SILK: Scout Paths in the Linux Kernel

Andy Bavier [◇] Thiemo Voigt [*] Mike Wawrzoniak [◇] Larry Peterson [◇]
Per Gunningberg [‡]

February 22, 2002

Abstract

SILK stands for Scout In the Linux Kernel, and is a port of the Scout operating system to run as a Linux kernel module. SILK forms a replacement networking subsystem for standard Linux 2.4 kernels. Linux applications create and use Scout paths via the Linux socket interface with virtually no modifications to the applications themselves. SILK provides Linux applications with the benefits of Scout paths, including early packet demultiplexing, per-flow accounting of resources, and explicit scheduling of network processing. SILK also introduces the concept of an *extended path* to provide a framework for application QoS. We demonstrate the utility of SILK by showing how it can provide QoS for the Apache Web server.

1 Introduction

In recent years, many research efforts have focused on improving operating system architectures. Architectural features have been advanced to help systems avoid receive livelock and overload, fight denial of service attacks, account for kernel resources used on behalf of applications, and provide application QoS. Despite the quality of many of these efforts, their underlying ideas have not spread to the average desktop. For example, Linux is a popular operating system with freely available source. Yet the standard Linux networking stack still cannot prioritize among incoming network packets, and so a Linux application cannot take full advantage of QoS provided by the network. Nor can an MPEG player running in Linux inform the system that it requires a certain rate on the CPU to play its video. Researchers have solved these problems, but adoption of the mechanisms they have proposed, and the applications to exploit them, has been slow. The question is, how can research more quickly have an impact on the systems and programs that average people use every day?

We have identified several factors that inhibit the quick distribution and widespread evaluation of systems research ideas. They are:

- Steep learning curves. Several research systems have been built from scratch, usually around better abstractions and architectures, and are freely available. These systems often possess real advantages. However, most people are reluctant to invest the time to master a new system.
- Lack of applications. This problem plagues research systems that are built from the ground up, but also affects any research idea that requires heavy changes to existing applications. The issue is one of chicken-or-egg: new mechanisms will not be quickly adopted if few applications can use them effectively, yet people will not spend effort rewriting applications to take advantage of mechanisms that are unavailable.
- Kernel patches. Most good research efforts are targeted at specific problems. However, a sysadmin may want to combine a number of these solutions in her system. She faces unpredictable results if she downloads six kernel patches from six different research projects and applies them all at once—there could be feature interaction, or the patches themselves might conflict. Also, a patch may not be available for the kernel version she is using.

This paper presents SILK, which stands for Scout In the Linux Kernel, in response to these problems. SILK makes three contributions. First, SILK is a replacement networking subsystem for Linux based on the Scout path architecture [18]. Scout paths combine a number of widely advocated research ideas into a simple, clean system abstraction. Applications interact with SILK through the Linux socket interface. Second, SILK provides a QoS framework through

[◇] Princeton University, [acb,mhw,llp]@cs.princeton.edu
[*] Swedish Institute of Computer Science (SICS) and Uppsala University, thiemo@sics.se
[‡] Uppsala University, perg@docs.uu.se

the idea of an *extended path*. SILK conceptually extends the path from the network to the application to coschedule application and network processing. We believe that many current applications can take advantage of this framework with minimal modifications, providing them with immediate benefits. Third, SILK is packaged as a kernel module that can be loaded into a standard Linux 2.4 kernel. SILK allows people to experiment with advanced research ideas with very little effort and risk, and serves as a vehicle for widely distributing these ideas and evaluating them in real contexts.

We demonstrate the capabilities of SILK using the Apache Web server, a popular and complex application. Our results show that SILK's performance is competitive with the native Linux network stack, and that Apache with SILK can provide nearly constant response time for preferred requests independent of the total number of clients accessing the server.

The rest of the paper is organized as follows. Section 2 presents an overview of Scout paths, which form the heart of SILK, and shows how SILK fits into Linux. Section 3 describes important parts of the SILK design. Section 4 presents a set of experiments done with Apache, to show that SILK can provide benefits to a widely-used application. Section 5 discusses further issues and future work, and Section 6 discusses other work related to SILK.

2 Overview

2.1 Scout

Scout [18] is a modular, configurable, communication-oriented operating system developed for small network appliances. Scout was designed around the needs of data-centric applications with particular attention given to networking. It incorporates a number of ideas found in other network architectures as well. They are:

Early demultiplexing of incoming packets to flow queues. This allows the system to isolate flows as early as possible, in order to prioritize packet processing and accurately account for resources.

Early dropping when flow queues are full. The server can avoid overload by dropping packets before investing many resources in them.

Accounting of the resources used by each data flow, including CPU, memory, and bandwidth. Knowledge of the resources used by a flow is necessary in order to provide overall fairness or to place resource limits on individual flows.

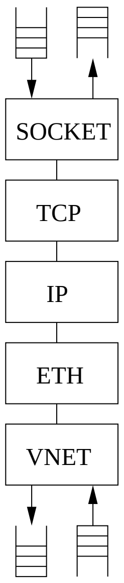

Figure 1: A Scout path

Explicit scheduling of flow processing, including network processing. Scheduling and accounting are combined to provide resource guarantees to flows; for example, CPU or bandwidth reservations.

Extensibility through Scout's modular design. This makes it easy to add new protocols and construct new network services. Different protocol versions can exist side-by-side. A new service can be deployed by specifying a sequence of modules for a data flow.

Scout's main contribution is to combine all of the features listed above into a single, clean abstraction: the *path*. A path is a structured system activity. Each Scout path encapsulates a flow of data, for example, a single TCP connection. A path consists of a string of code modules that process and perhaps transform the data as it flows through the system, and all resources consumed on the flow's behalf are charged to the path. Previous research has demonstrated the usefulness of Scout paths for distributing multimedia processing across configurable network nodes [20], scheduling packet processing in a software router [23], and for protecting against denial of service (DoS) attacks [25].

Figure 1 shows a picture of a Scout TCP path. The path corresponds to a single TCP connection. It consists of a chain of protocol modules that process packets belonging to the connection, with input and output queues at each end. When a packet arrives on the network device VNET, it is demultiplexed based on its header information to locate its corresponding path. If the packet belongs to the TCP connection of this path, it is placed in the input queue at the bot-

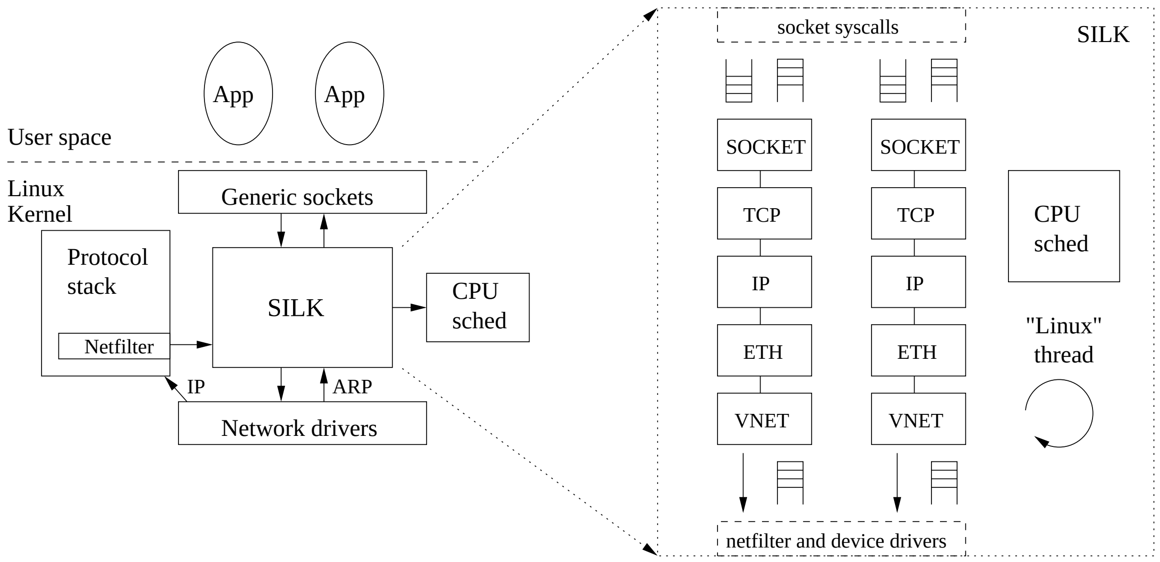

Figure 2: The SILK module in Linux

tom of the path; if the queue is full the packet is dropped. A path is considered *ready* to run once it has data in its input queue.

The Scout scheduler chooses a path to run from among those that are ready. Scout provides a number of CPU schedulers for scheduling paths; fixed priority, Earliest Deadline First (EDF), Weighted Fair Queueing (WFQ), and Best Effort Real Time (BERT) schedulers can be configured in Scout. When a path is run, a thread belonging to the path dequeues a piece of data from its input queue, runs the code modules in sequence, and deposits the result in the output queue at the opposite end of the path. Scout can prioritize among different data flows using a fixed priority scheduler, or give each a CPU share with a WFQ scheduler. The combination of paths and configurable scheduling allows Scout to produce a rich variety of system behaviors.

2.2 SILK

SILK is an encapsulation of Scout in a Linux kernel module. SILK provides a drop-in, extensible networking subsystem for Linux based on Scout paths. SILK also provides a framework for building application-level QoS solutions through the concept of an *extended path*. In this section we give a high-level view of SILK.

Figure 2 shows the SILK kernel module within the Linux kernel. In the left portion, the SILK module exchanges data with the network device drivers, the socket interface, and the packet filter interface `netfilter` in standard ways. SILK also modifies the scheduling parameters of Linux tasks to influence the decisions made by the Linux CPU scheduler. The right part of the figure shows two Scout paths within SILK, corresponding to two TCP connections. (In the remainder of this paper, when we refer to a "path" we mean a Scout path in SILK.) Each path has two input queues and one output queue; the lower output queue is unnecessary since the path sends outgoing packets straight to the device. [1] The SILK module also contains its own CPU scheduler that cooperates with the one in Linux. This cooperation is represented by the "Linux thread"; by executing this thread, SILK transfers control to the Linux scheduler. The next section will discuss each piece in more detail and describe how they fit together.

3 Design of SILK

We have three main design goals for SILK:

1. Minimal changes to Linux. Since we want SILK to be widely used it is important that it is able to run in an unmodified Linux kernel. Only minimal changes should be required to Linux applications to enable them to use paths in SILK.

2. SILK has control of the CPU. Our aim is to take advantage of the different CPU schedulers implemented in Scout. In particular, we want to be able to prioritize among paths, and to provide them with CPU guarantees.

[1] In other words, the output queue is the packet queue maintained by the network device driver.

3. Coscheduling of paths and applications. SILK should be able to schedule Linux tasks as well as paths to provide application-specific QoS.

This section describes how our design and implementation of SILK meets each of these goals.

3.1 SILK Sockets

Linux applications create and use paths in SILK through the Linux socket API. New applications can use the new PF_SCOUT protocol family to construct paths with experimental or non-standard protocols. SILK can also intercept socket calls on the PF_INET family, allowing unmodified legacy applications to access TCP and UDP paths. [2] In both cases, SILK implements socket operations and intercepts network packets using interfaces exported to kernel modules by Linux, and hence does not require any kernel patches. The rest of this section focuses on the TCP path shown in Figure 1, and describes how this path exchanges data with Linux.

The VNET module at the bottom of the path connects to the `netfilter` interface, the packet filtering interface provided by recent Linux kernels. Incoming packets belonging to a TCP path must be processed only by SILK and not by Linux. To accomplish this, SILK inserts a netfilter hook at the earliest possible location, before Linux has performed any IP processing; all incoming IP packets are diverted to this hook. SILK demultiplexes each packet to see if it matches a path. If it matches, then the packet is enqueued at the bottom of the path, the path is scheduled as described in Section 2.1, and Linux is instructed to drop the packet. If demultiplexing does not match the packet to a path, SILK lets Linux continue with network processing.

SILK uses another method to receive ARP packets (netfilter only handles IP packets). SILK snoops ARP packets directly from the network device using a lower-level interface than netfilter. This interface is not as powerful as netfilter since it is not possible to use it to filter packets. However, it is sufficient for implementing new network-layer protocols within SILK, because by default the Linux networking stack will drop packets it receives for unknown protocols.

At the top of the path, the SOCKET module connects to Linux using the generic socket interface in the kernel. SOCKET supplies routines that map the familiar socket calls into operations on paths. For example, `connect()` creates a path for a new TCP connection and then starts the three-way handshake; `recv()` reads data from the output queue of the path and passes it to user space; `send()` reads data from user space and enqueues it on the path's input queue; and `close()` tears down the connection and destroys the path. These hooks are straightforward.

An interesting point is that the SOCKET and VNET modules are actually specializations of a single Scout module called GenericNet. This module provides generic versions of the operations that always occur at path endpoints (e.g., demultiplexing and enqueuing messages), while allowing the module to be specialized for different contexts. For example, when the application invokes `send()`, the data is passed to SILK in a user-space buffer. The specialized code in SOCKET reads the data from user space and converts it to a Message (SILK's internal packet abstraction). Likewise, VNET converts the outgoing packet from a Message to an `sk_buff` (Linux's internal representation). GenericNet is also used to implement hardware device drivers in Scout.

3.2 CPU Scheduling

SILK contains its own CPU scheduler and thread package that coexists with the Linux scheduler. In this discussion, SILK runs *threads* within paths, while the Linux scheduler runs *tasks*. SILK implements thread scheduling on top of a Linux kernel task created when the SILK module initializes. As far as we are aware, we are the first to implement a self-contained thread package and scheduler in a Linux kernel module; we realize that this may be regarded as an abuse of the kernel module concept. This section describes how the SILK and Linux schedulers interact, i.e., how SILK controls the CPU.

SILK creates a Linux kernel task at startup and sets its priority to maximum realtime priority, the highest priority in the system. Therefore, the SILK kernel task will run almost immediately whenever it is runnable [3]. Kernel tasks run nonpreemptively in Linux, hence another task can be scheduled to run only when the SILK kernel task yields or sleeps. SILK multiplexes all of its threads onto this high priority kernel task.

SILK temporarily transfers control back to Linux through the "Linux" thread shown in Figure 2. This "Linux" thread is an actual thread in SILK and SILK can schedule it like any other thread. When SILK executes this thread, it causes the SILK kernel task to yield and thus transfers control to the Linux scheduler. Note that a Linux task that yields is not considered runnable again until another task has run. Therefore, when SILK executes the "Linux" thread, the Linux scheduler chooses one other task to run and then transfers control back to SILK. Through the mechanism of the "Linux" thread, SILK gains the ability to allow Linux to run one task.

The scheduling parameters assigned to the "Linux" thread determine SILK's policy for transferring control to

[2] This behavior is configurable. If on, SILK takes over all networking functions; if off, SILK and Linux networking exist side-by-side.

[3] It may have to wait for another kernel task to yield.

Linux. This policy will depend on which scheduler SILK is running—for example, with the WFQ scheduler the thread can be given a CPU rate of 50%, and this will cause SILK and Linux to evenly share the CPU. In this manner SILK controls how often Linux "as a whole" is allowed to run.

3.3 Extending the Path

One of our goals is to provide application-specific QoS to Linux programs that use SILK. A scheduling-aware application should be able to specify how it and its paths are scheduled by the system. On the other hand, since SILK is modular and configurable, intelligence can be built directly into SILK so that it can provide QoS to existing "dumb" applications without the application's participation or even its knowledge. To this end, SILK introduces the concept of an *extended path* to encompass coscheduling of applications and paths.

The idea is that, by giving the SILK scheduler the ability to control Linux's scheduling decisions, SILK can coordinate processing between the network stack and the application. SILK must be able to do two things to "extend the path" in this sense. First, it must identify the task associated with each SILK path. This is simply the task that calls `connect()` or `accept()`. Second, SILK must cause the Linux scheduler (running Linux tasks) to mirror the decisions made by the SILK scheduler (running paths). Exactly what form this cooperation takes depends on which scheduler SILK is using. We have implemented path extensions for SILK's fixed priority scheduler as follows.

We had two objectives when implementing an extended path for the priority scheduler. The first was that networked Linux tasks using SILK's networking stack should be scheduled at the same relative priorities as their corresponding paths. The second was that, since we want to provide these tasks with QoS, they should run at an absolute priority higher than other tasks in the system to avoid interference from these tasks. Note that the Linux scheduler usually provides some form of fairness to tasks. When choosing a task to run, Linux takes its priority (i.e., as set by `nice`) into account but does not strictly schedule by task priority alone. However, Linux can be made to perform strict priority scheduling by using the realtime priorities. A task with realtime priority p runs at a higher priority than a realtime task of priority less than p as well as any non-realtime task. Realtime priorities meet both of our objectives, and hence we map SILK priorities onto Linux realtime priorities.

SILK forms an extended path by mirroring the network path's priority in the realtime priority of the Linux task that uses it. For example, if a path has priority 2 (in SILK) then a Linux task reading from it would inherit a realtime priority of 2 (in Linux). Furthermore, the priority inherited by a Linux task from a path can change over time. A task that blocks on `accept()` first receives the priority of the SILK listen socket's path. Then it adopts the path priority of the socket returned by `accept()` and finally it returns to its original priority when closing the socket. SILK does not change the priorities of Linux tasks, except for the ones that use SILK paths.

Priority inversion will result if SILK chooses to schedule a path when there is a runnable Linux task with a higher priority. To avoid this, the SILK kernel task yields to Linux when a runnable task has a higher priority than any ready path. For each priority p starting with the highest, if there are no ready paths with that priority, then the SILK priority scheduler checks a list of Linux tasks having realtime priority p to see if one is runnable. If so, then SILK runs the Linux thread described in Section3.2. This causes the SILK kernel task to yield, and the Linux scheduler then runs one of the tasks with realtime priority p. If SILK finds nothing runnable at any priority, it runs the Linux thread by default; this allows Linux to schedule a task unrelated to SILK. In this way, SILK socket priorities are inherited by Linux tasks and SILK controls the scheduling of Linux tasks as well as paths.

4 Evaluation

SILK is an entire networking subsystem and not just an architectural feature. As such, it introduces new implementations of network protocols and scheduling algorithms into Linux. Because SILK itself encompasses so much, and because the benefits of Scout paths have been shown elsewhere, our experiments focus on demonstrating SILK's high-level behavior with a real application: the popular Apache web server. Our evaluation focuses on three areas: performance, prioritizing network processing, and providing service differentiation through extended paths.

Our testbed consists of a server and two traffic generators. The server machine is a 1.4 GHz Athlon with 256 MB of memory running SILK on Linux 2.4 and Apache version 1.3.20. We are running the standard Apache configuration unless explicitly mentioned. The traffic generator machines are both Pentium IIIs, running at 733 MHz and 600 MHz. All three machines have Netgear GA622T Gigabit Ethernet cards and are connected via a 1 Gb/sec Ethernet switch. Our desire is to stretch the limits of SILK by using muscular machines and fast networks.

As a traffic generator we use sclient [4]. Sclient uses a single process to manage a large number of concurrent connections to the server. In our version, sclient makes a request to the server and waits for the response. Immediately after receiving the response, sclient makes a new request to the server. By increasing the number of concurrent connections (referred to as clients), the load on the server can be increased. We use up to 97 clients in our experiments.

4.1 Performance

In the first set of experiments we compare the performance of Apache on SILK against Apache on Linux. Our goal is to show that, for a real application, the overall performance of Linux and SILK are comparable. This would provide evidence that SILK can be a viable network subsystem replacement for Linux.

Two metrics are important when measuring a Web server's performance [7]: overall throughput and response time. The throughput measured in requests served per second provides a good indication of the efficiency of the system. The response time or latency is the time measured by the client from when it opens a connection until the last byte of the response arrives. Response time is a measurement of the usability of the system, since people perceive long delays as unacceptable [8]. Each data point in our experiments represents an average of the observed throughput or response time over a run of 2 minutes.

Our experiments measure the throughput and latency for a group of clients making requests to the server. For each run of the experiment we configure sclient with a static number of clients, all of which request the same file. We vary the number of clients and the size of the requested file across runs. Note that we do not use a more realistic request pattern because our goal is to stress different aspects of the networking stack, and constant file sizes are more useful for this purpose. Shorter file sizes place more emphasis on per-connection overheads [6], for example, SILK path creation and destruction. On the other hand, inefficiencies in the SILK protocol stack, such as excessive data copying, should be more visible using large files.

For this experiment we configure all SILK paths with equal priorities. Since the SILK kernel task runs at a higher priority than any Linux task, this means that all paths are prioritized equally within SILK but have higher priority than Linux tasks. In Linux this is also the case because protocol processing occurs in a high-priority interrupt context. Therefore, we expect that SILK and Linux will behave similarly. In particular we expect that the overall throughput for both Linux and SILK remains roughly the same regardless of the number of clients. The response time should increase linearly with the number of clients, since we would expect each client to receive about $1/n$ of the resources when n clients are active simultaneously.

Figures 3 and 4 present the results for SILK and Linux. Figure 3 shows results for three small files of sizes between 1 KB and 50 KB. The bottom graph shows the throughput and the top graph shows the response time. The figure shows that the throughput of SILK is slightly lower and the response time slightly higher than for Linux, except for 1 KB files, where for some reason SILK is faster than Linux. The same measurements for three large files between 100 KB and 500 KB are shown in Figure 4. Ignoring the 1 KB file results which we feel to be anomalous, these graphs show that SILK's networking performance is marginally slower than Linux.

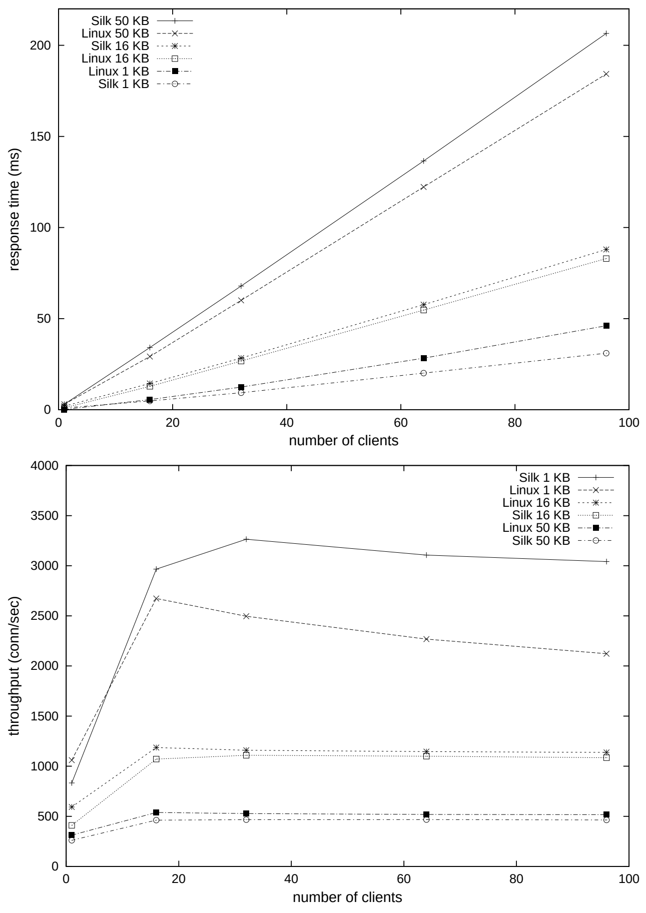

Figure 3: Comparison between standard Linux and SILK: response time and throughput for small files

We are aware of inefficiencies in SILK that can affect performance. For example, SILK uses Scout Messages as its packet abstraction. This requires copying data between a Linux `sk_buff` and a Message when moving a packet between SILK and Linux. Byte copying is one of the most expensive operations for Web servers [19]. The copy could be avoided by integrating Messages and `sk_buff`s; this optimization is in progress. Also, path creation and destruction are fairly heavyweight operations. Table 1 shows some microbenchmarks for SILK averaged across runs for different file sizes. We note that caching SILK paths for reuse could save considerable overhead, especially when the server is handling thousands of requests per second. Packet demulti-

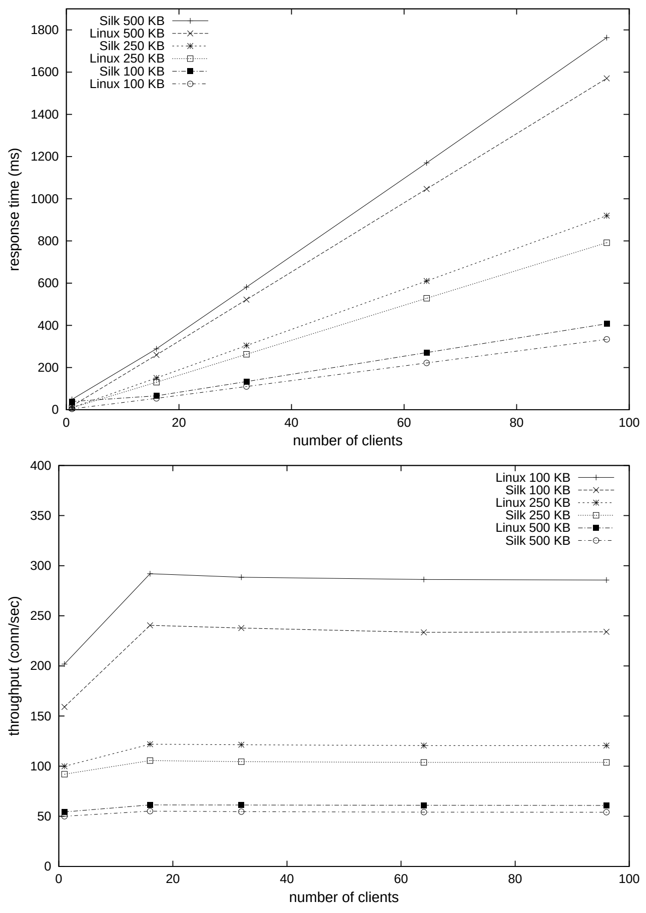

Figure 4: Comparison between standard Linux and SILK: response time and throughput for large files

plexing is already fairly cheap, but since demultiplexing occurs in an interrupt context, it should be further optimized in order to decrease the vulnerability of the system to receive livelock [17]. Finally, it is much more expensive to copy data from a Message (which can be fragmented across multiple buffers) than from an `sk_buff`. SILK performs the more expensive copy when sending an outgoing packet; this overhead represents a significant performance hit for a Web server. In summary, there is room for optimization in SILK, but its overall performance appears to be respectable.

4.2 Prioritizing Paths

SILK sockets provide the ability to schedule network processing on a per-connection basis. For instance, if the underlying network provides some quality of service (traffic priorities or bandwidth reservations), SILK can extend this QoS to the application itself. One question is how much current non-QoS-aware applications can benefit from this

Description	**Avg Time in μs**
Path create	41.5
Path destroy	12.3
Packet demultiplexing	1.8
Copy from Msg to sk_buff	4.3
Copy from sk_buff to Msg	1.1

Table 1: Microbenchmarks for SILK

capability. In this section we investigate allowing SILK to schedule only paths and not Apache.

For the experiment there are two classes of requests: preferred and standard. We differentiate between these based on source IP address. This determination could also be based on information gathered from the network (e.g., the Type of Service field in the IP header), or by the URL requested or an embedded cookie.

We assign the following priorities to SILK paths: paths handling preferred requests have priority 2, the listen socket's path has priority 1, and standard request paths have priority 0. [4] These priorities affect network processing as follows. All incoming SYN packets are delivered to the listen path. When the listen path runs, the source IP address of the SYN is inspected and a new path of the appropriate priority is created to handle the request. We chose priority 1 for the listen path because SYNs belonging to both request classes arrive on this path. We wanted to give preferred SYN packets a higher priority than standard connections, yet ensure that standard SYN packets had a lower priority than preferred connections. Note that only the network processing done in SILK is prioritized in this way. Linux runs the Apache server processes as usual, and the "Linux" thread in SILK runs at priority 0.

We repeat the experiment in Section 4.1 while adding a client generating preferred requests. We run sclient on two host. The first runs a static number of standard clients requesting the same file as before. The second runs one client also requesting the same file but these requests are preferred. Again, we vary the number of standard clients and the requested file size across runs. We also increase the number of Apache server processes to 100; using the low standard number of Apache server processes, the time requests spend in the listen queue dominates the response time. We would expect to see some improvement in the response time of the preferred requests.

The results in Figure 5 compare the response times of preferred requests in this experiment to the response times we observed in Experiment 4.1 for four representative file sizes. The x-axis denotes the number of standard clients competing with the preferred client. Our initial assumption

[4]Higher values mean higher priority.

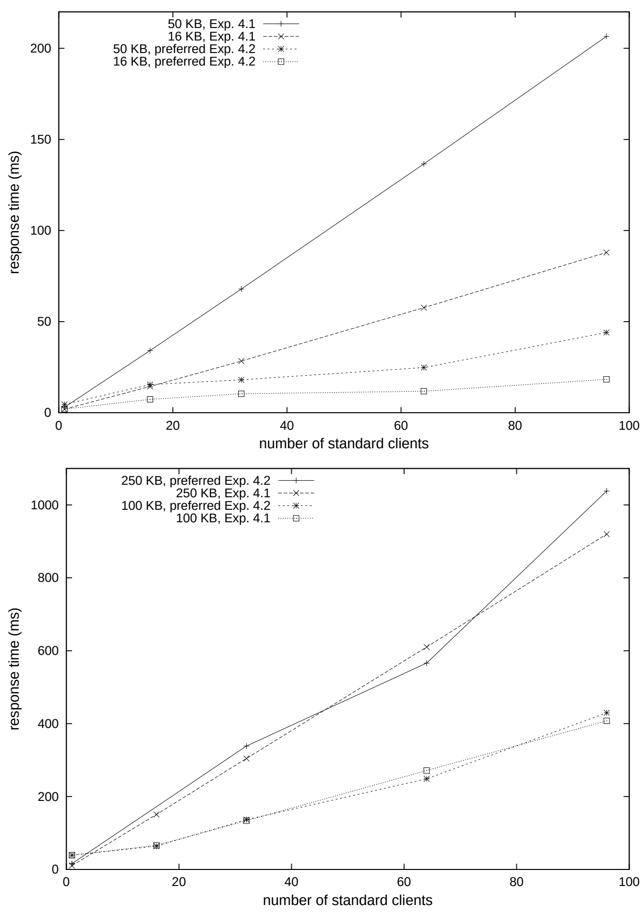

Figure 5: Comparison between response time for small (top) and larger file requests (bottom) in the first two experiments

was that prioritizing paths in SILK would show a greater improvement for preferred requests of large files, since they require more network processing. However, the results clearly show a benefit for small files and none for large files. The second column in Table 2 shows the reason why. An Apache server task that is sending a large file will repeatedly fill up the socket send buffer and block. When the send buffer opens and the task is unblocked, some time elapses before it is scheduled to run again. Since Linux tries to schedule all processes fairly, this time increases proportionally to the total number of processes in the system. In this case the Linux scheduler dominates the response time for preferred requests. For small files, this delay was zero in all experiments because the entire file fit in the send buffer.

Number of standard clients	**Avg Time no path ext.**	**Avg Time with path ext.**
1	387.7 μsec	39 μsec
16	18.0 msec	46.6 μsec
32	32.6 msec	47 μsec
64	50.1 msec	53 μsec
96	57.5 msec	58 μsec

Table 2: Average time elapsed between when a server task unblocks and when it runs, for preferred requests and large files only

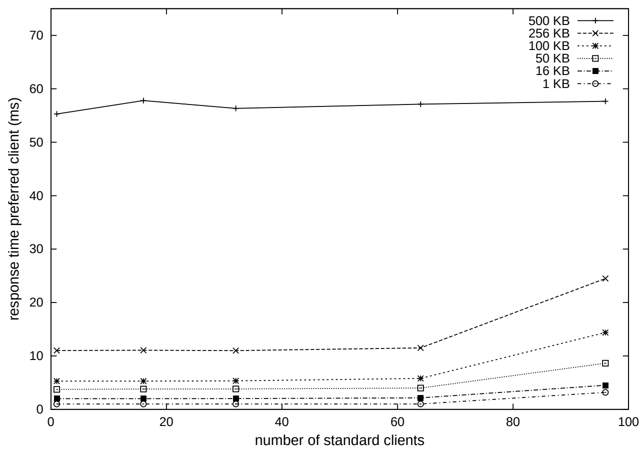

Figure 6: Response time with path extensions

4.3 Extending the Path

An *extended path* is SILK's abstraction for coscheduling applications and paths. In this section, we redo the experiment of the previous section while enabling extended paths. This time, SILK changes the realtime priority of a Linux task to reflect the (SILK) priority of the path that the Linux task "extends". In the current setup, this means that a server task servicing a preferred request runs at realtime priority 2, a process waiting on the listen queue has realtime priority 1, and a process servicing a standard request is given realtime priority 0. We expect that this will remove the Linux scheduler as the bottleneck for providing better service to large preferred requests.

The results are shown in Figure 6. The response times are very low even for the larger files. For example, with 96 standard clients the response time for a 250 KB preferred request has decreased from about one second in the previous experiment to less than 25 milliseconds. In fact, though the numbers in the figures are averages, the maximum response time seen by a preferred client is quite close to the average response time. For 96 clients and the 500KB file the

File size (KB)	Exp 1	Exp 2	Exp 3
1	3041.1	2752.8	2880.5
16	1084.4	974.5	992.7
50	463.2	434.7	440.35
100	234	215.2	221
250	103.8	103.5	103.9
500	54.1	53.2	54.2

Table 3: Throughput in conn/sec with 96 standard clients in the three experiments

average response time was 58 ms while the maximum was 98 ms. Also, the response times are almost independent of the number of competing standard clients. There is a small increase in the response time with 96 competing standard clients, and the main reason for this is that the Linux interrupt handler has higher priority than SILK. [5] The third column of Table 2 shows the average elapsed time that an unblocked "preferred" server task [6] waits to run with extended paths. Note that there is a difference of three orders magnitude compared to not using path extension with 96 standard clients. This is the reason for the improvements shown in Figure 6.

Table 3 shows the overall system throughput of SILK and Apache, with 96 standard clients for each of the three experiments detailed in this section. Recall that the first experiment had only standard clients, while the other two had 96 standard and one preferred client. We can see from the table that introducing priorities and path extension into SILK has a minor impact on the system performance. The throughput was slightly worse in the second experiment than in the first, and this is probably because in the second experiment we "penalized" the standard clients by lowering their priority from 1 (in the first experiment) to 0. The third experiment with extended paths shows overall higher throughput compared to the second, and for large file sizes the throughput is the same as in the first experiment.

In summary, we have shown that SILK can handle a high request load, give low response times to preferred requests, and all without rewriting Apache.

[5]Another noticeable fact is that, based on the results for other file sizes, we would expect the response time for the 500KB file to be lower. From our detailed timing log we could see that this result was due to an interaction between the server's TCP implementation and the client's Linux TCP, and was not caused by any scheduling overhead.

[6]In other words, a server task servicing a preferred request.

5 Discussion and Future Work

5.1 Collisions with Linux

Scout has its origins as a standalone operating system. For this reason, both Scout and Linux independently manage resources that are actually shared between them. In order to realize SILK's goal of requiring no changes to Linux, it is necessary for the system adminiatrator to ensure that SILK and Linux do not collide in two areas: CPU scheduling and TCP port space.

The relationship between the SILK and Linux schedulers described in Sections 3.2 and 3.3 assumes that there are no other realtime Linux tasks. If this is not the case, careful thought must be given to the task's interaction with SILK when choosing priorities for it, the SILK kernel task, and tasks scheduled by path extensions. Such a discussion is beyond the scope of this paper.

SILK and Linux cannot automatically coordinate which TCP port numbers each is using without any Linux changes. If SILK chooses to use the same port as Linux (for instance, two Apache processes listening on port 80 with one using SILK sockets) then SILK will grab all of the packets matching that port and Linux will never see them. One solution is to manually partition the port space between SILK and Linux. If a minor change to Linux is permissible, then another solution could be to export the port management functions from the Linux kernel, and to modify SILK so that it could use them too.

5.2 System Priorities

Realtime resource kernels minimize the work done in interrupts, for the reason that interrupts run asynchronously and at a higher priority than any task. This can disrupt the system's ability to offer firm guarantees of timing behavior. Since Linux manages hardware devices for SILK, SILK socket processing can be preempted by Linux interrupts. Therefore, SILK can provide only soft realtime guarantees.

Many of the key ideas of Scout and other recent network architectures stem from the observation that today's operating systems cannot assign different priorities to network processing. The standard Linux kernel is no exception—all network processing occurs in a bottom-half handler that runs at interrupt priority. Packet demultiplexing to SILK sockets occurs in this Linux handler, but packet processing itself runs at the priority of the SILK kernel task. This means that, strictly speaking, Linux network processing runs at a higher priority than SILK sockets which can lead to receive live-lock [17]. However, our expectation is that systems running SILK will either use Scout networking exclusively, or at least minimize the amount of network traffic handled by Linux. A solution that may allow both SILK and Linux networking to coexist is to use polling of the network device; this capability already exists in Scout and we intend to port it to SILK.

5.3 Extending the Path

The extended path mechanism described in Section 3.3 assumes a one-to-one correspondence between Linux processes and SILK sockets. In contrast, recent Web servers such as Flash [22] and Apache 2.0 use a single process to manage multiple connections. Single-process servers with multiple connections must make heavy use of the `select()` system call. We believe that the extended path concept could be used in this environment by limiting the set of paths returned via `select()`. We have not yet implemented and evaluated this mechanism and leave it as future work.

Extended paths provide a framework for implementing application QoS. One interesting question is whether QoS policies reside in the application or in SILK itself. To allow applications to take advantage of special features in SILK, we are defining an API for configuring paths using the `setsockopt()` interface. For example, QoS-aware applications can specify path priorities and request extended paths through this interface. On the other hand, since SILK is a modular and configurable system, it can include application-specific policy modules for providing QoS to "dumb" legacy applications. We are also building a SILK-based QoS kernel module for Apache as a demonstration of this capability. We note that SILK provides flexibility as to where in the system we implement policy intelligence.

6 Related Work

SILK provides a network architecture for Linux based on Scout paths. Lazy Receiver Processing [11] incorporates many of the same ideas as Scout, including early demultiplexing and protocol processing at the priority of the receiving application. A difference is that LRP waits to process the packet until the receiver requests it, which in turn depends on when the system runs the receiving process; in contrast, through path extension SILK can implement high-level QoS by scheduling the receiver itself to run. In conjunction with LRP, Resource Containers [5] allow considerable flexibility in accounting for all system resources, including kernel processing, used on behalf of an activity. Resource Containers can be associated with multiple processes or network connections based on the structure of the application. The Scout path abstraction encapsulates a single data flow and is less general than a Resource Container. However, resource containers themselves are just an accounting mechanism, while Scout paths combine a number of ideas embracing aspects of both Resource Containers and LRP. Additionally, unlike SILK, LRP (implemented in Sun OS) and Resource Containers (originally implemented in FreeBSD and, more recently, Linux [1]) require significant changes to the OS kernel.

The coordination of application and kernel scheduling underlying the path extension concept has a long history, mainly with a focus on multimedia. Processor reserves [16] can be used to provide QoS for multimedia applications in a microkernel environment. Client applications make CPU reservations which are then guaranteed by the system, and work done by a server on a client's behalf is accounted to the client. In [14], Jeffay *et al.* propose early packet demultiplexing along with coordinated proportional share scheduling of both packet processing and user tasks to avoid receive livelock in overload. We believe that SILK configured with path extensions and WFQ scheduling very closely resembles their scheme.

Several research efforts have focused on building new operating systems around abstractions that support QoS for multimedia applications. Operating systems such as Nemesis [13] and Rialto [15] can provide finer-grained QoS guarantees to applications. However, systems that are built from the ground up often suffer from a lack of real applications which prevents their wider adoption.

Another approach is to modify existing operating systems to provide QoS to applications. Bruno *et al.* have implemented the Eclipse operating system into FreeBSD and support hierarchical proportional-share CPU, disk and link schedulers [10]. QLinux [12] incorporates hierarchical schedulers for CPU and network, LRP and an advanced disk scheduling framework. Both systems depend on particular versions of the hosting operating system.

Linux/RK [21] is an adaptation of the resource kernel concept, developed in RT-Mach, to Linux. A resource kernel provides applications with explicit guarantees to system resources through abstractions such as CPU Reserves. Linux/RK shares some goals with SILK, including modularity and minimal changes to Linux. Linux/RK also maps its own scheduling policies to Linux task priorities like SILK does; their experience may be relevant to SILK as we create path extension mechanisms for schedulers other than fixed priority. However, SILK is not a resource kernel, but rather a networking subsystem that supports coordinated network and application processing.

Scheduler Activations [3] address problems with multiplexing user-level threads onto kernel threads. Most of these problems stem from poor coordination of thread scheduling between the user and kernel domains. SILK sidesteps these issues by multiplexing its threads onto a Linux kernel task and then controlling how this task itself is scheduled. Since the task runs nonpreemptively, Linux cannot choose to take control away from SILK; since the task has the highest priority, Linux must always run this task when it is runnable. The extended path concept avoids priority inversions by further coordinating scheduling across the SILK and Linux schedulers.

A few papers in the area of Web server QoS deserve mention. *WebQoS* [9] is a middleware that provides service differentiation and admission control. Reumann *et al.* [24] have presented virtual services, a new operating system abstraction that provides resource partitioning and management. This approach relies on some kernel modifications. Almeida *et al.* [2] use priority-based schemes to provide differentiated levels of service to clients depending on the Web pages accessed. While in their approach the Web server determines request priorities, Voigt *et al.* [26] provide mechanisms for QoS and overload protection that reside in the kernel and can be applied without context-switching to user level. Our approach goes even further than the architectures described above since we can apply scheduling policies already during the TCP connection setup.

7 Conclusions

SILK is a replacement networking subsystem for Linux based on Scout paths. “Extended paths” provide a QoS framework through coscheduling applications and paths. We have demonstrated that the performance of SILK is comparable with Linux for the Apache Web server, and shown how extended paths can be used to provide differentiated service for Web requests. SILK can serve as a platform for research in network protocols, resource management, CPU scheduling, and QoS policies. Finally, it works with an unmodified Linux 2.4 kernel and existing applications, and hence provides a vehicle for distributing research solutions so that they can be widely used and evaluated. The SILK code will be freely available and is scheduled for release in early 2002.

References

[1] M. Alicherry and K. Gopinath. Predictable management of system resources for linux. In *Proc. of Usenix Annual Technical Conference*, Boston, MA, USA, June 2001.

[2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *Proc. of Internet Server Performance Workshop*, March 1999.

[3] Tom Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[4] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, December 1997.

[5] G. Banga, P. Druschel, and J. Mogul. Resource containers: a new facility for resource management in server systems. In *Proc. of OSDI*, February 1999.

[6] G. Banga and J. Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proc. of Usenix Annual Technical Conference*, New Orleans, LA, USA, June 1998.

[7] Paul Barford. Web server performance analysis. Tutorial at ACM SIGMETRICS, May 1999.

[8] N. Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. In *9th International World Wide Web Conference*, Amsterdam, May 2000.

[9] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, September 1999.

[10] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting quality of service into a time-sharing operating system. In *Proc. of Usenix Annual Technical Conference*, Monterey, CA, USA, June 1999.

[11] P. Druschel and G. Banga. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proc. of OSDI*, pages 91–105, October 1996.

[12] P. Goyal, J. Sahni, P. Shenoy, R. Srinivasan, H. Vin, and T. Vishwanath. Qlinux. http://www.cs.umass.edu/ lass/software/qlinux/.

[13] I.M.Leslie, D.McAuley, R.Black, T.Roscoe, P.Barham, D.Evers, R.Fairbanks, and E.Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[14] K. Jeffay, F. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Real-Time Technology and Application Symposium*, pages 480–491, December 2–4, 1998.

[15] M. B. Jones, J. S. Barrera III, A. Forin, P. J. Leach, D. Rosu, and M. Rosu. An overview of the Rialto real-time architecture. In *ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[16] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE Int. Conference on Multimedia Computing and Systems*, May 1994.

[17] J. C. Mogul and K. K. Ramakrishan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of USENIX Annual Technical Conference*, January 1996.

[18] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proc. of OSDI*, pages 153–167, October 1996.

[19] E. Nahum, T. Barzilai, and D. Kandlur. Performance issues in WWW servers. In *Proc. of ACM SIGMETRICS*, May 1999.

[20] A. Nakao, A. Bavier, and L. Peterson. Constructing end-to-end paths for playing media objects. In *The Fourth IEEE Conference on Open Architectures and Network Programming*, April 2001.

[21] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Real-Time Technology and Application Symposium*, June 1999.

[22] V. Paj, P. Druschel, and W. Zwaenepoel. Flash: an efficient and portable web server. In *Proc. of Usenix Annual Technical Conference*, June 1999.

[23] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling computations on a software-based router. In *Proc. of ACM Sigmetrics*, June 2001.

[24] J. Reumann, A. Mehra, K. Shin, and D. Kandlur. Virtual services: A new abstraction for server consolidation. In *Proc. of USENIX Annual Technical Conference*, June 2000.

[25] O. Spatscheck and L. Peterson. Defending against denial of service attacks in scout. In *Proc. of OSDI*, February 1999.

[26] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proc. of Usenix Annual Technical Conference*, Boston, MA, USA, June 2001.