The computation layer, or *dataplane*, executes filters as fast as possible. The core element is a scheduler that continually pushes filter onto a runqueue as input arrives on incoming arcs. To implement computation spaces, Streamline extends this model with multiprocessing support. It improves throughput by limiting synchronization between and within spaces (5.2), by isolating data access between processes while minimizing locking and copying (5.3) and by identifying parallelizable sections on multiprocessors (5.4).
5.1 Overview

The dataplane extends Unix pipeline processing to kernel and peripheral device tasks and speeds up execution to render it suitable for high performance tasks. Code is for the most part encapsulated in independent, reusable filters: only scheduling logic has to be centralized. Streamline refines the Unix custom of using pipes as synchronization primitive in three directions.

Signaling (5.2) contributes to application cost in the form of task-switching, interrupting or polling (as Section 7.2 demonstrates). Streamline must forward events between userspace, kernel and device tasks. It minimizes event handling cost in two ways: it batches the number of events sent between spaces by batching them and it reduces the cost per event by replacing IPC with user-level threading and function calling where possible.

Write Isolation (5.3) is required to avoid data corruption with concurrent users. Traditional pipes have only two users with well known access policies. Streamline allows concurrent read access to data streams, metadata streams and scratchpad memory by many filters and applications to minimize copying. It isolates write access by scheduling writes after reads. It reduces write cost by replacing data operations with metadata equivalents and by replacing editing with append operations (that do not require a copy).

Parallelism (5.4) grows in importance with the number of cores per chip. Existing schedulers do not exploit structural pipeline information. Streamline extracts data parallelism by identifying stateless and other well behaved filters and extracts task parallelism by exploiting the composite graph structure.

5.2 Signaling

When data arrives on an input port of a filter, that filter must be scheduled swiftly to minimize end-to-end processing latency and to maximize likelihood of serving data from a fast cache. Filter scheduling queues are updated continuously, because execution flow depends on the classification by upstream filters. Streamline integrates an event network that spans all spaces. It does not reuse an existing solution, because to the best of our knowledge no off the shelf notification system spans across userspace applications, kernel tasks and peripheral devices. Besides this unique demand, functional requirements of the event network are simple: it must efficiently forward wake
Signaling up events from filters to their downstream neighbors. Because the Streamline architecture differentiates between computation and communication concerns, the dataplane is only responsible for passing notifications between filters, not for making data itself available across spaces. An event network forges compile time scheduling optimizations such as inter layer processing [CT90] to expose all configurability at run time. A reconfigurable design inevitably introduces data hand off, or signaling, overhead. The assumption behind application tailoring is that this cost can be offset by adapting execution to match local circumstances, such as the number of parallel hardware threads of execution.

Signaling cost can be curtailed independent of application tailoring. The main source of overhead is in the form of task switching. On a single core as a result of scheduling decisions, on a multiprocessor in response to hardware interrupts. A task switch is not free because it must write CPU state to the process stack and flush L1 caches and TLBs $^1$. Depending on the depth of the processor pipeline, size of its register file and number of active execution units (floating point, SSE), tasks switches can take up to a thousand cycles. This upper estimate corresponds to the popular x86 processors and does not yet take into account the cost of cache effects. A quick calculation teaches us that a 3 GHz system that switches tasks at every event reaches 100% CPU utilization at around 3 million events per second, the number that even a single 10 Gigabit network card generate. Duplex traffic can swamp a modern dual core CPU even before any functional processing is applied. Synchronization cost can be significantly reduced, however. 10 Gigabit NICs habitually implement event batching, a technique to reduce the number of events at high rate without significantly affecting latency. With event batching, the number of events sent is batched at high rate by specifying a minimum timeout between events. Hardware has to support batching by supplying large buffers that can hold multiple data items. Software must be modified to process as many items as possible, not just as many as the number of received events.

To minimize event handling cost, Streamline implements (an adaptive variant of) event batching along with two other optimizations. It (1) executes multiple filters within a single hardware task to minimize the cost of a single event, (2) batches events between tasks by a cache-dependent factor and (3) co-schedules filters to access data from shared data caches. This section covers the first two optimizations. Parallelization is discussed independently in Section 5.4.

$^1$in general and especially true for most x86 processors. Architectures differ, however. Even within the x86 exceptions exists: the Core i7 supports TLB tagging to reduce flushing.
5.2.1 User-level Threading

Fine-grained pipelines hand off data so frequently that forwarding all events using task switching imposes an unacceptable cost. The overhead is reduced by around two orders of magnitude when filters hand off data through local function calling. Streamline schedules not filters, but empty shell processes hardwired with a filter call stack, which we call *container tasks*, and pushes filters onto the stack in accordance to a parallelization policy chosen by the control system. Based on a conceptual split between software capsules (filters) and hardware tasks (processes), this approach is a special instance of userlevel threading [MSLM91] – a technology that is frequently used to avoid task switching; it was recently also advocated in the context of manycore systems [SATG+07]. Local function calling can replace full task switching in all but two situations. If one filter accesses a DBuf that another may not touch, they cannot share a container task (unless sanitization can be enforced, as explained in Section 6.5.2). Second, if two local filters share a stream that filters or client applications in other tasks must also access, task scheduling between the local filters can be avoided, but metadata must still be stored in an IBuf.

Within a single space, Streamline calls filters in rapid succession for each datablock. Conceptually, it pulls each block through the entire graph as fast as possible before spending any time at other blocks. The approach minimizes latency and maximizes DCache hitrate at the cost of the ICache. When a filter produces data, the runtime system places all downstream filters that share a space on the call stack of the current container task (and sends events to the other tasks that contain dependent filters). This recursive algorithm results in a depth-first traversal of the (local part of the) graph. For graphs with filter codesize smaller than accessed data this is indeed the most efficient solution, because the combined instructions will fit in the ICache more easily than the set of records in the DCache. In our experience, the majority of filters are indeed very small, validating the heuristic. StreamIt [STRA05b] is more refined in this regard: it quantitatively assesses working set size and -overlap to make a case-by-case informed decision.

Event avoidance through function calling is applied to all spaces in a pipeline, resulting in a graph with the minimally required event count. Events are only sent when bridging spaces and metadata is only stored in IBufs when any non-local filters or clients require access. This approach combines the advantages of two worlds: it renders performance comparable to monolithic processes for the majority of streams that are only of local interest (as shown below), but makes all streams available to inspection by any process in principle. The control filesystem (Section 3.2) exploits this property to make kernel
I/O transparent.

Having as many filters as possible share hardware tasks trades isolation for performance, which has obvious stability implications. A composite system can provide the best of both worlds, because it can isolate buggy or un-trustworthy filters into private tasks at any moment. Streamline by default mimics a monolithic OS to maximize performance, but it can equally well implement a microkernel design or any other model. Streamline can at will migrate kernel operations to userspace processes to mimic microkernels with their advantages (isolation) and disadvantages (synchronization cost). Inversely, it can move all processing out of OS container tasks to directly connect peripherals and application tasks for low-latency operation. When filters can migrate between trusted and untrusted environments in this fashion and access control is enforced at the finer grain of filters, the traditional view of strictly separated kernel and application domains loses meaning.

Results  To quantify the benefits of user-level threading, we compare Streamline directly to Unix pipelines. We execute a very simple byte matcher algorithm zero or more times on both systems. To also learn the upper boundary of achievable throughput we also execute a handwritten C program that performs the same task in a five line for-loop. All pipelines process 1KB blocks as fast as possible. The machine used is the same as for all the other tests (Section 7). Figure 5.1 shows the results of this test. To demonstrate IPC overhead, it shows throughput when the first byte in the block is matched and no further bytes are touched (in logarithmic scale). The figure contains error-bars, but due to the huge y-range, these are immaterial. All upper and lower quartiles were within 12% of the median of 11 runs.

We show two kinds of Streamline pipelines, sslow and slfast, to honestly compare with the others. Sslow performs the exact same actions as the Unix pipeline: it writes a block to a buffer, processes it and then reads it from another buffer. Because the write and read operations block, no event batching can be applied. Contrary to Unix, Streamline only uses a single DBuf and IBuf pair regardless of the number of executing filters: one for the write and one for the read. Slfast, on the other hand, is comparable to the 5 line C snippet. It processes blocks directly, but never copies them. As can be expected, this gives it considerably higher throughput on the match case. Here, the four methods are all about an order of magnitude apart. We also see that sslow throughput is almost linear, because most cost lays in the write and read calls at the start and end of the pipeline. The figure shows that when explicit copying is not functionally needed, user-level threading gives a 100x throughput increase over copy-based Unix IPC. When it is, the savings de-
pend on the length of the pipeline, as indirection savings accrue with each hop. Not shown is that as data access per call increases, the advantages are reduced. With 1000B reads, savings decrease to between 1.5x and 2x.

5.2.2 Event Moderation

Function calling is not possible across tasks. An event has to be sent to another container task that tells it to put a filter on its call stack. Such cross-space communication involves synchronization, in the form of hardware interrupts, polling or task switching. Existing synchronization interfaces between userspace and kernel are known to introduce non-trivial cost as a result of synchronizing at each call [Ous90, MB91, FT96]. Like others [TS93, NAP], Streamline reduces overhead by relaxing synchronization constraints. Signals are batched by the sender and handled lazily by the receiver. Both methods are actively managed to stay within application latency bounds.

Streamline combines interrupts and polling to minimize signaling cost across a range of data rates. The two signaling approaches are inefficient at opposing end of the spectrum: polling wastes cycles when traffic arrives below polling rate; interrupts are inefficient when event rate is so high that most CPU time is spent switching instead of performing useful work (‘thrashing’). Maximizing efficiency at all speeds calls for a hybrid solution. Various hybrid solutions have been proposed in the past, such as clocked interrupts [TS93],

Figure 5.1: Unix and Streamline pipelines compared to a single thread
temporary interrupt masking [NAP] and batching up to a maximum delay (e.g., Intel GbE NICs) or queue length (e.g., IBM FC 5700 NICs).

Streamline combines event batching with timeouts to amortize cost at high rates while bounding worst case latency. When a signal request is received, it does not actually send the event, but only increments a counter until that reaches a given event threshold \( T_b \). If this is the case it sends a single event and resets the counter. To guarantee that events always arrive within a set latency bound it schedules a timer to ring after \( T_t \) ticks each time the counter is first incremented. If the timer goes off before a batch event is sent a timer event is sent instead. Otherwise, the timer will be cleared and reset by the first new increment. The approach amortizes cost at high rates while bounding worst case delivery latency.

Uniquely, Streamline tailors event batching to each stream. The maximal acceptable latency \( T_t \) depends on application goals and is derived from request options (Section 3.3.3). The maximal batch rate \( T_b \) is strictly limited by the length of the underlying data buffer and a lower soft threshold is given by DCache size: to ensure that data is still in the cache after an event, the event must be sent before the producer overwrites the item in the cache.

Results The effect of event batching can be significant. Figure 5.2 shows throughput for a Posix compliant pipe under different event thresholds. The figure shows a large buffer of 16MB, or 8000 2K slots. This gives a maximum queue length for batching of 4000, which the system floors (counting in binary, for technical reasons that have no relevance to this test) to 2048. The latency constraint is set to 1ms, which means that at least 1000 events are always scheduled per second. Throttling increases throughput, but at diminishing returns. As queue length \( Q \) grows, the chance of hitting the event threshold reduces, while the change of hitting the latency threshold remains constant. The figure shows that, indeed for \( Q = 32 \) we already achieve 92% of the gain of \( Q = 2048 \): 2.36x baseline compared to 2.56x. For this reason, we chose 32 as default queue length.

5.3 Write Isolation

To avoid data corruption when editing shared buffers, write operations must be isolated. Streamline is built on the assumption that most filters pass data unmodified and that as a result many unnecessary copy operations can be safely avoided by reusing the same data copy among filters. This approach contrasts with Unix, where processes manually copy data from input to output pipes if they wish to forward traffic unmodified. Unix takes a safe de-
fensive approach by enforcing copies at each step, while Streamline takes an optimistic approach by avoiding copying unless expressly warranted. This section explains how Streamline can support all kinds of data modification with greatly reduced copying compared to Unix pipelines.

5.3.1 Copy Avoidance

Streamline avoids the defensive data copy for a filter if one or more of the following statements hold:

1. The filter will not edit the data
2. Only the filter’s descendants access the data after editing
   (and no one accesses the data during editing)
3. Editing can be replaced by an append operation
4. Editing can be replaced by a metadata operation

The first case is trivial. The second ensures that changes by a node are isolated to the subgraph downstream of it, while allowing for two optimizations: copying is not required for linear pipelines (where isolation with respect to task parallelism is not pertinent), nor when parallel read-only subgraphs can be executed before the edit is performed. The third case replaces an edit operation, which consists of an allocation plus a copy if direct access cannot be granted, with an allocation only. Filters that compress data, for instance,
change data beyond all recognition. In that case, they may just as well write to a new block if this is cheaper. Both appends and edits leave the original data item intact if others access it and operate on a physically different block elsewhere in the buffer (similar to copy-on-write). If none (or little) of a data item is retained after the edit, a full copy constitutes waste. Finally, the fourth case exploits the fact that, at 16 bytes per tag (Section 3.3.2), metadata is small and therefore cheaper to access and duplicate than data. It replaces direct data editing with updates to metadata values, such as offset and class. This class of filters can reorder data logically without touching it physically. Streamline implement a form of TCP reassembly in this manner; we discuss that example in detail shortly.

5.3.2 Application

Selecting a copy avoidance strategy requires understanding of a filter’s data access behavior. In some cases, Streamline can infer the most efficient strategy from system state. For instance, it knows how many filters access a given stream and can therefore decide whether to grant direct write access or not. On the other hand, it does not know that a filter requesting write access will compress data and can therefore append instead of edit. For this case, it needs help from the developer. Before explaining how Streamline implements the optimizations themselves, we therefore first categorize filters according to their data access patterns. From the more than 40 filters in the reference system we deduce the seven structural classes summarized in Table 5.1.

<table>
<thead>
<tr>
<th>Class</th>
<th>Metadata Access</th>
<th>Data Access</th>
<th>State Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistics</td>
<td>R</td>
<td>R</td>
<td>R/W</td>
</tr>
<tr>
<td>Inspection</td>
<td>R</td>
<td>R</td>
<td>R/W</td>
</tr>
<tr>
<td>Classification</td>
<td>R/W</td>
<td>R</td>
<td>∂</td>
</tr>
<tr>
<td>Reordering</td>
<td>R/W/A</td>
<td>R</td>
<td>∂</td>
</tr>
<tr>
<td>Editing</td>
<td>R/W/A</td>
<td>R/W/A</td>
<td>∂</td>
</tr>
<tr>
<td>Source (zero-copy)</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Filter classification by data access (A: append-only, ∂: optional).
Statistics The simplest filters only read metadata tags. They do not observe streams directly, nor do they modify tags. As they have no influence on streams, they must affect state to perform any useful operation. This (small) class includes counters and histograms (e.g., of inter-arrival times).

Data Read

Inspection The information that can be extracted from the metadata stream alone is limited. Inspection filters also have read-only access to data streams to make more interesting observations. Intrusion detection algorithms are advanced examples, but the class also encompasses string search, regular expression matching and application specific accesses, such as HTTP request identification. Since these filters do not update stream contents, multiple such filters can safely access a single data copy.

Metadata Write

Classification Classification filters also analyze stream contents, but use the information to update the class field in the metadata tag. Data access is again read only, but metadata access must be read-write, to update the tag. Combined with stream classification (Section 3.3.2) this techniques enables data dropping: the first, very restricted, form of manipulation.

Reordering Dropping is not the only form of manipulation that can be performed solely with write access to metadata. Reordering modifies streams by moving the offset and length field embedded in metadata tags and by removing and inserting tags. The approach is best explained by example, for which we look at TCP reassembly: besides a classical implementation, Streamline implements a zero-copy reassembly filter. This version does not copy contents to recreate contiguous data, but edits metadata tags to present data in order and without protocol headers. The filter discerns between streams through connection multiplexing (Section 3.3.2), by selecting a different class for each TCP stream. It updates offset and length fields to crop headers. TCP reassembly involves buffering items during out of order arrival. Because data is already stored in a shared DBuf, the zero-copy reassembly filter eschews all additional data buffering, including that which normally occurs during out-of-order arrival. If a gap occurs, it only buffers rich pointers. As soon as the gap is filled, all queued rich pointers are flushed at once. As a result, in some calls the filter returns no records (when a gap is discovered and data must be held in a reconstruction window), while in others it returns multiple (when
said gap is filled and queued blocks can be released). The implementation is as follows: in the first case the filter communicates to the dataplane (by selecting a special invalid class) to forgo data forwarding, in the second it manually calls the zero-copy bootstrap interface that we discuss shortly for each queued item. The SafeCard example application has to process high bandwidth TCP traffic. It stores data in a single data ring and uses zerocopy TCP to present a sequential data view to its various filters.

Data Write

Editing  The previous classes show that many common edit operations can be performed with read-only data access. Metadata updates cease to be useful when blocks are thoroughly transformed, however, for instance in the case of encryption. Editing filters must be able to modify data directly. This does not necessarily mean that they require read-write access to the underlying shared DBufs, however: a private copy stored in another DBuf suffices. Because there is little relative performance gain in sharing metadata read-only when data is shared read-write and because some editing filters want to update the class tag, metadata is also shared read-write.

Source  Sources inject new data into the graph. In practice, they are commonly callback routines that act on external stimuli, such as the soft interrupt handler of the Intel e1000 device driver. Streamline exports a minimal bootstrap function for these handlers to call, which takes only two parameters: a locally valid pointer and a length value. On return from the call, Streamline has integrated the data into its buffer network and has prepared an initial metadata tag. The calling process can immediately release its copy of the data.

Zero-copy Source  Storing data in a DBuf is an expensive unnecessary operation if data already exists in a form of long lived (compared to stream lifetime) storage. Besides the straightforward bootstrap interface, Streamline presents a zero-copy interface. Here, the source registers an application-specific buffer as DBuf specialization once at initialization time initialization time and consequently only has to write tags to the metadata stream at runtime. Zero-copy reception is described in detail in Section 4.5.6). Strictly speaking, this filter class belongs under metadata write, but we delayed explanation to group it with the other source API.

As corollary to sources, one could also expect a class of sinks. We briefly mention that this does not exist for the simple reason that inaction does not
demand an API. Sinks can be spotted by looking at the type of a filter's outgoing ports. If it is null, the filter does not produce streaming data.

### 5.3.3 Implementation

Copy avoidance is only partly automated, because it is unfortunately infeasible for the runtime system to infer all intentions automatically. Instead, it presents filters by default with a conservatively shared view together with interfaces for acquiring additional permissions. The filter developer can use these to choose between data and metadata manipulation, for instance. Similarly, state has to be explicitly requested. By default, a filter has read and write permissions on metadata, read permissions on data and no state. This choice follows from the observation that metadata updates are cheap and frequent, while data updates are expensive and – thanks to the metadata channel – infrequent.

**Metadata** Safely granting write access to metadata necessarily involves making private copies of tags at task-parallel splits. Because tags are only 16 bytes long, the cost is not excessive – not worth implementing complex copy avoidance algorithms. The only optimization Streamline employs is that it reuses tags in (linear) pipelines.

**Data** The higher cost of data copying calls for more aggressive cost saving. Filters may not modify data arbitrarily even when they have write access permissions on the underlying buffer. To modify a block, they must explicitly request write access for each edit to each block. The approach follows from the assumption that data edits are rare and that therefore the cost of an extra gateway function call at each write remains negligible. To request write access, filters call a function similar to the pseudocode definition

\[
\text{rich_ptr} = \text{slblockWritable(rich_ptr, copy\_contents)};
\]

When called, the runtime system identifies software (concurrent task-parallel claims) and hardware (virtual memory protection) state and returns an index that points to a writable block. The second parameter communicates whether on return this block must contain the contents of the current record or whether a memory region in undefined state suffices – in other words, whether the client wants to edit the contents or only overwrite the region.

Streamline replaces edits with appends when that saves a copy or when it is dictated by access control. The gateway function can do one of two things. If the filter has write permissions for the underlying DBuf and no one will access the block apart from the current filter and those downstream of it,
the system returns the original block. Otherwise, it allocates a new block, copies over contents if `copy_contents` is set and returns a rich pointer to the new block. The new block is not necessarily allocated from the same DBuf as the original block. Streamline always allocates from a local, write accessible, DBuf to ensure that edit operations always succeed, even if the process has only read permissions on the original buffer. This strategy also speeds up operation in NUMA systems where accessing the original buffer takes longer.

**State** Streamline tightly controls all memory allocation to be able to instantiate filters in any space and to be able to parallelize them. Having private global state causes problems, because it forbids concurrent execution of the same filter to extract data parallelism (as explained in the next section). Streamline forbids heap allocation by filters to keep them free from memory address restrictions. It can solve the stateful processing problem in two ways: by outlawing all state or by making state accessible for checkpointing along with the filter. We review both options.

Stateless stream programming trivially avoids problems. Stateless filters, or *computation kernels*, can be safely spread across distributed resources. Stateless stream programming is widely popular from large scale distributed to small scale embedded environments (e.g., for programming the graphics processor pipeline), because it guarantees that filter execution is free from side effects. It is this same point that makes pure functional and logic programs available to automatic parallelism extraction. The whole group of languages sharing this property is frequently referred to as declarative programming (although that term is contended [AS96](1.1.7)).

The application classes show that computational kernels are too restrictive to implement all practical filters. Streamline allows stateful processing, but only in a controlled manner to maintain control over checkpointing, i.e., definitely not through a byte addressable heap. We observed three uses of stateful processing:

1. to store data private to a filter that must persist across filter invocations
2. to store data private to a filter session, as identified by the class value
3. to communicate between filters and applications

**Private State** Although filters can acquire state using operating system or library calls (such as `bzero` or `malloc`), those interfaces make filters non-portable. As alternative to heap allocation, the runtime system presents a higher level interface to relocatable memory regions. Like ring buffers, random access memory region can be requested using the `open` Posix call. The
runtime system automatically manages allocation, virtual memory remapping and destruction of regions. This tight control has two advantages. It prevents accidental conflicting use of global variables by multiple instances of the same filter implementation. For performance reasons, filters are given raw access to their memory regions through pointers. The runtime system automatically resolves the descriptor into a locally valid memory pointer and stores this pointer in a structure private to the filter. Because the runtime system knows the extents of the data region, it can move data and update the cached pointer. Therefore, filters may not create additional copies of the pointer.

**Session State** Streamline handles the state multiplexing required to split filter execution between sessions. With a normal allocator, filters need to manually divide a region into multiple segments; when stateful filters multiplex streams based on class identifiers (Section 3.3.2), Streamline automates the division. For this special case, the runtime system implements a session lookup table that relates each class to a region plus offset within this region. Prior to scheduling a filter it updates the filter’s cached pointer to point to current session state. The implementation is straightforward, but centralizing the logic has non-trivial benefits, as we shall see in the section on parallelism. The number of sessions is not limited in principle, as Streamline can allocate regions at will. In practice, the shared region currently has a fixed number of slots. If a filter runs out of slots, the allocator fails.

**Communicating State** For communication, two or more parties must be able to access the same region. Here, the open call proves well suited. Its first parameter passes a key in a system wide namespace: commonly a filepath. Because Streamline backs data in memory, the scheme implements an associative array. Because the regions are based on the transport plane, lookup and mapping reuse the implementation from the sequential buffers. In essence, regions are yet another specialization of DBufs.

### 5.4 Parallelism

Pipelined logic is parallelizable in principle; this section identifies specific parallelism opportunities in Streamline. Parallel processing does not require any infrastructure beyond what is already present, since the datapath is fully reconfigurable: the two principal hardware abstractions, container tasks and channels, can be brought online and connected ad lib. Because compiled applications have fixed control flow and implicit buffering, they cannot be
adapted as easily to the number of hardware threads of execution. We first turn our attention to the current state of operating systems and contrast this to pipelines. Then, we present the scalability control methods in Streamline.

**Challenges** The relative impact of various I/O cost factors, such as page fault and task switches, changes as parallelism increases. These changes should be reflected in the system design. For instance, when tasks are run concurrently task switching completely disappears, but polling and locking become more pronounced. Modern operating systems such as Linux, Solaris and Windows have shown to be capable of scaling to thousands of processors, but those installations scale up other components in tandem: nodes consist of a CPU, memory and I/O interface. Multicore parallelism is much more widespread, but here parallelism is still much lower: the majority of installations contain at most four cores. In commodity operating systems, where applications are largely sequential, parallelism has been extracted by running kernel operations side-by-side with application code [BJL⁺06]. This strategy is practical on dual core machines, but will not scale to more processors, nor is it likely that the two parts will stress their processors equally. To scale, at least the most demanding part must be split further.

**Opportunities** Pipelines decompose naturally into loosely coupled segments. Over other structured approaches such as object oriented operating systems, pipelines add the constraint that interfaces between components are always asynchronous and uniform. The deliberately limited interface ensures that no blocking or locking is functionally required anywhere in the computation layer. The explicit nature of pipeline dataflow bares data access information that remains hidden in compiled applications. We split discussion of the parallelization in two themes: data and task parallelism.

**Data Parallelism** Streamline detects pure computational kernels and uses this information to extract data parallelism. For instance, packet switching is a stateless task that can be implemented by a pipeline consisting of only kernels. The control system uses template rewriting to replace a simple graph with a task-parallel one where multiple versions of the same filter are enclosed by a pair of split and join vertices. It introduces data parallelism by replicating a subgraph $S$ to form the set $S_1, S_2, ..., S_P$ with $P$ the upper concurrency bound defined by system parallelism. It then generates a new comprehensive graph by inserting a split/join pair of filters and connecting each such graph so that data streams from the split to each of $S_i$ through a stream with
classification constraint \( i \) and from each subgraph to the join. In essence, a single filter \( f \) is rewritten as

\[
\text{split}_{\text{rr}} \ | \ f \cdot f \cdot \ldots \cdot f \ | \ \text{join}
\]

where the first filter splits data in round robin fashion among all children. Figure 5.3 depicts the same configuration.

Streamline parallelizes not just kernels, but also filters that multiplex sessions (Section 3.3.2). A well known example is TCP receive processing, where state must be kept for each TCP stream. Sharing this state between cores is expensive due to write conflicts; instead, high performance is reached by spreading streams across cores, while keeping state local to the core [WRC06] (“connection scaling” as opposed to “packet scaling”). This is only one example of the general rule that stateful logic scales more poorly than kernels as a result of lock contention and cache conflicts [NYKT94]. Exception to the rule is the class of filters that have state, but do not share it and therefore do not require locking (although this class may experience cache capacity misses). Filters that multiplex sessions, such as the TCP reassembly unit in Streamline, can be automatically parallelized if they leave state segmentation to the runtime system (Section 5.3.3).

In connection scaling, as in block scaling, blocks are split among parallel paths by classifying blocks in disjoint classes. The difference with block scaling lies in the split filter, which must ensure that blocks belonging to sessions are consistently forwarded along the same path. An application filter must perform the task-specific part: classification. A task-independent split filter then performs a lookup from class to parallel filter instance. A cheap method uses a hash function for indexing and lookup; this is common in embedded hardware, such as on NICs with multiple receive rings and on-board filtering, where elaborate algorithms are difficult to implement. It does not necessarily balance load, however. Dynamic lookup takes existing load per space into account when it assigns a session to a space. Insertion and lookup are more
expensive, but load is more balanced. Streamline has a filter that implements the first. Developers can replace this by writing their own filters.

**Task Parallelism** The composite nature of pipelines renders task parallelism identification trivial; the challenge in effective task parallelization is to choose the optimal number of container tasks and to map filters onto these tasks so that cache hitrate is maximized while synchronization overhead minimized. The first choice – number of tasks – depends on policy. For instance, an application may want to conserve power by disabling cores, or run maximally spread out to minimize latency. The second choice depends on system layout. Both are control questions; an initial mapping is calculated by the control system (Chapter 6).

### 5.5 Summary

This chapter presented the computation layer. Operational logic is encapsulated in filters; the only centralized logic concerns filter scheduling. Streamline implements a multiprocess event network, where each space has one locally managed jobqueue. Because filters can only be scheduled when data arrives on their input, schedules are constantly updated. Streamline optimizes for DCache hitrate by executing consecutive filters in the same space in rapid succession for each block.

Streamline improves throughput over a straightforward multiprocess event network by reducing the amount of signaling (5.2), by reducing the amount of locking and copying while editing (5.3) and by extracting parallelizable code-paths (5.4). Streamline replaces events with cheaper function calling within a space and batches events between spaces. It replaces copying with in-place editing when safe, with metadata editing when the use-case permits and with appending when no original data needs to be retained. It tightly manages filter state to keep that amenable to migration alongside filters, e.g., for parallel execution. Finally, Streamline identifies parallelism in the form of computation kernels, stateful session handling and task-parallel networks. It migrates
filters at runtime to stay within the limits of a given policy and co-schedules on parallel hardware in a cache-aware manner. Together, the ideas presented in this chapter contribute to the central goal of a high-throughput, flexible I/O system by substantially reducing the base cost that composite applications (which are a prerequisite for reconfiguration) incur over integrated processes and by exploiting the compositionality to automatically scale to available resources.