  Different read and write methods give different data access costs. The next graph, G₅ refines the naive store and forward model by modeling the actual cost of storing to and loading from a DBuf and comparing that to indirect transport through and IBuf and through pointer forwarding. Streamline presents three main read options: pointer access (cached data), read from a DBuf and read from an IBuf. In Figure 6.3, these are represented by replacing the w and r streams with specialized wD for DBuf write, wI for IBuf write, wP for pointer forwarding and rD, rI, rP for the corresponding read methods. Similarly, it has three write options: no write (for pointers), write to a DBuf and write to an IBuf. The last option further splits in two possibilities: if data is already stored the call completes, otherwise it triggers a write-through to a DBuf. G₅ expresses all these paths. Each filter has three write options: pointer, indirect and direct. On the other end, each filter can similarly read data from a DBuf, from an IBuf plus DBuf, or from a local pointer. Pointer write followed by pointer read is only feasible if both occur in the same space. In this refined model IBuf use is not free, as the gains at write time are traded off by additional cost at read time.

Figure 6.9 depicts the possible communication patterns between two vertices. It draw special attention to two common paths. The bold solid line represents local pointer forwarding, the bold dotted line cross-space forwarding with IBufs when data is already stored. It also shows the possibilities that the upstream filter buffers (IBuf and/or DBuf) even though the downstream filter only reads a pointer or DBuf. These paths occur when a controller application specifically requests storage or when a filter has multiple downstream neighbors. Since not all arcs in the model represent streams, buffer variation appears to increase total model size with a factor at most three.

Unfortunately, this step proves infeasible to implement without state ex-
plosion because the constraint that a DBuf write/read can be forgone only when data is already in a DBuf introduces a measure of statefulness to the model. In a parallel graph, multiple subgraphs can take different paths through the resource and then recombine. But, when we add the constraint that subgraphs can only join when their state (having, or having not, saved data) corresponds, we start having to duplicate each downstream path that used to recombine into one that has saved state and one that has not. Each filter that can forward to a remote space can generate all possible paths (saved and unsaved), to each of its potential children, causing rapid growth combinatorial complexity. It may well be possible to model this interaction in a scalable manner, but we have not found a feasible method. In the remainder of this thesis, we use a trivial and unsafe implementation that is oblivious to state and therefore allows all combinations. We manually check resulting configurations for correctness. As more practical approach, we for now suggest hard-coding the decision to write to DBuf as soon as possible, even though this heuristic causes unnecessary writes on some occasions.

6.4 Control System

Controller applications communicate with the runtime system to discover system features and to enact system changes. This section discusses the interface and implementation of this control logic. We glance over the straightforward parts and elaborate only on some of the more surprising features: fault tolerance, application overlap detection and dynamic filters.

**Controller Application** To discover system information and to initiate system changes, a controller application (written in Python) communicates
with the Streamline runtime system through a messaging API exposed from the Streamline dynamic library. When it has acquired a user request and recent system state information, the program executes the graph transformations of Section 6.3.4 to arrive at an integer program with large network flow component. Optionally, it graphically depicts the output of each step, as shown in Figure 6.10. The final model is a description in a subset of the standard (for this domain) MathProg language that can be processed by the GNU Linear Programming Kit (GLPK). The controller calls GLPK, interprets its output and from this linear model creates a mapping from each filter in the original request to a space in the runtime system. It communicates the changes that have to be carried out in the runtime system to the spaces through messages. For safety, the concurrent construction processes are separated by making an entire construction process a single critical section (the data path is not quiesced, on the other hand). We now turn our attention to this lower construction layer.

**Infrastructure**

The core of the control layer is a message-passing network that interconnects all spaces with each other and with the controller applications, as shown in Figure 6.11. This design matches with the model of spaces as independent systems that manage their own active and passive objects (filter implementations and instances, buffers, etc.). The controller has
to break up complex requests into micro-calls, such as “can I execute (an implementation of) filter X here?” or “open a channel to space Y”, and transmit these to the applicable spaces. Micro-calls fall into two domains: sensor and actuator calls. The first set includes many discovery requests, such as the first example call, some of which need a broadcast mechanism. Construction requests, on the other hand, are generally similar to the second example, where a command is sent point-to-point to a specific space. Optionally a response is expected. In total, then, the message passing layer implements broadcast, RPC and notification IPC.

Each space is responsible for its own state. It accounts all active and passive state in a local registry and performs access control independently on each micro-call. Each space has a single router for all incoming messages, which doubles as access control choke point for locally-bound messages. When a message arrives, the space reads the embedded micro-call number, looks up a matching micro-call handler and executes that routine. The indirect lookup step allows different spacetypes to implement features in different ways.

Spaces may be very confined and incapable of implementing a full IP stack. To avoid such a dependency, Streamline reuses existing communication elements, notably channels, to communicate control.

Another peculiarity stemming from embedded spaces is that space control logic needs not be co-located with the actual space. For kernel spaces, it is no more than logical to execute all security checks in the kernel as well; some embedded processors, however, lack the power. The IXP network processor (discussed in Section 7.3.1) runs control for both the control and µ engine co-processors on the control CPU; communication between µ engines and their control is implementation specific. Applications are not aware of these details.
Table 6.4: Construction API: Sensors

<table>
<thead>
<tr>
<th>Message</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG_FUNC_DISCOVER</td>
<td>match an implementation of a filter name</td>
</tr>
<tr>
<td>CM_GET_SPACE_LIST</td>
<td>list all spaces</td>
</tr>
<tr>
<td>CM_GET_FUNC_LIST</td>
<td>list all filter implementations</td>
</tr>
<tr>
<td>CM_GET_INST_LIST</td>
<td>list all filter instances</td>
</tr>
<tr>
<td>CM_GET_ROUTE_LIST</td>
<td>list all potential channels (‘routes’)</td>
</tr>
<tr>
<td>CM_GET_CINST_LIST</td>
<td>list all channel instances</td>
</tr>
<tr>
<td>CM_GET_TRAN_LIST</td>
<td>list all buffers</td>
</tr>
<tr>
<td>CM_GET_INST_INFO</td>
<td>return details of a specific instance</td>
</tr>
<tr>
<td>CM_GET_INST_DATA</td>
<td>return data from a filter’s stateful memory</td>
</tr>
<tr>
<td>CM_GET_TRAN_INFO</td>
<td>return details of a specific buffer</td>
</tr>
<tr>
<td>MSG_FILTER_FINISH</td>
<td>ask for call back when a request has finished</td>
</tr>
<tr>
<td>MSG_GET_PARENT</td>
<td>find the authority to acquire global IDs</td>
</tr>
</tbody>
</table>

Figure 6.12: Lifecycles of Filters and Streams

**Sensors and Actuators**  Micro-calls are a control application’s sensors and actuators into the runtime system. The first thing the controller has to do is to discover all spaces, filters, channels and buffer implementations (templates are stored centrally in the higher layer interface, not one of the individual spaces). In other words, sensing primarily consists of discovery of different types of objects. It will be no surprise that many sensor messages are broadcast to all spaces at once. Table 6.4 lists the sensor calls.
Control

<table>
<thead>
<tr>
<th>message</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG_FUNC_OPEN</td>
<td>create a filter</td>
</tr>
<tr>
<td>MSG_FUNC_CLOSE</td>
<td>destroy a filter</td>
</tr>
<tr>
<td>MSG_FUNC_EXPORT</td>
<td>make filter queue data and signal events</td>
</tr>
<tr>
<td>MSG_FUNC_DEEXPORT</td>
<td>stop queueing and signaling</td>
</tr>
<tr>
<td>MSG_FUNC_CONNECT</td>
<td>create a stream between two filters</td>
</tr>
<tr>
<td>MSG_FUNC_DISCONNECT</td>
<td>sever a stream</td>
</tr>
<tr>
<td>MSG_FUNC_IMPORT</td>
<td>listen on a buffer</td>
</tr>
<tr>
<td>MSG_FUNC_UNIMPORT</td>
<td>reverse import</td>
</tr>
<tr>
<td>MSG_FUNC_ACTIVATE</td>
<td>start processing</td>
</tr>
<tr>
<td>MSG_FUNC_PAUSE</td>
<td>stop processing</td>
</tr>
<tr>
<td>MSG_FUNC_UPDATE</td>
<td>update a filter parameter</td>
</tr>
<tr>
<td>CM_SET_INST_DATA</td>
<td>update filter state</td>
</tr>
</tbody>
</table>

Table 6.5: Construction APIs: Filter Actuators

Controllers build applications by issuing sequences of micro-calls. A graph decomposes into separate filter and stream construction tasks. These, in turn, deconstruct further into allocation, connection and activation steps, to name a few. The lifecycle of a filter is a state machine with six states, to implement fault tolerance. Figure 6.12 shows the state diagram. Most transitions consist of a single type of message. One is more complex: setup of communication channels across spaces requires multiple setup and teardown messages, where we discern between shared memory (the fast preferred case) and bitpipe (a fallback option) implementations. All transitions take zero or more actual actions, depending on the number of counterparties. For instance, the number of local inputs determines the number of calls during transition from ‘open’ to ‘connected’. Tables 6.5-6.7 lists all construction messages, about half of which correspond to the transitions in the state model.

The restricted flow of control in the model enables fault tolerant construction. Because users share spaces, dangling state from a crashed application can affect other users. Simple restarting of the space is impossible, for the same reason. Transactions ensure that the system never reaches an undefined state, even when errors happen during during construction. For fault-tolerant operation, all construction micro-calls are atomic, have at most a binary outcome (success or fail) and have an inverse call that reverts its action and that cannot fail. Transactions are introduced by logging sequences of calls and performing rollback when an error occurs in the middle of a sequence, the other calls are rolled back in reverse order. Logs are kept separately in two locations. Controllers are the principal owner, but spaces need
to locally keep a copy of all pending transactions that affect it, because they cannot trust controllers to handle all faults. Commonly, user code is the cause of faults in the first place. Buggy processes cause segmentation faults, leaving the kernel and devices with dangling state. When contact with an adjoining space is lost, all construction from that space is reversed.

To show that no hidden magic is glossed over, Table 6.8 presents all remaining calls that complete the low level interface. Besides RPC return values for success and failure, these include equivalents of ICMP control mes-

<table>
<thead>
<tr>
<th>message</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG_TRANS_CREATE</td>
<td>create a buffer</td>
</tr>
<tr>
<td>MSG_TRANS_SETNAME</td>
<td>add a name to a buffer</td>
</tr>
<tr>
<td>MSG_TRANS_DUP</td>
<td>create an additional buffer interface</td>
</tr>
<tr>
<td>MSG_STREAM_CLOSE</td>
<td>release a buffer interface</td>
</tr>
<tr>
<td>MSG_AREANEW_OPEN</td>
<td>create a memory mappable segment</td>
</tr>
<tr>
<td>MSG_AREANEW_CLOSE</td>
<td>close a segment</td>
</tr>
<tr>
<td>MSG_AREADUP_OPEN</td>
<td>import a mappable segment</td>
</tr>
<tr>
<td>MSG_AREADUP_CLOSE</td>
<td>reverse segment import</td>
</tr>
<tr>
<td>MSG_PIPENEW_OPEN</td>
<td>create a pipe and open it for output</td>
</tr>
<tr>
<td>MSG_PIPENEW_CLOSE</td>
<td>close a pipe</td>
</tr>
<tr>
<td>MSG_PIPEDUP_OPEN</td>
<td>open a pipe for import (cf. AREADUP_OPEN)</td>
</tr>
<tr>
<td>MSG_PIPEDUP_CLOSE</td>
<td>close a pipe 'import'</td>
</tr>
<tr>
<td>MSG_PIPE_READBUF</td>
<td>start listening on an open pipe</td>
</tr>
<tr>
<td>MSG_PIPE_UNREADBUF</td>
<td>stop listening</td>
</tr>
<tr>
<td>MSG_PIPE_WRITEMSG</td>
<td>write to a pipe</td>
</tr>
<tr>
<td>MSG_PIPE_WRITEBUF</td>
<td>make a pipe listen on a buffer</td>
</tr>
<tr>
<td>MSG_PIPE_UNWRITEBUF</td>
<td>stop a listening pipe</td>
</tr>
</tbody>
</table>

Table 6.6: Construction APIs: Buffer Actuators

<table>
<thead>
<tr>
<th>message</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG_PORT_SET</td>
<td>acquire a network port</td>
</tr>
<tr>
<td>MSG_PORT_CLEAR</td>
<td>clear a network port</td>
</tr>
<tr>
<td>MSG_RTADD</td>
<td>add a network route</td>
</tr>
<tr>
<td>MSG_UPDATE_ID</td>
<td>ask for a globally valid space ID</td>
</tr>
<tr>
<td>MSG_SIGNAL_PREFORK</td>
<td>prepare kernel state for a fork call</td>
</tr>
<tr>
<td>MSG_SIGNAL_POSTFORK</td>
<td>cleanup after the fork</td>
</tr>
<tr>
<td>MSG_PURGE</td>
<td>reset the space</td>
</tr>
</tbody>
</table>

Table 6.7: Construction APIs: Other Actuators
<table>
<thead>
<tr>
<th>message</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG_RPC_RESPONSE</td>
<td>default RPC response</td>
</tr>
<tr>
<td>MSG_FAIL</td>
<td>default RPC error</td>
</tr>
<tr>
<td>MSG_NOAUTH</td>
<td>access denied RPC error</td>
</tr>
<tr>
<td>MSG_PING</td>
<td>ICMP equivalent: echo request</td>
</tr>
<tr>
<td>MSG_CTL_TTLZERO</td>
<td>ICMP equivalent: echo reply</td>
</tr>
<tr>
<td>MSG_NOROUTE</td>
<td>ICMP equivalent: time exceeded</td>
</tr>
<tr>
<td>MSG_SIG</td>
<td>ICMP equivalent: destination unreachable</td>
</tr>
<tr>
<td>UPCALL_FILTER_FINISHED</td>
<td>message encapsulates signal</td>
</tr>
<tr>
<td></td>
<td>notify that a request has finished</td>
</tr>
</tbody>
</table>

Table 6.8: Construction API: Other

sages and two unique calls: one message forwards signals between spaces in absence of faster signal channels, another is an asynchronous callback to request MSG_FILTER_FINISH.

Application Construction

Construction encompasses a few actions not represented in the filter state transition diagram. To avoid unnecessary computation, overlapping sections in the runtime graph are shared. Also, even though Streamline is primarily a composition system of static code, it supports just in time compilation and code loading, because not all filters can be statically compiled.

Prefix Sharing  Computationally overlap in practice, because many applications reuse the same (library or kernel) pipelines. Prefix sharing is the reuse of runtime filter and stream instances for multiple requests. Filters especially recur often close to data sources and within common network stacks. This is fundamentally a subgraph identification problem that is implemented by recursively merging requests from their sources onward. The principle is simple, but we must be vigilant against data corruption. Shared filters implies shared metadata segments, which may not give the same result as parallel isolated filters. Two preconditions guarantee that metadata is only shared safely. A filter implementation may be shared among request graphs if and only if input streams overlap and the instance is stateless or in its initialization state. By default, if all these conditions are met and a running instance matches a filtername request, the name reifies to the running instance.
Dynamic Filters  Especially on embedded devices it is not uncommon
to have to manually load object code into an instruction store, because these
systems lack demand paging and multitasking. For instance, the constrained
‘microengine’ processors of the IXP network processor (that we discuss in de-
tail in Section 7.3) are geared specifically to packet processing. They can pro-
cess packets at higher rate than the control CPU, but are too small to run ar-
bitrary applications. Instead, they excel at carrying out small targeted tasks,
such as packet classification. Streamline can generate task-specific object
filters from safe, high level expressions in the FFPF Packet Language or the
Ruler [HvRB07] pattern matching language. Both languages also have Linux
kernel module targets for execution on systems besides the IXP.

Code Loading  To support this hardware, it is essential to integrate code
loading: the shipping of executables from a repository on the host to the de-
vice and performing necessary control operations (e.g. on I/O ports). Em-
bedded devices frequently ship with proprietary code loading software that
is not easily integrated into a comprehensive control system. Streamline per-
forms code loading by issuing a request for a code loading filter. The filter
usually wraps an external vendor tool as subprocess. The approach has the
advantage that object code is automatically shipped to the space holding the
loading facility through the messaging system: no separate shipping protocol
has to be implemented.

Compilation  Code loading onto embedded devices is often preceded by
a compilation step from a domain specific language (e.g., a regular expres-
sion). Again, an external tool is wrapped into a Streamline filter, so that the
controller can find and execute it. Then, compiled object code is shipped
through the pipeline from the compiler to the code loader without any addi-
tional effort on the part of the application. This is particularly useful as the
two tasks frequently execute in distinct spaces. Compilation is a userspace
task on the host, whereas code loading is a kernel task at best, or has to be car-
ried out from an embedded control processor. Alongside known languages
such as regular expressions and BPF, Streamline bundles two networking-
oriented domain-specific languages developed internally: the FFPF Packet
Language 3 [CGXB07b] for arbitrary packet operations and Ruler [HvRB07]
for packet and stream inspection and rewriting.
6.5 Fine-grained Access Control

Streamline safely opens up kernel and devices to user applications by extending Unix file permissions to these contexts and to the filters and streams that operate in it. Unix permissions commonly operate at the level of files and processes, but this is insufficient for Streamline, as filters can move between processes and streams can map across them. This section introduces access control at the level of filters and streams and shows that it can give the same level of protection as Unix, but also trade off isolation for throughput on a case-by-case basis. Specifically, it argues that many applications can reduce OS isolation without degrading actual threat protection. To give the same level of protection as Unix operating systems, Streamline has to

1. protect the kernel and shared peripherals from untrustworthy users. Specifically, it must avoid system corruption due to illegal instructions or data corruption; rule out service disruption due to resource monopolizing such as infinite looping; and prevent privilege escalation.

2. protect application data against snooping and corruption. The first is a privacy concern: users must be able to keep information private. The second is a weaker constraint: even if data is shared, applications may not modify it at will; write access has to be separate from read access and all writes have to be synchronized.

Philosophy

To maximize throughput, Streamline does not isolate processes only through protected memory and block copying. While it can enforce this behavior, we encourage filter offloading to protected domains and data sharing between domains to increase performance. This complicates the task of guaranteeing safety. For both system and data access, this section presents the control logic in Streamline, argues why it is as secure as regular Unix and shows how it can be relaxed on specific occasions to safely increase throughput. Before introducing technical contributions, we state our basic assumptions on security.

System Security Relaxing security constraints over Unix can increase pipeline throughput. If special purpose processors exist, offloading filters to these frees more contended CPU cores and generally increases throughput; if filters discard high ratios of data (as packet filters do, for instance), throughput will be highest when these filters operate as close to the data source as possible, since this minimizes communication overhead. Both opportunities can be exploited with the right access control mechanisms. We will introduce
security mechanisms that make it possible to selectively relax security only for trusted users or users of trusted filters. Trusted filters are based on safe languages that permit programmable access by untrusted users (because the language and interpreter or compiler are trusted) or safe templates that permit access when individual filters cannot be trusted, but whole pipelines can.

**Data Security** On many modern computers systems, more data can be shared safely than is current practice. One example is in network receive processing. Raw packets arriving from the network are by default not shared on most operating systems to maintain privacy. On a multi-user system where users cannot access the physical medium the OS indeed is a data access chokepoint. This configuration occurs with virtual desktops and in university computer laboratories (if cables are not exposed). Because shared systems were common in the past, it is understandable that the option is standard in operating systems. More and more, reception queues can be safely mapped, however. This is certainly true when (1) all data in a reception buffer is destined for the same application or (2) isolation is already pierced elsewhere. The first case holds on systems with multiple or multiring NICs. Even without such hardware, it is often the case: on dedicated networked servers that perform a single task, only administrator data is mixed in with application data. If all superuser traffic is encrypted, as for instance SSH sessions are, all network packets can be safely mapped read-only into the application context. The second case occurs when the network outside the host cannot be trusted, in other words: for all Internet connections. In this common case, then, reception queue isolation through copying incurs cost without offering real protection. Instead, rings can be mapped in one of two ways. Read-write mapping enables in place modification by applications, but poses a risk when multiple processes share a buffer. Read-only mapping saves the copy and avoids all security risks but privacy invasion. As few applications perform in-place modification of receive traffic and the physical network generally also punctures privacy, this second option can be employed as default. Streamline aims to enable system administrators to open up their system to data sharing and computation offload to exactly the level they are comfortable with. The remainder of this section introduces the tools that help them set safe policies.

### 6.5.1 System Access Control

Access control in Streamline derives from Unix permissions, but we have to revise the implementation to cover filter migration and data sharing. To achieve this, Streamline treats spaces as (active) access control subjects and filters, streams and shared buffers as (passive) objects. In line with Unix practice,
spaces have an identity made up of one unique user account and an optional set of groups to which the space belongs, denoted by a UID and set of GIDs. Space UID and GID derive from their controlling user for userspace processes. Kernel spaces are owned by the root superuser.

Objects are owned by one pair of user and group and have access permissions set for the owner, the group and all others. Streamline ports file permissions from processes to filters, for instance. Each filter implementation (the runnable code that ties a filter to a space type, e.g., the Linux kernel implementation of the deflate algorithm), has an owner and set of permissions. Both are in principle hardcoded by the administrator. Permissions limit execution of filters by user, group and globally using the Unix UID and GID.

Because filters have full access to their memory protection domain, the most important reason to block execution of a filter in protected domains is that it cannot be trusted to perform only safe operations, where the vulnerability can be faulty behavior or data leakage. Through permissions, administrators split filters in two sets: those that can be trusted in the hand of untrusted users and those that cannot. By default, the filters that ship with Streamline are owned by the root. Except for data sources, they are all trusted to operate safely and are therefore executable by all users. Source filters can leak data; we return to this point when discussing data access control. Irrespective of our choices, system administrators ultimately control the system wide defaults.

Besides blocking filters that are unsafe, administrators will further want to limit access to filters that are computationally intensive, or monopolize scarce resources (e.g., an FPGA) to users of a privileged Unix group such as the wheel group (which gives superuser access in Unix through su). For other users, the lack of access will cause a drop in performance, but not necessarily more. If less efficient alternative implementations are available, this will be transparently substituted. For example, if a hardware MPEG decoder is blocked, the pipeline falls through to a software implementation. This stands in contrast to regular Unix operation, where certain administration tools (e.g., tcpdump) can only be executed with superuser privileges.

To completely guard the system against resource overuse, limits have to be set and enforced at the language level as well, such as bounds on the number of open streams and filters per user. Loop prevention for certain classes of users would make another logical language-level policy. Streamline does not currently enforce this kind of resource control.
Safe Languages

Untrusted users can be safely given access to protected domains through safe filters, but as the tcpdump example indicates, not all applications can be built from fully precompiled filters. An alternative that Streamline supports is to support safe languages: languages that cannot express any unsafe operations. Applications written in these languages, even if written by malicious users, can be trusted. They can be thoroughly checked when the source code is interpreted at runtime. BPF and Java are two very different examples of this approach. Streamline has a BPF filter in kernel space.

Interpreted filters are generally at a performance disadvantage compared to compiled equivalents, however, and embedded devices are not always powerful enough to run a heavyweight interpreter. Streamline can also execute filters in a safe compiled language, the FFPF Packet Language, or FPL [CdBB05]. FPL programs cannot directly access heap memory and are therefore safe with regard to arbitrary memory corruption. The set of operators is also limited to eliminate recursion and infinite loops. Resource control does not extend beyond these rather crude measures. Scout [MMO+94] presented more strict resource control and QoS for kernel streaming I/O that would be equally valuable in Streamline. To safely execute a compiled executable, Streamline has to be able to trust that this binary object is indeed equivalent to the safe source program. Trusted compilers offer such safety guarantees through signed compiler statements [BS02]. Streamline does not currently include this kind of admission control, but has done so in the past; we will elaborate on this point shortly. Instead, it integrates the compilation and code loading steps to ensure safe provenance of user-developed filters. The main insight is that by applying the access control policies to compilers and loaders, the system administrator can restrict which users may insert which types of dynamically generated filters.

Safe Code Loading

In compiled safe languages, trust derives from the compiler. Streamline integrates the compiler and code loader as filters, so that these protected operations are subject to the same access control framework that governs all applications. Besides, inclusion gives the benefit that invocation of both programs as well as movement of object code between them is simplified. A filter is compiled and loaded by executing the simple pipeline

\[
\text{compile | load}
\]
whereby the compiler filter accepts given high-level language and the loader accepts the low-level binary code generated by the compiler. It is up to the user to select matching filters. Ownership of the new filter falls to the executing user. Ownership and permissions of the new filter are set by the loader as it registers the new filter. Since users that can compile from the given language and that can execute on the chosen hardware will be able to replicate the filter, they might as well reuse it. All other users are forbidden from using the language or the hardware, each of which is sufficient reason to block access to the new filter. To avoid privilege escalation, therefore, access permissions must be at least as strict as the combination of those of the compiler and loader that the user ran. Default is to make the new filter only accessible to the current user. Loaders can allow this choice to be overridden through parameters, but cannot extend beyond the permissions of the loader itself.

**Safe Templates**

With individual permissions for the compiler and loader alone, it is not possible to enforce that untrusted users must use the compiler and the loader. Streamline enforces such rules through *safe templates*. Templates are rewrite functions from one pipeline to another (Section 6.2.1). By attaching permissions not just to filters, but optionally to templates, Streamline allows untrusted users to load safe pipelines of otherwise unsafe filters. The compilation pipeline is one use case. Another very common example for this practice is socket handling. Sockets require access to protected data reception and transmission, but regular users may not be given unchecked access to the data sources. With safe templates, it becomes possible to give access, but enforce that a port-specific filter is appended to the source that filters out all data to which the user has no access. In the next section on data access control, we will see special “copy” filters that use safe templates in this manner.

**Effects on Isolation and Throughput**

Streamline can assure system integrity to the same level as Unix. If kernel and device spaces are inaccessible to all users, then it reverts to a fully sealed operating system. This solution impairs operation, however, because it also forbids commonly supported I/O tasks, such as socket networking. Trusted filters, safe languages and safe templates together make it possible to close off most of the system to most users, but to allow restricted computation (e.g., for cryptographic offloading) and data access (e.g., for sockets, we return to this point).
6.5.2 Data Access Control

Data privacy and integrity cannot be governed by permissions on passive filter implementations alone. At runtime, streams can be mapped into multiple spaces to maintain high throughput. To protect live data, permissions are also attached to streams. Contrary to filters, these permissions are not attached to passive implementations, but to active stream instances. Each stream has an owner and access permissions for sharing. Permissions do not have to be set manually, but are generally calculated recursively. Data leaving a source filter is by default owned by the user instantiating the source. Default permissions are the same as those on the source; this does not weaken security, as each user that can execute the source can access its output anyway. From then on, each filter derives permissions on its output from those on its inputs. To preclude data leaks, output permissions have to be at least as strict as those on each input. Therefore they are computed as the logical intersection (bitwise And) of all inputs. For example, from the set of input permissions (in octal notation) 0640 and 0444 the derived permission 0440 is calculated.

The recursive calculation of permissions simplifies access control. When a user finds default permissions unnecessarily restrictive (and they degrade throughput by needlessly preventing sharing), users can override the algorithm by annotating filters with a mode parameter. Even though the parameter is attached to a filter, it only affects data produced by the filter, not the implementation (which the user executes, but may not own). It will affect the permissions of downstream data, however, whose permissions are computed from this replacement value. There is only one other exception to the rule that permissions are at least as strict as the permissions on their input. We will see shortly that to allow restricted data export from one space to another, the permissions on data can be changed by copying data between buffers.

Shared Buffers

The use of shared buffers introduces security perimeters independent of virtual memory protection domains. As we explained in the communication section, buffers are mapped into a domain once for the duration of use. Protection is enforced per 'buffer plus computation space' pair. Streamline checks permissions only once per pair: just before mapping a buffer into a space, which can occur during a call to open or as result of a buffer-fault (similar to a page-fault). Because this is the only occasion when computation spaces are given access to a new security perimeter, it is a security 'choke point' and thus acceptable as the sole policy enforcement gate. Policy enforcement dif-
fers between computation spaces and depends on hardware capabilities. On multitasking CPUs, the memory management unit enforces read and write permissions for each buffer. With IOMMUs, the same can in principle hold for buffer access from devices (but is not implemented). Restricted device access is particularly important in the case of Streamline, where device logic can execute user filters, which are clearly not part of the trusted computed base.

Blocks from multiple streams may share a buffer; like streams, buffers must therefore have an owner and set of permissions, that – for safety – must be at least as strict as those of the streams it contains. Shared buffers complicate access control bookkeeping, because streams may have different permissions. The approach does not weaken security in itself, because technically speaking each stream – even each block – can be given a private buffer with matching ownership and permissions. That solution will simply be slow, however. Instead, buffers inherit permissions from their streams. To minimize data copying and virtual memory operations we increase sharing between filters from the safe base case of ‘one stream, one buffer’ as follows: First, streams with identical ownership and permissions can be safely merged. Second, streams that belong to the same group and that grant the same permissions to their group as to their owner can be co-located. Finally, streams that are accessible by all can be shared. As a result of the recursive permission algorithm, commonly only few stream permission groups exist. Unless users override the algorithm explicitly, there can be no more than the number of sources. These, in turn, inherit from their users, limiting the maximum to the number of active users. If users enable sharing with other group members or the system at large, only a handful of security perimeters remain. As a result of using Unix access permissions, it is impossible to isolate data access between multiple applications of the same user; if further isolation is needed, a user will have to create multiple accounts or not execute tasks simultaneously.

Stream Sanitization

The model of application paths that extend into the kernel collides with the practice to disallow access to raw network traffic. Commonly, users consume network data from known safe kernel endpoints (such as sockets), not from the raw network interface cards. Streamline reconciles these conflicting models by granting regular users access to trusted safe templates consisting of sources and other filters, but not necessarily to the sources by themselves. To avoiding giving users access to raw traffic, an administrator can force users to copy the substream to which they have access out of the larger stream into a
buffer under user control. Trusted “copy” filters copy data and set the permissions on their outgoing streams to that of the destination buffer. Copy filters are clearly identifiable data leaks and administrators can set permissions on these trusted filters accordingly. In line with common practice, Streamline contains a copy filter that only passes data belonging to a socket registered by the application’s user. This model maintains strict isolation, while enabling safe kernel data access by regular users. On traditional network stacks, such as that of Linux, only the superuser may run a packet filter in the kernel, because there is no way to limit leaking of all raw traffic. In Streamline, all users may insert BPF or other filters into the kernel, as long as these attach to the output of safe “copy” filters. Administrators encode this constraint in a safe template.

Effects on Isolation and Throughput

Streamline can offer data isolation on par with Unix. Even though security perimeters protect entire buffers instead of individual blocks, it can enforce protection of arbitrarily small data sets by creating a buffer for each unique access control group and user. Pathological behavior, whereby each block has a different user, reverts to per-block allocation because each block will have a private buffer. Compared to raw blocks, the shared buffers in Streamline are heavyweight structures and thus more costly to allocate. Therefore a coarse-grain policy is only practical in situations with few security groups. We have seen that this is the common case.

Stream sanitization is needed to offer the combination of data separation and functionality of Posix operating systems. As a result of using shared buffers, Streamline can directly map buffers containing, e.g., raw network traffic packets into processes belonging to unprivileged users. Operating systems normally copy data between kernel and userspace in small blocks to maintain isolation between spaces. Strict export policies can trigger this (functionally superfluous) copy in Streamline by enforcing safe templates that embed “copy” filters in the pipeline. If sharing is deemed safe, instead, the copy is avoided. Application developers do not have to be aware of these policy details.

6.5.3 Policy-based Access Control

Streamline purposefully stays close to Unix access control. This choice limits the security policies that it can express. Streamline:FFPF originally shipped with a more expressive trust management system that can, for instance, express rules on filter resource control by interpreting certificates from trusted
compilers. The same admission control can easily be ported to present Streamline. We paraphrase the original description [BP04] before comparing the merits of the two access control designs.

In Streamline:FFPF, admission control is implemented as a stand-alone daemon. At instantiation, the complete pipeline is combined with authentication and authorization information and sent as a whole to the kernel module. The kernel module pushes the instantiation request on a queue that is read by the admission control daemon. The daemon reads the request, compares the pipeline with the user’s credentials and the host’s security policy and returns a verdict (‘accept’ or ‘reject’). Credentials and trust management in Streamline:FFPF are implemented in KeyNote [BFIK99].

The admission control daemon also provides more powerful authorization than filter access permissions alone. For instance, it is possible to specify for a specific user that a pipeline will be accepted only if the number of calls to a specific operation in an FPL filter is always less than the number of calls to another. Consider for example, the following safety policies (1) a call to a string search operation is permitted for packets arriving on NIC-1 (but not for other NICs) only if it is preceded by a sampling function, (2) all calls to a function produce_end_result must be followed by a return statement, (3) if no call is made to an external sampling function, an FPL callback operation should wait for at least 1000 packets.

Unix file permissions are less powerful, but more familiar to most users than the fine-grained policy-based control of Streamline:FFPF. While the policy language can express more detailed policies, it is also more cumbersome in use (involving trust management, policy specification and delegation, and trusted compilers). The advanced options are lacking in Streamline, because they develop few opportunities to safely increase throughput, while the policy language poses a barrier to uptake. Instead, Streamline is limited to coarse-grain policies on whole filter operation, such as that anyone may run BPF filters in the kernel, but that only users of group `wheel` can load home-grown filters into the kernel.

6.6 Summary

Careful selection of filter implementation and location improves end-to-end pipeline throughput. Because the state space is large and the optional choice not always readily apparent, Streamline automates the optimization process. The control application maps a digraph model of the application onto a digraph model of the environment in all feasible combinations and selects the one that gives the highest end-to-end throughput with the lowest cycle cost.
The control application translates this mapping into a running task by issuing construction requests to the applicable spaces through a message passing network. Construction is a fault tolerant process and integrates dynamic filter generation. Kernel and device operation is secured by extending Unix permissions to filters, streams and underlying buffers. If needed, the exact isolation policies of Unix can be enforced. More relaxed operation increases throughput by selectively enabling filter offloading and data sharing. Safe languages, safe templates and stream sanitization increase the range of safe relaxations.

Together, the ideas presented in this chapter contribute to the central goal of a high-throughput, flexible I/O system by automating the optimization step that is required of any flexible (in the sense of, adaptive, accommodating) system. The security design ensures that the system does not simply trade in security for throughput or reconfigurability (although it does allow this in restricted cases).