An important task of computer systems is the swift processing of sequentially ordered data. Such streaming I/O is the common pattern behind networking and multimedia, for instance: tasks that are increasingly commonplace and whose bandwidth demands continue to grow with Internet uptake and service fidelity. Streaming I/O applications cannot scale service naturally with Moore's Law, because they stress multiple hardware components (CPU cores, caches, main memory and I/O devices), some of which do not scale with Moore's Law themselves (such as memory access latency). This thesis proposes a number of new techniques that ensure high streaming application throughput on diverse systems. In I/O, stress originates not only in application code, but accrues systemwide, especially at crossings between tasks and processors, in the form of signaling (task switching, polling and interrupting) and memory operations (copying, page fault handling and cache/TLB misses). Adapting application behavior to match hardware features can significantly reduce this cost and increase throughput. Cache aware transfers are 3 times as fast as their default equivalents in Linux (Section 4.5); video player communication is 67% cheaper (Section 7.2). But diversity among computer system makes these opportunities moving targets. Reaching high throughput in the face of core and cache variation demands careful optimization — which for most users is too daunting or tedious. Automation offers consistent diligence and is cost-effective for these long lived streaming I/O applications.

This thesis proposes the Streamline system I/O software layer to allow ap-
plications to achieve high throughput while hiding most tedious work from the programmer. Streamline consistently reaches high throughput by reducing data movement and signaling between applications, kernel and devices. It is based on Unix [Rit84] and built into Linux. Streamline extends Unix pipelines to kernel and device contexts, because much I/O processing takes place here. By spanning across all contexts, it controls and can optimize all data movement and processing. Over Unix, it reduces crossing cost by replacing copying with shared memory and indirect access and by replacing context switches with userlevel threading where possible and by batching switches elsewhere. It increases processing efficiency by moving logic to the most suitable context. It identifies hardware, calculates capacity and plans execution to maximize end-to-end throughput. In short, Streamline adapts Unix I/O to diverse systems. A reference implementation is open source (BSD/LGPL) software released for Linux 2.6.19 and higher. Streamline features can be grouped in interface, communication, computation and control concerns. For each of these groups, this thesis contributes new ideas. We briefly summarize all here (but defer thorough explanation of technical terms to the respective chapters):

**Interface (Ch. 3)** A virtual filesystem opens up all kernel I/O to user monitoring and control using any common tools or scripts. Pipes into the kernel give processes live access to all OS streams. For application construction, Streamline revisits the Unix Shell pipeline syntax. It extends it with parallel and conditional paths to support branching, e.g., for protocol demultiplexing, filtering and connection handling, all without data touching. Split-join parallelism and backlinks build arbitrary directed graphs; at joins, duplicate data can be transparently filtered with Boolean logic. Loop constructs render the language Turing complete and enable multi-pass filtering. To apply all performance optimizations to existing applications, common socket and packet filter interfaces are built on top of these interfaces.

**Communication (Ch. 4)** Large shared buffers replace frequent copying and virtual memory operations, increasing pipe throughput 4x. Software indirection emulates pointer queues across protection domains to avoid data touching systemwide; splicing uses the optimization for a 3-10x throughput increase over common copying. Buffer specialization optimizes the implementation of the large shared buffers to match access patterns; for example, cache scaling increases throughput by 200%, fragmentation avoidance wins 20%.
**Computation (Ch. 5)** Userlevel threading replaces signaling with function calling (up to 100x faster) and event batching amortizes the remaining cost (3x as cheap). Write substitution replaces data edits with metadata operations, appends or bookkeeping to reduce data contention (locking) and update collision (copying). Parallelism extraction detects safe speedups: stateless kernels and session-handling filters are identified along with the task parallelism inherent in the pipeline.

**Control (Ch. 6)** Reification enables runtime selection of filter implementations. Buffer specialization and stacking do the same for streams. A graph transformation algorithm maps all such options onto a single linear network flow program and computes an end-to-end (theoretical) optimal mapping. State space grows quickly, but practical problems are calculated in tens of milliseconds. Execution logic turns plans into real applications; for embedded operation and safe hotspot optimization, it integrates compilation and code loading. Unix file permissions on filters and streams enable safe restricted access to kernel and devices by untrusted users, e.g., for task offload and zerocopy network reception. They make trading of isolation for performance possible on a case-by-case basis.

Collectively, these changes improve legacy application throughput on common hardware 1.4x for the Bind DNS daemon, 3x for the MPlayer video player and 3-10x for the Tcpdump traffic monitor. Heterogeneous computer systems reach Gigabit linerates for deep packet inspection, application layer intrusion prevention and cryptographic token-based switching as a result of employing all hardware (Chapter 7).

In short, Streamline presents an adaptive operating system design for high throughput I/O. This thesis follows a standard challenge-design-implentation-evaluation layout, where the implementation part is divided into the four mentioned concerns: interface, communication, computation and control. Before we clarify and experimentally verify all introduced mechanisms, the next section first examines the application domain and argues why pipeline optimization is uniquely suited *in principle* to increase throughput across diverse systems.