To understand how throughput of network and media applications can be increased, we examine their structure (2.1) and identify common bottlenecks (2.1.1), then propose an optimization approach, *application tailoring*, that can consistently avoid bottlenecks across ranges of applications and computer systems and apply that to Unix pipelines in Streamline (2.2). I/O optimization has considerable research to which we offset this work (2.3).
2.1 Application Domain

Networking, multimedia and dataflow applications are at their core all operations on sequential streams of data (input/output, or I/O). Network applications encompass servers, clients and core elements. The first generally handle large numbers of small streams, the second a few large streams and the third ordinarily process stateless traffic. Illustrative examples are DNS servers, email clients and IPv4 routers. All encompass device, generic I/O and task-specific software processing. Multimedia applications produce or consume streaming audio and video content. Music players, Internet phones and digital television sets come to mind. Again, all consist of a combination of software processes and block, network, audio and graphics device logic. Dataflow processing is a generic term encompassing all applications whose tasks consist of operations on externally generated streams of data. Intrusion detection in network streams, fraud detection in financial streams and text mining in conversations are examples. Firewalls that perform content analysis can also be included in this category, as can high performance computing (HPC) applications that operate on live streams, such as live astronomical observations [ACK+02, RBvM+06].

Guiding Example: Intrusion Prevention To make the challenges and opportunities more concrete, we take one application as guiding example: network intrusion prevention. The popularity of the Internet brings an equal popularity of network-based computer intrusions. Because on this global space, local laws are largely ineffective and police protection concomitantly absent, task of securing data to reverts to individual users and systems. With an abundance of types of attacks, effective intrusion prevention at the system perimeter involves a range of tests: payload inspection and behavioral anomaly detection are just two examples. A powerful security strategy is “defense in depth”: combining orthogonal detection methods to increase detection rate. This approach is effective, but also resource intensive, as it executes multiple processes with (soft) realtime constraints and has to transfer potentially large streams between them.

SafeCard [dBSvR+06] is an intrusion prevention system (IPS) that practices defense in depth while minimizing system load. It is a network intrusion prevention system for edge hosts that combines full packet payload scanning, application-layer protocol filtering (which requires traversing the entire protocol stack) and flow-based behavioral detection. Figure 2.1 presents a high-level depiction of the sanitization phases each network packet most traverse. At this point the specific meaning of each phase is not important, we will return to that when we discuss the implementation. We selected state-of-the-
art methods that need considerable system resources and demonstrated for the first time the feasibility of an application layer IPS without burdening the CPU, by offloading all tasks to programmable peripheral processors on the network interface card. SafeCard is a critical, high cost streaming I/O application that exploits peripheral resources when available and integrates new protection mechanisms as they arrive. As such, it demonstrates the application type targeted by this thesis.

We will return to the SafeCard example occasionally, whenever it can clarify utility of individual components. We present the application itself in detail in the evaluation section, where we also quantify its efficiency in practice.

**Test Application Set** Exposing all bottlenecks in this broad set of tasks requires a basket of applications that vary in how they stress physical resources. For this purpose, we compile a compact application set for evaluation, each element of which exemplifies a common I/O processing profile. We selected applications from all three identified domains and all three locations in the network.

**DNS server.** As first application class we look at highly parallel network servers. The most extreme of these handle hundreds of thousands of requests per second per host. At these rates, communication is necessarily stateless and communication limited to only few bytes per request. Such servers pose an interesting benchmark case because their per unit application overhead low, exposing per-packet system I/O cost.

A prime example of this class are DNS servers. These form a critical part of the Internet architecture as their domainname to address resolution service is consulted routinely as part of application communication. For each request, the fast [JV06] industry standard *Bind9* server only traverses an in-memory tree [NSH06] and payload is brief compared to common maximum transfer unit sizes: both requests and responses are around 100 bytes compared to 1500. We benchmark unmodified *Bind9*. 
Video client. Multimedia clients expose a different system bottleneck. Here, request parallelism is limited, as these user-centric tasks generally process one recording at a time (of course, recordings can consist of a number of tracks). Like DNS, media streaming can scale to high data rates, but it saturates pipes with only few streams filled with large blocks. If computationally demanding processing is offloaded to the GPU, this profile demonstrates communication-intensive processing with large per-unit cost. We evaluate communication cost by measuring the CPU load (that includes I/O waiting time) when streaming a high-rate stream to the popular MPlayer media client while bypassing expensive computational processing on the CPU.

Traffic analyzer. The last class of avoidable overhead that we want to measure is the kind introduced by parallel processing. For this purpose we select a data flow application that attaches possibly multiple times to an existing stream. Network traffic analyzers, such as intrusion detection systems, are uniquely suited. They operate as a core network service, but often directly at the edge nodes that they mean to protect (e.g., as part of anti-virus packages). Here they execute in their own protected processes, but access the data from other tasks. They expose unnecessary data duplication by legacy operating systems. Modern software firewalls perform multiple types of defense: on parallel architectures these actions should be parallelized to minimize latency and spread load.

To observe performance degradation under increasing parallelism we attach the popular tcpdump traffic analyzer to a moderate datastream. The tcpdump application shows per packet information; it performs no stateful processing. It introduces very little computation and therefore adds stress mainly on the memory system. By default the application touches only headers, but it can be configured to touch up to all data in the stream. As a result, it can show scalability under parallelism for a range of application profiles.

2.1.1 Opportunities

Throughput of streaming I/O applications depends in large part on efficiency of data transfer between devices, kernel and applications. Such communication always crosscuts the traditional OS layering shown in Figure 2.2, where it can introduce significant cost in the form of task switches, kernel mode switches, cache and TLB misses, interrupts and page faults (to name only the most important). The actual bottlenecks depend on computer system details,
which vary from one machine to another: systems are increasingly diverse, heterogeneous, parallel and memory constrained, we argue in Appendix A. Before looking into specific cost saving opportunities, we observe where layered I/O leaves room for improvement. We present six common situations that witness avoidable transport cost. All are present in Linux at least up to version 2.6.30, but they are not necessarily limited to that OS. In each case, we identify specific cost saving opportunities. Afterward, we extract the root causes and identify opportunities for improvement.

**Avoidable Bottlenecks**

I/O bottlenecks take one of two forms. Communication overhead accrues where data is forwarded: at the crossings between (hardware or software) compartments. Computation inefficiency occurs when logic fails to exploit specific hardware features (such as ISA extensions or coprocessors). On strictly layered systems such as Windows, Linux or BSD (including MacOS X), applications encounter one or more of the communication and computation bottlenecks shown in Figure 2.3, each of which deserves a brief clarification.

1. **ABI** For security reasons, applications must communicate with devices by calling into gateway code that separates untrusted application from trusted system logic: the Application Binary Interface, or ABI. These *system calls* transition execution into and out of the protected OS kernel environment (kernel mode switch) and to maintain strict memory isolation they copy data between the two environments. Both actions incur cost: kernel mode switches add tens to hundreds of CPU cycles to save state and incur secondary cost by requiring cache flushes.
Challenges and Approach

Copies cause roundtrips to main memory and unnecessary cache capacity misses from having to store data duplicates. The combined overhead can form a sizable portion of total processing (Section 7.2).

2. IPC ABI overhead affects not just device access. The trusted kernel also isolates applications by making Inter Process Communication (IPC) primitives available in the ABI. Traffic between applications is copied both into and out of the kernel, resulting in three versions of the same data. Because buffering in the kernel is minimal, switching into and out of kernel is frequent. What’s more, besides kernel mode switches between CPU protection rings, IPC cause task switches between user processes. These are more expensive: in the order of 1000 cycles on modern x86 processors.

3. Group I/O Multiprocess access to the same data is seen in group communication, (which subsumes 2-party IPC). This is common in some HPC applications, but also simply when a virus scanner is enabled. Group I/O is a generalization of IPC, where the number of data copies and task switches grows linearly with group membership. Gains of shared memory and synchronization reduction therefore increase with group size. Especially on manycore, efficient group communication is important.

4. Kernel Within the operating system kernel, data is copied unnecessarily when subsystems cannot communicate effectively. A classic example and practical bottleneck [PDZ00] is data hand-off between the file and network subsystem. When a process transmits TCP data, the OS has to construct packets, copy over payloads and add the packets to the (re)transmission queue. If these packets hold file contents, their payload already resides in the file cache. Then, creating private copies for TCP constitutes unnecessary data duplication: pinning of the file cache page holding this data suffices. Simpler copying is generally used, however, because the file cache and network subsystems are functionally isolated. Similar cost accrues on the network reception of file data. TCP segments are reassembled, transferred to userspace, transferred back to the kernel and added to the filecache, causing much copying and switching.

5. Direct I/O Data is forced to traverse the trusted kernel even when no trusted operation has to be performed. Virtual machines make a telling example. These are known to be poor at I/O intensive tasks. This derives at least in part from the fact that I/O devices always trap into the host OS kernel, which does not touch the data at all, but must now issue a
switch to the guest, thereby degrading guest VM I/O performance by factors [MCZ06, LHAP06]. The same double task switch occurs in non-virtualized environments when devices and applications want an exclusive communication channel (and thus no resource multiplexing by the kernel is asked for) but must traverse the kernel. High-speed devices (e.g., DAG cards [CDG+00]) sometimes bundle software libraries that can avoid the kernel stack, but those generally suffer from one or more of these issues: they require superuser privileges, they require exclusive device access, or they introduce vendor-specific application programming interfaces (APIs).

6. Strict Software Layers Applications frequently hit the above bottlenecks unnecessarily because I/O logic is strictly layered between devices, kernel and user tasks, irrespective of application profiles or system abilities. To give two practical examples of how strict layering hurts application throughput: a network fileserver saves two copies and one task switch when it moves fast-path logic from a user process to its network processing kernel task; a DNS daemon reduces latency and cache pollution by bypassing the kernel completely and performing its (minimal) packet processing inline in its user process. In the figure, we show an expensive IPSec operation that would benefit from using the cryptographic co-processors on some network cards (network processors), but that can’t, because the implementation is kernel-specific. The first optimization has been carried out frequently, but always in a non-portable manner requiring superuser privileges. The second can be built with user level networking [vEBBV95], but not without considerable changes to application interfaces. The challenge is to design a system that executes both cases efficiently – with preferably relying little or no interface changes.

Test Set Bottlenecks We argued earlier that the application test set can expose a broad set of I/O bottlenecks. The DNS server performs fairly simple operations on streams of small packets. It will suffer mostly from the context switches involved in crossing the ABI (pt. 1), a consequence of strict isolation (pt. 6). Media streams consist mainly of MTU sized packets, making the cost of data movement (copying, virtual memory operations) more pronounced. Ideally, data is shared between the networking and video subsystem (pt. 4). In practice, it is copied up to a user application and down to the video subsystem. The last application, traffic inspection, needs access to all packets, also those flowing to and from other processes. This is a form of group com-
Root Causes  Bottlenecks curb application throughput by increasing total communication cost, which can be expressed as a combination of memory access and synchronization. For data intensive applications, memory access is the weakest link. These tasks must fetch data from fast caches whenever possible. All six bottleneck examples force trips to main memory, however, by issuing copy instructions. Copying adds immediate waiting time by introducing a double transport across the memory bus: one for reading and one for writing. On top of this it introduces delayed cost by evicting productive cachelines in favor of multiple copies of the same data. Context switching similarly stalls CPUs and pollutes caches. We discern two forms. Task switches between processes require Level 1 (“L1”) cache flushes (including TLB shootdowns) and on many architectures L2 invalidations. Kernel-mode switches occur on entry into and exit from the trusted kernel. These stall the CPU for a shorter length of time and require L1 and TLB flushes only on exit from the kernel.

Equally damaging as inefficient data exchange is efficient, but pointless exchange: even the most economical data path constitutes waste if it leads nowhere. Fixed logic introduces such unnecessary copying and context switching by forcing execution needlessly far from the data source. Splitting execution across hardware tasks decreases locality of reference and increases synchronization cost. On uniprocessors, splitting causes task switching.
multiprocessors it causes cache contention (unless working sets of neighbor-
ing cores overlap). The challenge is to match operations to tasks and tasks to
cores in such a way that contention is minimized and overlap is maximized.

**Opportunities** The presented bottlenecks arise when software stresses
hardware unevenly. The issues are systemic, i.e., they permeate all I/O logic,
from applications down to device communication. We introduce the term *I/O
architecture* to define the communication fabric that spans applications, OS
kernel tasks and device logic and therefore crosscuts classical OS layering.

Traditional layered I/O architectures, as exemplified by Linux, offer strict
isolation, but at a performance cost: copying and context switching cause
cache replacement and TLB flushing, among others. These concerns have
been known for decades [Cla85, CT90, DAPP93, BS96], but become more press-
ing as the performance gap between CPU and memory grows. Moreover, the
trend towards diverse manycore systems adds questions about how to safely
and efficiently integrate special purpose processors, how to extract maximal
recall from diverse cache and interconnection networks and how to parallel-
ize code so that throughput is maximized end-to-end. In other words, our
pressing concern is that

modern computer systems are single-purpose, memory constrained,
parallel and heterogeneous, but typical system software is struc-
tured for the time-sharing, computationally limited, uniproc-
sor systems of the past.

In the next section we distill an alternative operating system design that
adapts more freely to changes in hardware. We return to the six bottlenecks in
Section 7, where we evaluate the application test set on both standard Linux
and Streamline, our implementation of the alternative design. The evalua-
tion shows that the second achieves significantly higher throughput (1.3 to
10 fold).

### 2.2 Approach

This section presents an alternative design for system I/O that conserves
memory bandwidth and spans heterogeneous resources. We do not discuss
an architecture, _sec_, but reason towards a feasible candidate from our current
position, establish the soundness of that approach, distill a software archi-
tecture and, finally, present a conforming software package. Comparing the
package to best of breed alternative for the selected application set will ex-
perimentally validate the chosen approach.


2.2.1 Adaptation

The challenges presented in Section 2.1.1 are not fundamental, but stem from inefficient communication patterns dictated by inflexible software. All identified bottlenecks are avoided when superfluous communication is removed and the remainder is restructured to reduce memory system stress, which equates to minimizing the number of times data is touched and serving it from cache to avoid contention. One way to accomplish this feat on diverse combinations of processors and memory regions is to adapt application structure to hardware both in terms of computation and communication.

**Computation** Application structure dictates when and how often CPUs will fetch data. Heuristics such as inter layer processing (ILP [CT90]) increase throughput by removing redundant trips across the Von Neumann bottleneck. Strict segmentation of logic across OS layers reduces opportunity for ILP, a point illustrated by the direct I/O and fixed logic bottlenecks. Contrary to computational tasks, I/O applications habitually crosscut layering, rendering the argument in favor of strict separation tenuous. Operating system layering is a logical approach to achieving portability, process isolation and resource multiplexing, but network processing is mainly relegated to the kernel for performance reasons, not on these functional grounds. By the same rationale (that performance warrants code migration), this I/O application logic can be moved out of its current kernel thread to a more effective location (e.g., closer to the data) without jeopardizing system integrity.

Two reasons call for a reexamination of strict logic layering in the context of I/O applications. One, a flexible mapping of logic to tasks enables functionality migration across layers to remove redundant memory accesses or context switches. Take for example the aforementioned DNS server. Here, protocol processing is so minimal that scheduling a kernel task to handle it will likely introduce significant non-functional overhead (“useless kernel syndrome”). Bottlenecks such as this arise in rigid systems because tasks are forced to hand off data at inconvenient points in their execution. Fusing the few IP and UDP protocol processing steps into the application process trivially circumvents this cost.

Two, to scale on parallel and heterogeneous architectures, application code has to be broken up further. Breaking up protocol stacks to execute portions concurrently is the inverse of ILP extraction. Again, losing strict layering gains throughput, only on different hardware. Both solutions have utility, which is why neither should be precluded by hardwiring code. Current efforts at parallelizing protocol processing have largely been confined to extracting data parallelism. This approach alone will give diminishing returns as criti-
cal sections surrounding shared structures (such as the TCP connection table) introduce sequential portions [Amd67]. Splitting these coarse grain data parallel threads further into separate lookup and processing steps increases parallelism without increasing lock contention to further improve throughput. Moreover, each individual task will be simpler, which makes them more amenable to implementation on special purpose cores. In short, to bypass unnecessary communication, scale with parallel hardware and integrate heterogeneous resources, applications are best composed from small relocatable tasks (as long as per-task call overhead remains low). In the context of I/O, we term such composite applications I/O paths.

**Communication** Communication cost is reduced by minimizing data movement and by adapting transfer methods to match hardware characteristics. Applications want to avoid context switching, cache misses, page faults, interrupts and other such hardware costs. Which of these constitutes the main communication bottleneck depends on application pattern and local computer architecture. On uniprocessors, frequent context switching causes noticeable overhead (up to tens of percentages as a ratio of total cost, expressed in cycles). When tasks are spread across processors, frequent switching can be avoided, but now inter processor synchronisation and cache contention take center stage. A worst case scenario is that the same data has to be pulled from main memory for each processor involved in an I/O path. Key to achieving high throughput on a range of platforms is extensive use of shared memory — and thus caches — and relaxed but controlled synchronization between the various stages in an I/O path. Limited asynchronicity reduces both context switching (from synchronization) and data fetching (from cache overruns). The right balance between the two is computer system dependent.

Like computation, communication must adapt to match software requirements (e.g., latency sensitivity) and hardware features (e.g., L2 data cache, or DCache, size).

### 2.2.2 Method

Adaptation to handle diversity can be implemented in countless variations, but at their core all derive from only a handful of approaches. Three binary design decisions logically partition the solution space into six classes (not counting hybrid solutions): optimization can be compile or runtime, manual or automatic, and the responsibility of applications or the OS.
Runtime Optimization

In the present situation, where applications must be portable across platforms that differ in number and types of processors and caches, integrated compiled executables will perform suboptimal on most architectures, because important cache, memory and processor information is only available at runtime. At compile time far-reaching structural decisions can be made, but the output of this stage should remain reconfigurable. Even when targeting a single Instruction Set Architecture (ISA), performance differs dramatically depending on whether optional extensions (such as SSE) can be exploited, whether working sets fit cache sizes and whether processes can be effectively scheduled onto available cores, to give just a handful of examples. Optimization to such information cannot but be delayed until runtime. This is not to say that all optimization has to be delayed, of course: both stages have their application. In this work we take a system's view and leave the compiler out of focus. In this discussion, we do not interpret the term at runtime to mean strictly 'during execution', but use it in the broader sense of after compilation and installation. We explicitly include last minute optimization prior to execution.

A radical approach to runtime optimization is to switch exclusively to just in time compilation of bytecode interpreted languages [Sta]. The approach is promising, but requires pervasive changes to software. The Inferno distributed operating system [WP97] is the only known mature implementation of this idea, but has not published performance numbers for high rate applications and does not target complex (multicore) computer architectures. We investigate a more moderate approach to software modification, where important computation and communication optimizations are delayed to runtime, but applications are compiled and distributed in their current form. Applications incorporate self-optimizing control code in the software that manages (parallel) scheduling, memory management and communication. We show that optimization of performance critical elements can take place behind common APIs, shielding applications from the new complexity. Just in time compilation is more generic and will more easily incorporate fully heterogeneous architectures, but the pragmatic approach fits systems built from one dominant ISA supported by special purpose processors. So far, the products on shelves and in development all fall into this category.

Automation

Computer system complexity and diversity renders optimization tedious, if not outright daunting. Traditional system configuration development is es-
essentially human-centric: documentation is offline and verbose; software interfaces must be manually discovered. Such irregular system structure stimulates manual administration, including optimization. As system complexity and diversity grow, this path becomes increasingly costly, introducing excessive maintenance cost [BDKZ93, SPL03] and suboptimal resource utilization.

Even though hand-crafted programs invariably beat automatically optimized competitors, automation is the preferred route to battling cost deriving from unavoidable and increasing administration complexity. In complex optimization tasks, such as VLSI logic synthesis or aerodynamic structural design, automation has long been commonplace. The use in online decision making support is far less common. Manual optimization is a logical choice for applications that have large development budgets and no need to be portable, such as high performance scientific and business-critical applications that run on dedicated ‘big-iron’ machines, but the task is too difficult and time consuming to ask of all developers for all applications [Pat07]. For portable applications, automatic optimization offers a more cost-effective and robust alternative. While it cannot extract the same level of performance out of a specific machine, it will trivially outperform poorly adapted manual code across a range of systems. A related objection revolves around trust: self-optimizing code is sometimes considered less deterministic than hand-written applications and therefore intrinsically less trustworthy. This position is incorrect. The higher constraints imposed on software structure for self-optimization make the system more understandable: strict logic enables formal code verification at at least one level of operation (the component architecture), an assurance that exceeds trustworthiness of manual coding, which has been shown to have an error rate of around 3 faults per thousand lines of code [Hat97, DTB04].

To offload tedious decision making to support logic, software has to be amenable to introspection from this logic. The principal solution is to restrict system operation to conform to a simple model. A large body of work has accumulated on the architectural requirements of self-managed systems [BCDW04, KM07, OGT+99], many of which advocate a component-based approach [KM07]. System information is not trivial to convey and a semantic gap between model and reality will produce solutions with little use in practice. The challenge thus becomes to select a model with enough practical utility. In system administration today, automation is mostly limited to scripting of frequently used combinations. In the case of (Unix) pipelines, utility is not questioned. Its ubiquity is a good indication of the demand for configuration automation. It is no coincidence that the most common form is in Unix shell scripts. Their simple semantics makes automation straightforward and robust: automation of less structured systems will be much harder. If needed, techniques such as
information gap theory (Appendix B.2.1) can establish trust in the optimization outcome even when considerable uncertainty exists in individual input parameters.

**Cost Effectiveness**

On the constrained hardware of the past optimization overhead constituted a significant portion of total running time, but growth in raw computational power means that increasing numbers of cycles can be spent on supporting number crunching without introducing significant overhead relative to total computation [KKM+07]. A prime example of the decreasing cost of optimization is the cost associated with supporting the complex x86 architecture. The overhead, measured in processor die ratio, has stayed constant in absolute terms, but had shrunk from 40% [Sto06] on the Pentium Pro to “a few” [Web03] percent of a microprocessor die as early as 2003.

Decreasing cost is no reason in itself to incorporate online decision support. Introducing it is hardly ever justified, because it inevitably introduces overhead of its own. The true benchmark is not cost, but cost effectiveness. Added complexity can be justified in one of two ways. One, the decision support logic must noticeably aid the end-user. This kind of support is most famously exemplified by the office assistant (“Clippy”) introduced in Microsoft Office 97. This early attempt was heavily criticized for being obtrusive and without merit. Smarter support logic that does add to the user experience has come within reach, however, as the cost of executing common desktop applications such as email and word processing has dropped well below the cycle budget of PCs. Online, search engines (e.g., google.com) already support the user with detection of commonly misspelled words and clustering of topics.

A second strong justification to introduce automation is when doing so is immediately cost effective. In other words, the cost of performing online automation must be considerably lower than the difference in cost of the unoptimized and the optimized execution, where cost is commonly measured in cycles. We emphasize that the reduction must be considerable to warrant the added code complexity. For the relatively long lived I/O applications that we target here, runtime benefits today outweigh control overhead. Head-to-head comparisons in the evaluation section will demonstrate that throughput of application tailored systems surpasses legacy applications even on the hardware for which these were specifically designed; the cost of optimization is negligible for these tasks. Cost effectiveness is measured as the ratio of optimized to unoptimized execution. Goal is to reduce this objective function. It is most likely to be minimal for long running tasks. To maximize utility, the concept of task here must not be immediately equated to process, nor...
optimization to scheduling, as cycle budget at that level is low. Instead, online decision support must (at least initially) target one-time optimization of long running applications that may span multiple threads and processes. In this model, optimization is not a form of complex scheduling, but a complementary higher layer process more related to job control in distributed systems. It is even viable that one of the outputs of the optimization step is a situation-specific scheduling algorithm. With the advent of manycore hypervisors [SATG+07] application-controlled scheduling is gaining in popularity.

System Service

While optimization can be relegated to individual applications, it is better implemented as an operating system (but not necessarily kernel) service. The job encompasses resource discovery, selection and allocation: all tasks of considerable complexity in their own right. Discovery and control need to be closely interwoven with the operating system and require superuser privileges to access (protected) hardware. Splitting optimization in application and system parts is undesirable because this introduces a rich interface to communicate hardware features, while rich interfaces add maintainability and portability concerns. More importantly, it duplicates logic among applications. The cleanest solution to building a self-optimizing system is to centralize all optimization logic. This is not to say that applications should not play a role in optimization. Their requirements must guide optimization. Specifically, they must be able to specify the relative importance of metrics, such as throughput, latency or fault-tolerance. Such steering of the optimization process requires much less application logic, introduces simpler interfaces and can even be foregone in most cases.

We have argued towards a system’s approach to optimization of I/O logic. We are not the first to propose optimization at this level [KKM+07], but it is definitely not common practice in operating system design. In distributed computing, especially in the context of grids, automated resource discovery, selection and allocation are well entrenched [KB02, YB05]. As individual machines start to resemble distributed systems, one can argue that it is only natural that mechanisms from distributed systems find their way into host operating systems. To a certain extent, then, we only propose to reuse and combine proven ideas. Where this work differs from distributed systems, is in the resources that it manages. On a single machine, communication is more reliable. Fault-tolerance is less of a concern, as errors (e.g., power loss) are likely to affect the entire system at once. On the other hand, the view of a communication system as a set of fixed-bandwidth channels interconnecting isolated processes is untenable. We must adapt established control practice
to kernel and embedded environments that lack process and memory isolation and other userland comforts.

We introduce the term *application tailoring* to mean the automatic process of generating quantitatively optimized software applications from a set of composite application and system specifications and a set of relocatable computation and communication operations. The method is a practical midway point between portable but inefficient compiled programs and fast but error prone just-in-time recompilation of entire applications to each target platform. The approach is particularly well suited to high throughput I/O because I/O application code is already composite: it crosscuts classical OS layering and naturally decomposes into computation and communication steps. Also, I/O applications are generally long-lived, which is necessary to have increased operational efficiency offset the cost of the application tailoring process. Taken together, we call our approach for solving the issues set out in section one *application-tailored I/O*.

We take a purposefully conservative approach to software optimization by basing automation on linear programming, which has been applied to multiprocessor scheduling before [Sto77, NT93]. Even though quantitative optimization in systems has been frequently advocated in recent years [Sir06, KKM+07, GDFP09], practical examples remain few and far between. Successful implementation in an operating system should bolster confidence in the approach.

### 2.2.3 Model

Optimization begins with selection of application and system models. In parallel and distributed computing, modeling applications as sets of computation and communication operations is commonplace. The theoretical underpinnings of much work lies in the process algebra of communicating sequential processes (CSPs [Hoa78]): serially executing computations whose interaction is restricted to message passing. The model can formally encode algorithms and is extensively used for proving properties such as (lack of) deadlock. We are less concerned with formal proofs, but like others (e.g., Newsqueak [Pik89], Limbo [DPW97] and Ease [Zen93]) are attracted by the ability to express multiprocessing concisely and naturally. CSP is a coordination language [GC92, Arb96]: a grammar for compositing applications from independent operations – operations that are themselves expressed more easily in computation languages, such as C, Java or Lisp. In the context of another coordination language, Linda [Gel85], these languages are described as “an extension that can be added to nearly any [computation] language to enable process creation, communication, and synchronization [KM89]”. Clear sep-
The coordination of communication and computation makes it easier to distribute an application across heterogeneous hardware. Limbo is specifically designed with distributed multiprocessors and wide area computer systems in mind. With the arrival of large scale on-chip parallelism, many of the same issues crop up [Moo08] and solutions are proposed [Pat07]. The relationship between coordination and concurrency has been discussed in relation to Linda and other parallel languages [Arb96, Bal91].

**Pipeline Application Model** The Unix pipeline is the most established coordination language, used in many applications and known to many developers. In this model, streams are bitpipes that connect self-contained computation atoms, or filters, written in separate computation languages. The pipeline is similar to CSP in spirit, but less expressive. It can only compose sequences and relaxes synchronization constraints. Because we target single machine installations and the Unix pipeline is predominant here, we base our interface on it as well. In networking and multimedia processing, filters encapsulate indivisible I/O computation tasks such as protocol processing (fragmenting, multiplexing, checksumming, network address translation), filtering (traffic shaping, virus detection), replication and redundant coding (RAID), accounting, logging, compression, encryption, tunneling, graphics kernels, logical operations and many others.

Streamline uses the Unix pipeline syntax, not its implementation. Pipelines are natural concepts for reasoning about I/O and parallelism, but current Unix defines an implementation that is expensive on current hardware. It maps computation and communication elements one to one onto CPU tasks and OS pipes, which impose frequent task switching, copying and, consequently, cache misses. This does not invalidate the model, but merely shows that on modern hardware other implementations may be more efficient. The high cost of traditional Unix pipes has caused development of additional conceptually similar but incompatible interfaces [dBB08c], e.g., the composable network stacks pf in OpenBSD or netfilter in Linux. Needless to say, such duplication of effort is a bad idea as it reduces interoperability without bringing any benefits. As part of our research, we investigate how the data access and synchronization cost of Unix pipelines can be reduced to render make the language suitable for high throughput tasks.

The pipeline model can be implemented on top of distributed resources. Streamline implement pipes over UDP (among others) to execute Unix pipelines across a network of machines. Transparent distributed systems indicate that such an approach is practical [GW96, PPD+90] and that existing Unix pipelines can form a basis [PPD+90, KTY03]. To create practical applications, the con-
control system must adapt compositions to gracefully withstand the wide variation in latency and throughput between IPC and remote streams. At this point, this necessary control is missing from Streamline; distributed operation is not the focus of this work.

System Model To reason about parallelism and diversity, we expand the basic pipeline model with grouping per CPU, device or other resource that can host a filter (e.g., an ASIC chip). Computation spaces, or spaces for short, are typed containers in which filters can be run. Space are typed, because they embody specific operating conditions, such as an x86 ISA with Solaris userspace API. A space instance is an active execution environment combining a type with a memory protection domain and processor time. It maps directly onto a CPU hardware task running a specific operating system at a specific user level, but also matches more restricted resources such as a GPU shader. Each filter implementation (i.e., the executable code) can be started in one or more space types and each started filter instance executes in exactly one space. Communication links between spaces are, like streams between filters, modeled as pipes. To differentiate the two, we call this second kind channels. Channels are modeled explicitly, because their characteristics differ considerably, which can be taken into account during optimization. Functionally, they abstract away the technical differences between physical media, such as shared memory, DMA bus transfer and UDP network tunneling. The reference system ships with four implementations (mmap, pci, posix-io, udp).

Application tailoring is the process of merging the application and system model so that each filter is tied to one space and each stream has one channel implementation. This runtime model completes the set of three main models. Application models are hardware independent graphs of computational tasks and communication streams. System models are hardware specific graphs of computation spaces and communication channels. The runtime model combines the two in a (hopefully) optimal configuration. Figure 2.4 shows the three models.

2.2.4 Architecture

The chosen approach integrates established methods, such as software pipelines, and refines these where needed to match our purpose. Before discussing method technicalities, we structure our approach. Modularity increases clarity; a simple system model is particularly attractive in application tailoring, where it has to be explained to an automatic optimization algorithm. In pipelines, computation and communication are completely distinct concerns. Applica-
Approach

Figure 2.4: A pipeline mapped onto a system model

Figure 2.5: Three-layer Architecture

tion control, in turn, is external to both. All three pillars of our approach can be refined to increase global gain. To organize both the software and this dissertation we separate the three concerns as much as possible. We introduce the role and interactions of the major subsystems and briefly outline the internal design of each, mentioning which established methods are used and how they need to be refined. Figure 2.5 displays the whole architecture.

Three Planes To separate the interests of computation, communication and control, the architecture is split into three distinct components that all span across the physical hardware. Following established practice in networking, we call these spanning components planes. We introduce the notion of a computation plane as a pipeline oblivious to memory concerns. At this level of abstraction, data produced by one filter magically appears as input to the next in line. Underneath, a distinct communication plane holds the actual plumbing required to ship data between tasks with minimal interference. These planes together make up the runtime I/O architecture. Above
both, a *control plane* observes, interacts with and modifies the runtime system structure.

**Computation** Pipelines are appropriate for streaming I/O in principle, but a straightforward implementation fails to take into account the complexity of modern computer architectures (with parallelism, multi-layer caching, task-switching and preemption, etc.). Streamline introduces an independent system model and the tailoring phase to optimize for this diversity. It also reduces pipeline synchronization overhead. User-level multithreading is an alternative to full multiprocessing that trades of isolation for performance. Multiple filters that make up a single I/O path can certainly be safely co-located. The change not only reduces task switching, but enables inter layer processing, chiefly from increased data cache sharing. If filters cannot be co-located for some reason, Streamline amortizes synchronization cost through event batching. Finally, Streamline automatically extracts some forms of parallelism to scale applications to the parallelism in the local hardware.

**Communication** Efficient communication is instrumental in optimizing I/O application performance, as the impact of memory access on overall performance grows with the Memory Wall. Even though processing follows a pipeline model, underneath an independent *communication plane* moves data through the computer architecture not necessarily linearly, but in any manner that minimizes transport cost. We present a buffer management system (BMS) that spans all software tasks and peripheral hardware and that is designed for throughput as follows: all live data is kept in coarse-grain ring buffers, buffers are shared long-term between protection domains and data movement is frequently replaced with updates to metadata structures (similar to IO-Lite [PDZ00]).

**Control** A principal goal is that the application interface does not expose the complexity of the computation and communication planes to users, but that instead a control system makes good enough (preferably optimal) choices on their behalf. In this view, the control system is the link between the abstract user world and the physical computer system. It accepts task-oriented I/O path requests and maps these onto the available hardware by automating resource discovery, selection and allocation. The first and third steps directly interact with the runtime system through a message passing network that connects all spaces. In the selection step the optimization algorithm picks an optimal set of components from the available candidates.
To support legacy applications (including those in the representative set) and to demonstrate utility of this architecture quantitatively, we have engineered three popular legacy APIs on top of the native interface: sockets, pipes and the packet capture library. Applications can choose from one of three interface types: compliant legacy, optimized legacy and native. Compliant legacy calls follow the Posix rules on call semantics completely (if not on isolation). Optimized legacy calls are near compliant, but change semantics or call parameters in minor ways where doing so significantly improves performance – and offer a simple update path for legacy applications; The native interface, finally, offers the greatest potential for performance optimization and portability.

**Synthesis**  Figure 2.6 again shows the software organization from Figure 2.3 using the alternative architecture. Here, relocatable tasks read from and write to large shared circular buffers that are mapped across memory protection domains. The figure shows that a configuration easily comprises tens of computation and communication elements, which is the basis for the quantitative component selection underlying application tailoring.

### 2.2.5 Implementation

The software architecture handles various concerns, such as memory management, parallelization and device integration. The complete implementation effort can be similarly deconstructed into a number of independently published research projects, summarized in Table 2.1. In practice, there exists
Challenges and Approach

<table>
<thead>
<tr>
<th>System</th>
<th>Main feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFPF [BdBC+04]</td>
<td>Kernel I/O offload</td>
</tr>
<tr>
<td>Beltway Buffers [dBB08a]</td>
<td>Communication</td>
</tr>
<tr>
<td>PipesFS [dBB08c]</td>
<td>UNIX integration</td>
</tr>
<tr>
<td>Model-T [dBB08b]</td>
<td>Parallelization</td>
</tr>
<tr>
<td>NIC-FIX [NCdBB04]</td>
<td>Peripheral I/O offload</td>
</tr>
<tr>
<td>SafeCard [dBsvR+06]</td>
<td>Embedded operation</td>
</tr>
<tr>
<td>Streamline [dBBB08]</td>
<td>Application-tailoring</td>
</tr>
<tr>
<td>NIE [dB04]</td>
<td>Control Automation</td>
</tr>
<tr>
<td>BetaGis [dBBB05]</td>
<td>Control Automation</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of systems developed as part of the investigation

considerable overlap between the top seven, which together form the most mature practical outcome of our research: the incrementally grown Streamline open source software package. Unless otherwise stated, all implementational observations relate to this system. The other two projects are more independent investigations into automation logic. We briefly recall the independent contributions of all nine projects and their influence on present Streamline.

Elements

**FFPF** is the immediate precursor to present Streamline. The two have no clear division: they are the same package, but we changed the name once the software outgrew its original application domain. To make this clear, we will occasionally refer to the system as a whole as Streamline:FFPF in the context of packet filtering. The first version of FFPF (the term is an acronym of ‘Fairly Fast Packet Filter’) is more restricted in application scope and technical capability. It only targets network traffic inspection: a read-only, ingress-only, mostly stateless task. Like Streamline, its performance goal is maximization of application throughput, which is simpler to accomplish for this limited domain. The single method to improve throughput in FFPF is to move logic as close as possible to the network interface. In its first incarnation [BP04] it supports offloading of a single operation to OS kernel or device together with zero-copy transport to userspace of a single stream. The second version [BdBC+04] extends functionality into an only slightly more restricted programming model than present I/O paths. This version accepts arbitrary directed acyclic graphs (DAGs) of operations and optimizes the map-
ping onto available hardware. The control algorithm heavily favors traffic inspection and always "pushes processing down" as far as possible towards the NIC.

**Beltway Buffers**  are the independently published form of Streamline's communication plane. They extend Streamline:FFPF with concurrent stream support and multi-user data access control. The central design feature is that *all* streams are accessible as Unix pipes, *anywhere*, to support the widest possible selection of applications. On top of these basic functional requirements, Beltway adds a selection of copy avoidance mechanism. Specifically, it avoids all copy and context switch overhead introduced by the Unix system call interface by replacing this with local function calling on shared memory. Beltway Buffers form the bulk of Chapter 4.

**PipesFS**  opens up traditionally opaque operating system kernel I/O to inspection and control from external applications. As its name implies, PipesFS is a virtual filesystem for Unix pipes. It models kernel I/O pipelines as a Unix filepaths. Each filter is a directory; its output stream is accessible as a Unix pipe in this directory. Applications can insert, move and remove filters through common tools and inspect streams from arbitrary applications. A kernel component automatically forwards all data produced by one filter to all filters downstream of it. The PipesFS interface is one of the interfaces to Streamline and one of the topics of Section 3.2. The kernel component mentioned here is Streamline's central computation system portrayed in Chapter 5.

**Model-T**  is a resource controller for pipelined I/O on massively multicore architectures. The central concept is the assemblyline as natural runtime equivalent of a pipeline. Whereas hardware pipelines are clocked, assemblylines run at a continuously varying global rate. Our goal is maximization of this end-to-end throughput; Model-T increases rate by iteratively identifying and removing local bottlenecks (e.g., through task migration). In the end, it aims to fully exhaust a minimal number of proximate CPUs, so as to maximize shared cache hitrate and reduce memory system stress. Streamline aims for the same, but optimizes only at program load, not reactively at runtime.

**NIC-FIX**  is a branch of Streamline:FFPF that adds device offload support. It ports Streamline:FFPF to the StrongARM control CPU of an Intel
Challenges and Approach

IXP1200 network processor and introduces bridge code to integrate the device Streamline:FFPF instance into the control domain of a version executing on the host processor. To support offloading to the IXP’s special purpose microengines, NIC-FIX introduces on demand compilation and code loading of a new bytecode language, the FFPF Packet Language (FPL) [CdBB05].

SafeCard extends a later version of Streamline in a similar way as NIC-FIX to newer hardware: it ports the x86 codebase to the Intel xScale ARM processor on the Intel IXP2x00 network processor architecture and introduces a pattern recognition language for high rate intrusion detection. SafeCard is a self-contained embedded intrusion prevention application; it lacks a bridge to a host Streamline instance. It goes beyond NIC-FIX by supporting high rate (1 Gbps) full TCP reconstruction to support read/write access to application streams.

Streamline binds and extends the previous six distinct elements to implement the I/O application tailoring architecture presented here. Beyond what is already mentioned it contributes quantitative general purpose automation; legacy and otherwise familiar APIs; enhanced security and fault-tolerance; and generic read/write support for both reception and transmission paths.

NIE, short for network inference engine, is a Prolog based expert system for automatic composition of application-tailored network stacks. From a set of application constraints and list of available protocol handlers, the NIE builds an application tailored protocol stack blueprint. It takes the same conceptual direction as Streamline, but is less mature. The NIE does not really interact with an external ecosystem, but reasons about an idealized control domain. It can be considered an early proof-of-concept of quantitative optimization for operating systems design.

BetaGis, an abbreviation of “beta quality grid information system”, is a similar Prolog-based investigation into runtime automation. The purpose of this tool is to automate system configuration of wide area distributed systems, or grids. Contrary to the NIE, BetaGis does perform resource discovery and allocation autonomously. Streamline derives its concept of template substitution for solution space expansion (Section 6.2.1) from this.

Software Package Streamline is implemented as a combination of a Linux kernel module, a userspace library and a set of device drivers that together
completely replace native I/O in Linux 2.6 (for x86 and amd64 platforms). Streamline is open source\(^1\) software and forms the basis of real performance-critical network applications, such as a 10Gb cryptographic token-based switch [CGXB07a]. Streamline has also been ported to userspace Solaris and to the Intel IXP 1200 and 2x00 lines of network processors. Because Linux kernel and device driver logic updates quickly, no single revision of the package integrates all features of all projects. Especially IXP support is limited to a few specific versions.

The concept of an I/O architecture is both wider and narrower than an OS. On the one hand it exceeds OS delimiters, by spanning across system boundaries into applications and hardware. On the other it is more limited, as it deals only with I/O. Indeed, the reference Streamline implementation on the one hand completely bypasses the Linux I/O stack – from peripheral devices up to libraries – yet on the other hand requires Linux, e.g., for process control and virtual memory management.

### 2.3 Related Work

The architecture is a blend of ideas that have extensive research behind them. This section traces the lineage of the three elements of computation, communication and control and points to influential work on the topics of parallelism and heterogeneity in operating systems. It concludes with a discussion of two application domains at the cutting edge of I/O processing.

**Computation** Structured OS design has a long heritage. Most relevant to our research are those systems that dealt with structuring of the I/O logic. Streams [Rit84] was the first pipelined operating system network stack. It interconnects functional modules through full-duplex messages queues (“streams”) in a vertical stack. Full-duplex processing forces each module to have both an upward and downward task: a one-to-one mapping not always present in protocols. Layer crosstalk and a need for session handling complicated clean compartmentalization. A pure stack design also oversimplifies the control flow in networking code. Later versions added limited support for multiplexing. Streams was created in the same period as the now prevalent socket interface to networking. While architecturally influential, the implementation failed to match the performance of sockets: composite I/O paths proved less efficient than handcrafted counterparts, not in the least because of the inter-module transfer overhead they introduced. Universal uptake of TCP/IP rendered the flexibility advantage largely irrelevant.

\(^1\)Licensed under the LGPLv2. Available from http://netstreamline.org/
In the decades since the introduction of Streams many of the practical problems in pipelined I/O have been tackled, although no system combines all solutions. The x-Kernel [HP91] cleanly integrates session handling and extends the stack to include a configurable network interface (NIC). Processing extends across protection domains and overhead is minimized by using procedure calling between modules (instead of task switching). The x-Kernel was later extended into Scout [MMO+94], which adds QoS support and extends the network stack upwards to include application layer processing (e.g., MPEG decoding). Scout is also available as a Linux kernel module [BVW+02]. Click [KMC+00] applies pipelining to packet routing. It operates at layer 3, where no per-session or per-flow state has to be kept and paths can therefore be static. SEDA [WCB01] is a pipelined network server architecture that uses self-scaling thread pools to minimize queuing in the pipeline. As a result, SEDA can automatically scale across diverse hardware for diverse applications. Dryad [IBY+07] also maps a directed graph of operations to distributed resources. It automates resource selection, like Streamline, but like SEDA applies this to achieve scalability and fault-tolerance. It targets distributed systems, which changes the problem definition. StreamIt [GTK+02] uses the streaming I/O model to perform compile-time optimization. The compiler can adapt data and instruction layout to maximize cache efficiency on a specific platform [STRA05a]. Space [VNGE+06] maps a series parallel graph of operations onto consumer electronics devices.

Besides pipelining, Streamline employs generic I/O computation optimization, such as replacing context switching with procedure calling [Cla85] and executing multiple tasks within the same process [CT90].

**Communication** Streamline minimizes I/O overhead. The two major causes of non-functional overhead are copying and context switching. Copy-avoidance mechanisms replace copying with virtual memory (VM) techniques such as page remapping and copy-on-write; common solutions work at block granularity. Block level remapping trivially ensures security, but at a cost: recurrent modifications to VM structures reduce efficiency. Brustoloni [BS96] categorize previous efforts and showed them to perform roughly identical. Druschel et al. describe copy avoidance ideas for network buffers [DAPP93] and subsequently translate these into Fbufs [DP93]: copy-free communication paths across protection domains that remove the per-block costs in certain situations. Paths are efficient only if mappings can be reused between blocks. Fbufs were later incorporated into IO-Lite [PDZ00], a BMS that introduces a mutable structure of pointers to immutable buffers to replace copying. Another system, Xen exchanges descriptors between domains through
Related Work

rings to reduce switching, but does not allow shared data rings [FHN+04]. The Streamline BMS shares features with both systems. It avoids per-block overhead as well as unnecessary descriptor duplication by combining shared ring buffers [GA91] with pointer indirection (as well as other kinds of specialization).

Control Self-optimization of operating system logic is not a new idea [BR76], but application is rare in practice. We first summarize models and then review actual systems. The problem of mapping parallel applications onto parallel hardware has been extensively studied in the past in the context of multiprocessors. In 1993, Norman and Thanisch summarized the state of “models of machines and computation for mapping in multicomputers” after 25 years of research in the field [NT93]. They classify problems, the most complex of which combines software diversity, task precedence and communication constraints. Streamline falls within this category. At least one model they discuss is based on network flow programs (linear programming) [Sto77]. It does not match our problem perfectly, because it has the constraint that only a single process can be active at any one time. A model not mentioned there (it was then only recently published), “Temporal communication graphs: Lamport’s process-time graphs augmented for the purpose of mapping and scheduling” (TCG) [Lo92], maps directed acyclic graphs of filters onto message-passing parallel architectures and models independent receive, process and store phases for each operation to identify memory contention. Streamline does the same. The two differ, among others, in that all communication has to be synchronous in TCG. Perego et al. introduce a “framework” based partially on TCG that takes into account network cost of memory traversal in massively parallel computers [PDP95] and expands to incorporate asynchronous I/O. This is a compile-time solution and focuses on “unprecedence-constrained” (analogous to independent, i.e., not pipelined) tasks, limiting its applicability to modeling streaming I/O on diverse hardware. No doubt these models are more refined and theoretically solid than that of Streamline. We have taken a pragmatic approach and built something that works well enough and within acceptable time constraints; this aspect is most definitely not without opportunity for improvement.

In systems design, application of quantitative optimization remains modest. Although it has been recently advocated on multiple occasions [Sir06, KKM+07, GDFP09], experimental results are few. Related work can be grouped into three categories: component specialization, communication reduction and code relocation. The first uses reflection to select a task implementation at run-time. Synthetix [PAB+95] specializes I/O system call implementations
Challenges and Approach

(e.g., \texttt{read}), similar to Streamline buffers. Unlike Streamline, it also automatically chooses the best matching implementation based on application invariants. Object-oriented (OO) operating systems such as Apertos [LYI95] and Choices [Cij+91] use late-binding of objects, but have no selection criterion to choose among implementations and operate on a single object (as opposed to a composite I/O path), optimization of which can lead to local optima. Amoeba [TvRvS+90] is a distributed object based operating system that selects optimal communication channels based on object location. Flex [CFH+93] does the same, but also moves logic between address spaces if this reduces communication overhead. Lipto [DPH92] and Kea [VeI98] also allow relocation of logic between address spaces to reduce communication overhead. EPOS [FSP99] and Knit [RFS+00] construct operating systems from components. All presented systems are reconfigurable, but none automate the optimization step. Ensemble [LKvR+99] combines off-line and on-line optimization to extract common codepaths and invariant data in composite network stacks.

\textbf{Parallelism} Parallel and distributed computing are long standing research domains; even within single computers parallelism has been common since the earliest timesharing operating systems. Urgency decreased as single purpose personal computers and servers became the norm, but the arrival of on-chip parallelism again fueled industry interest. Changed computer usage (e.g., single use machines) and system architecture pose new research questions. Cache-aware scheduling is undoubtedly an important component [FSSN04]. Before that can be applied effectively, however, more parallelism must be extracted from applications. Scaling beyond a few threads is known to be difficult with the prevalent programming models, which directly expose low-level primitives such as atomic instructions, hardware threads and shared memory. Large industry sponsored research projects are investigating radically different programming models to make deployment of parallel programming at large scale feasible, such as domain specific languages and just-in-time compilation [ABC+06b, Sta]. Streamware [GCTR08] is a hybrid compiler/runtime system that uses machine independent bytecode in this manner specifically for stream programming.

An immediately available solution is to present application developers with more abstract interfaces that parallelize well. The Intel Threading Building Blocks (TBB) [WP08] is a C++ template library that presents scalable templates for processing (e.g., a parallel for loop) and datastructures (e.g., queues). Streamline advocates a similar approach of moving to higher level abstractions and automatically optimizing concurrency details underneath. Replac-
Related Work

Handling diversity in manycore systems is the focus of the Barrelfish operating system [SPB+08], which identifies three types of diversity (memory non-uniformity, core diversity and system diversity). Like Streamline, the authors propose using quantitative optimization to combat complexity, but details of the knowledge representation and optimization algorithms are not yet public. Heterogeneous hardware has been studied in the past in the context of extensible operating systems and programmable peripheral hardware. Scout [MMO+94] has been used in conjunction with a programmable NIC to build an extensible layered router, Vera [KP02]. Like Click [KMC+00], Vera works at the network layer. It uses layering similar to spaces, but I/O paths are handcrafted. Extensibility does not reach the lowest layer. Spin [BSP+95] moves application logic to the kernel. The solution is similar to Streamline's filter offloading to peripheral hardware [NCdBB04], but restricts itself to single safe extensions and NICs.

Applications Two application domains that repeatedly push the envelope of I/O throughput are network packet filters and stream databases. Packet filtering is the only application inside the core network that touches packet contents beyond network layer headers. Stream databases process application streams, such as financial transactions or personnel records: another type of events that arrive at variable, high rates.

Packet Filters Streamline replaces rigid protocol-based stream selection with user-controlled selection and modification operators. In this sense it is related to packet filters (from which it originated). Streamline embraces
existing bytecode machines, such as the Berkeley Packet Filter (BPF) [MJ93] and extends the state-of-the-art with its own filter languages, FPL [BdB04] and Ruler [HvRB07]. FPL is a stateful packet filter that can be compiled at runtime to various hardware architectures, such as Intel IXP network processors. Windmill [MJ98] and BPF+ [BMG99] are similar, but lack Streamline’s support for hardware selection and code-shipping. Ruler is a pattern recognition and transformation engine optimized for network streams. Like FPL, it has a backend for the IXP but also for FPGAs. Like Pathfinder [BGP+94] and DPF [EK96], Streamline supports prefix-sharing through overlapping requests. While Pathfinder was developed as a module within the x-Kernel, Streamline supports this behavior natively throughout the system.

Stream Databases Stream processing is not limited to network traffic. It also underlies stream databases (SDs) [Gro03, CcC+02, AAB+05, CFPR00, SH98]: query systems for continuous application-level data streams. SDs process higher-level records, such as financial transactions and call records, and operate on a different timescale. SDs use disk-based storage, which is infeasible for network processing. As such, they have a different definition of high-speed. As memory is not the bottleneck data duplication is allowed here. Software can be taken completely out of the kernel, removing many of the technical details that plague network processing (as well as opportunities for optimization). Borealis [AAB+05] adapts the datapath at runtime based on changing user interest. Aurora [CcC+02] uses shared queues to reduce in-memory data copying. A cross-over system is Tribeca [SH98]: a stream database that targets network traffic. Tribeca must minimize copying. It introduces projections to create logical views of dispersed data, similar to Streamline’s zero-copy reassembly.

2.4 Summary

Run-time optimization is a promising approach to achieving high throughput across diverse computer architectures. As sole resource controller, the operating system is the only software system that can make scheduling and memory management decisions and therefore the likely location for optimization logic. Optimization is based on choice: a mapping from many possibilities to a single reality. Many applications are monolithic entities that defy external reorganization, but I/O applications have long been designed as pipelines that allow it. Current operating systems do not map pipelines ‘intelligently’ onto the available hardware to exploit all features. One that does can be expected to reach consistent high performance across existing and future sys-
tems. Although software designs are notoriously hard to compare quantitatively, this point can and will be demonstrated.

Together, the ideas presented in this chapter contribute to the central goal of a high-throughput, flexible I/O system by introducing an OS architecture that can automate runtime application optimization. We will conclude each of the following four implementation chapters with a similar statement that emphasizes how the presented component contributes to the overall goal.

**Thesis Outline** The remainder of this thesis follows the software architecture of Section 2.2.4. As is customary, Chapter 3 first presents the system interface. The following three form the core technical exposition, where each treats one software layer: Chapter 4 introduces fast transport, or communication, Chapter 5 adds synchronization and parallelization for efficient computation and Chapter 6 shows how self-optimization control adapts applications to changes in hard- and software. An evaluation of the implementation follows in Chapter 7. Chapter 8 summarizes and concludes. In short, this thesis follows a standard challenge-design-implementation-evaluation layout, where the implementation discussion is divided into four concerns: interface, communication, computation and control.