Kernel Support for Faster Web Proxies

Marcel-Cătălin Roşu Daniela Roşu
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights NY 10598, USA
{rosu,drosu}@watson.ibm.com

Abstract

This paper proposes two mechanisms for reducing the communication-related overheads of Web applications. One mechanism is *user-level connection tracking*, which allows an application to coordinate its non-blocking I/O operations with significantly fewer system calls than previously possible. The other mechanism is *data-stream splicing*, which allows a Web proxy application to forward data between server and client streams in the kernel with no restrictions on connection persistency, object cacheability, and request pipelining. These mechanisms remove elements that scale poorly with CPU speed, such as context switches and data copies, from the code path of Web-request handling.

The two mechanisms are implemented as Linux loadable kernel modules. User-level connection tracking is used to implement `uselect`, a user-level select API. The Squid Web proxy and the Polygraph benchmarking tool are used in the evaluation. With Polymix-4, a realistic forward proxy workload biased towards cache hits and small files, the reductions in CPU overheads relative to the original Squid (with `select`) are 52-72% for `uselect`, up to 12% for `splice`, and 58-78% for the two mechanisms combined. Relative to Squid with /dev/epoll, `uselect` provides 50% overhead reduction.

1 Introduction

The advent of the World Wide Web has motivated a large body of research on improving Web server performance. Work has focused on improving the performance of the TCP/IP stack [25] (e.g., NewReno, SACK, Limited Transmit), of the Web server architecture (e.g. ZEUS[43], Apache [2], Flash[30], Squid[27], SEDA[41]), and of the interface between them (e.g., `select`[4], `/dev/epoll`[34], `sendfile`[24]).

In spite of the recent progress, the ability of existing operating system architectures to handle communication intensive workloads remains limited for server 'in-the-middle' configurations, such as Web proxies, CDN servers, and Edge Servers. First, these servers handle a large number of concurrent connections: to reduce transfer latencies for both cache hits and misses, proxies have to keep open connections to as many clients, peer caches, and origin servers as possible [8]. Second, a significant ratio of the requests arriving at forward Web proxies require transfers from origin servers or peer caches [42]. In serving these requests, content is transferred from server to client connections by copying it twice between kernel and application address spaces. Reverse Web proxies perform a similar processing, but cache misses represent a smaller fraction of their load.

Commercial operating systems include limited support for high-performance user-level Web proxies. Besides `sendfile`, the event notification mechanism `/dev/epoll` is being considered for inclusion in the Linux 2.6 kernel and a splice service for TCP connection tunneling is included in the AIX 5.1 kernel. However, the existing support is not sufficient, and, as a result, network appliances are used in many high-traffic proxy installations. Appliances are carefully optimized for I/O intensive workloads, but compared to general-purpose servers, have higher costs and limited flexibility. We submit that extending general-purpose operating systems with support targeted for Web proxy caches will boost the performance of off-the-shelf Web proxy applications and cache infrastructures, like Squid [27] and IRCache [28], respectively.

This paper proposes enhancing general-purpose operating systems with two mechanisms, *user-level connection tracking* and *data stream splicing*. These mechanisms enable Web applications to reduce their communication-related overheads by reducing the number of system calls and the amount of data copied between user and kernel domains, operations that are known to scale poorly with processor speeds [1, 29].

User-level connection tracking allows an application to coordinate its non-blocking network operations and to monitor the state of its connections with minimal switching between application and kernel domains. The mechanism is based on a shared memory region between kernel and application in which the kernel propagates elements of the application's transport and socket-layer state, such as the existence of data in receive buffers or of free space in send buffers. The application can access these state elements directly, without system calls. The mechanism is secure as only information pertaining to the application's connections is provided in its memory region. The mechanism can be used to implement low-overhead versions of the `select/poll` APIs, as well as new connection-tracking APIs. For instance, with socket-buffer availability propagated as actual number of bytes, applications can perform more efficient I/O by issuing I/O operations only when the number of bytes that can be transferred is greater than a specified threshold. Similarly, with transport-layer state like congestion window size and round-trip time, applications can learn about the latency characteristics of their connections and selectively customize their replies to improve the client response times [18].

Data-stream splicing allows an application to perform data forwarding between corresponding server and client TCP streams in the kernel, at the socket level. This proposal is the first to address the whole range of data transfer operations performed by a Web proxy cache application, supporting persistent connections, cacheable content, pipelined requests, and tunneled transfers. This mechanism draws significant benefits from the decision to implement it at socket level. Compared to user-level data forwarding, the mechanism eliminates data copies and system calls [11, 37]. Compared to IP-level alternatives [21, 39], the mechanism can be applied to data streams with different TCP connection characteristics (e.g., SACK) and it provides the application with full and efficient control over unsplicing and payload caching.

The two mechanisms are implemented in Linux and are evaluated using Squid [27], a popular Web proxy cache application, and Polygraph [40], a benchmarking tool for Web proxies. User-level connection tracking is used to implement a user-level wrapper for the native `select` system call, called `uselect`. Microbenchmarks demonstrate that `uselect` enables reductions in CPU utilization of 60-95% relative to `select` and of 20-90% relative to `/dev/epoll` for 4-128KByte objects and 100% cache hits. Data-stream splicing enables overhead reductions of 10-70% for a workload with 100% cache misses. With Polymix-4, a realistic forward-proxy workload biased towards small file sizes and cache hits, the reductions in CPU overheads relative to the original Squid (using `select`) are 52-72% for user-level select, up to 12% for splice, and 58-78% for the two mechanisms combined.

While our mechanisms have been proposed and evaluated in the context of Web proxies, they can benefit a wider range of applications. CDNs, Edge Servers, and internet applications based on SEDA [41] can benefit from low-overhead access to the socket- and network-layer state of their connections. Peer-to-peer infrastructures that include content forwarding, like Squirrel [14], can use the data-stream splice to lower node overheads.

Modular implementations of the networking stack and the socket layer enable simple implementations for the two mechanisms as loadable extension modules. Our approach is to replace methods of the transport and socket layers with new implementations, or with wrappers for the original methods. The Linux implementation presented in this paper does not require modifications of the kernel source tree.

The remainder of this paper is organized as follows. Sections 2-3 describe the proposed mechanisms. Section 4 describes the experimental testbed and methodology. Section 5 presents the results of the experimental evaluation. Section 6 discusses related research and Section 7 summarizes our contributions.

2 User-level Connection Tracking

Experiences with communication-intensive applications, such as Web servers, demonstrate that restricting the number of kernel threads used by the application is critical to achieving good performance. The most efficient architectures are event-driven [27, 30] because, by avoiding blocking I/O operations, they handle a large number of connections with a small number of control threads. Efficient non-blocking I/O requires a mechanism for tracking connection state, such that I/O operations are issued only when guaranteed to be successful.

The traditional OS mechanisms for connection-state tracking, `select` and `poll`, retrieve connection state from the kernel by performing two context switches and two data copy operations; the amount of data copied is proportional to the number of existing connections. Recently proposed event delivery mechanisms [5, 19, 34] allow more efficient in-kernel implementations, avoid the application-to-kernel data copy, and even the kernel-to-application copy [33, 34]. However, the benefits of these optimizations are partially offset by the possible increase in the number of system calls since the application has to register and cancel its 'interests' for every socket. The additional system calls are shown to represent a relatively high overhead for sockets with short lifetimes [19].

The *user-level connection tracking* mechanism proposed in this paper attempts to further reduce the number of system calls related to connection-state tracking and to extend the set of connection-state elements that applications can exploit. The approach is to propagate certain elements of a connection's socket- and/or transport-layer state at the user level, in a memory region shared between the kernel and the application (see Figure 1). The application can retrieve the propagated state using memory read operations, without any context switches and data

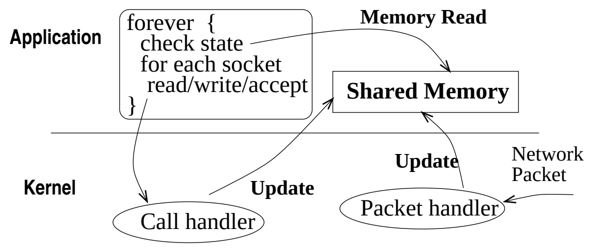

Figure 1: User-level connection tracking.

copies. The mechanism is secure because each application has a separate memory region which contains only information pertaining to the application's connections. The mechanism does not require any system calls for connection registration/deregistration. All connections created after the application registers with the mechanism are automatically tracked at user-level until closed.

The mechanism allows for multiple implementations, depending on the set of state elements propagated at user level. For instance, in order to implement the `select/poll`-type connection state tracking APIs, the set includes elements that describe the states of send and receive socket buffers. Similarly, the representation of state elements in the shared memory region depends on the implementation. For instance, for the `select/poll`-type tracking, the representation can be a bit vector, with bits set if read/write can be performed on the corresponding sockets without blocking, or it can be an integer vector, with values indicating the number of bytes available for read/write.

The same set of state elements is associated with all of the application's connections. The data structures in the shared region should be large enough to accommodate the maximum number of files a process can open. However, the shared memory region is typically small. For instance, an application with 65K concurrent connections and using 16 bytes per connection requires a 1MByte region, which is a small fraction of the physical memory of an Internet server.

In addition to direct memory reads, applications can access the shared memory region through user-level library calls. For instance, when the shared state includes information on socket-buffer availability, the application can use user-level wrappers for `select/poll`. Such wrappers can return a non-zero reply using only the information in the shared region; otherwise, if parameters include file descriptors not tracked at user level or a non-zero timeout, the wrappers fall back on the corresponding system calls.

The kernel component updates the shared memory region during transport and socket layer processing, and at the end of read and write system calls (see Figure 1). The shared region is not pageable and updates are implemented using atomic memory operations. The cost of updating variables in the shared memory region is a negligible fraction of the CPU overhead of sending/receiving a packet or of executing a read/write system call.

The kernel component exploits the modular implementation of the socket and transport layers. In Linux, the socket layer interface is structured as a collection of function pointers, aggregated as fields of a `'struct proto_ops'` structure. For IPv4 stream sockets, the corresponding variable is `'inet_stream_ops'`. This is accessible through pointers from each TCP socket and includes pointers to the functions that support the read, write, select/poll, accept, connect, and close system calls. Similarly, the transport layer interface is described by a `struct proto` variable called `'tcp_prot'`, which includes pointers for the functions invoked upon TCP socket creation and destruction. Also, each TCP socket is associated with several callbacks that are invoked when events occur on the associated connection, such as packet arrival or state change.

In order to track a TCP connection at user level, the kernel component replaces some of these functions and callbacks; the replacements capture socket state changes, filter and propagate them in the shared region. Connection tracking starts upon return from the `connect` or `accept` system calls. To avoid changing the kernel source tree, in this implementation, the tracking of `accept`-ed connections starts upon return from the first `select/poll` system call.

User-level Tracking with select API. In this paper, we use the proposed connection-state tracking mechanism to implement `uselect`, a user-level tracking mechanism with the same API as `select`.

For this implementation, the shared memory region between kernel and application includes four bitmaps: the *Active*, *Read*, *Write*, and *Except* bitmaps. The Active bitmap, `A`-bits, records whether a socket/file descriptor is tracked, i.e., monitored, at user level. The Read and Write bitmaps, `R`- and `W`-bits, signal the existence of data in receive buffers and of free space in send buffers, respectively. The Except bitmap, `E`-bits, signals exceptional conditions.

The implementation comprises an application-level library and a kernel component. The library includes (1) `uselect`, a wrapper for the `select` system call, (2) `uselect_init`, a function that initializes the application and kernel components and the shared memory region, and (3) `get_socket_state`, a function that returns the read/write state of a socket by accessing the corresponding R- and W-bits in the shared region.

The `uselect` wrapper, consisting of about 650 lines of C code, is composed of several steps (see Figure 2). First, the procedure checks the relevant information available at user level by performing bitwise AND between the bitmaps provided as parameters and the

```
int uselect(maxfd, readfds, writefds,
      exceptfds, timeout) {
 static int numPass = 0;
 int nbits;
 nbits = BITS_ON(readfds& R-bits& A-bits)
    + BITS_ON(writefds& W-bits& A-bits)
    + BITS_ON(exceptfds& E-bits& A-bits);
 if(nbits > 0 && numPass < MaxPass) {
    adjust readfds,writefds,exceptfds
    numPass++;
} else {
    adjust & save maxfd,readfds,writefds,
       exceptfds
    nbits = select(maxfd,readfds,...)
    numPass = 0;
    if( proxy_socket set in readfds) {
     check R/W/E-bits
     adjust nbits,readfds,writefds,
       exceptfds
    }
 }
 return nbits;
}
```

Figure 2: User-level select.

shared-memory bitmaps. For instance, the `readfds` bitmap is checked against the `A-` and `R-`bitmaps. If the result of any of the three bitwise ANDs is nonzero, `uselect` modifies the input bitmaps appropriately and returns the total number of bits set in the three arrays; otherwise, `uselect` calls `select`. In addition, `select` is called after a predefined number of successful user-level executions in order to avoid starving I/O operations on descriptors that do not correspond to connections tracked at user level (e.g., files, UDP sockets).

When calling `select`, the wrapper uses a dedicated TCP socket, called *proxy socket*, to communicate with the kernel component; the proxy socket is created at initialization time and it is unconnected. Before the system call, the bits corresponding to the active sockets are masked off in the input bitmaps, and the bit for the proxy socket is set in the read bitmap. `maxfd` is adjusted accordingly, typically resulting in a much lower value; `timeout` is left unchanged. When an I/O event occurs on any of the 'active' sockets, the kernel component wakes-up the application which is waiting on the proxy socket. Note that the application never waits on active sockets, as these bits are masked off before calling `select`. Upon return from the system call, if the bit for the proxy socket is set, a search is performed on the `R-`, `W-`, and `E`-bit arrays. Using a saved copy of the input bitmaps, bits are set for the sockets tracked at user level and whose new states match the application's interests.

The `uselect` implementation includes optimizations not shown in Figure 2 for simplicity. For instance, counting the 'on' bits, adjusting the input arrays, and saving the bits reset during the adjustment performed before calling `select` are all executed in the same pass.

Despite the identical API, `uselect` has a slightly different semantics than `select`. Namely, `select` collects information on all file descriptors indicated in the input bitmaps. In contrast, `uselect` might ignore the descriptors not tracked at user level for several invocations. This difference is rarely an issue for Web applications, which call `uselect` in an infinite loop.

The `uselect` kernel component is structured as a device driver module, consisting of about 1500 lines of C code. Upon initialization, this module modifies the system's `tcp_prot` data structure, replacing the handler used by the `socket` system call with a wrapper. For processes registered with the module, the wrapper assigns to the new socket a copy of `inet_stream_ops` with new handlers for `recvmsg`, `sendmsg`, `accept`, `connect`, `poll`, and `release`.

The new handlers are wrappers for the original routines. Upon return, these wrappers update the bitmaps in the shared region according to the new state of the socket; the file descriptor index of the socket is used to determine the update location in the shared region.

The `recvmsg`, `sendmsg`, and `accept` handlers update the `R-`, `W-`, or `E-`bits under the same conditions as the original `poll` function. In addition, `accept` assigns the modified copy of `inet_stream_ops` to the newly created socket.

Replacing the `poll` handler, which supports `select/poll` system calls, is necessary in our Linux implementation because a socket created by `accept` is assigned a file descriptor index *after* the return from the `accept` handler. For a socket of a registered process, the new `poll` handler determines its file descriptor index by searching the file descriptor array of the current process. The index is saved in an unused field of the socket data structure, from where it is retrieved by event handlers. Further, this function (1) replaces the socket's `data_ready, write_space, error_report`, and `state_change` event handlers, and (2) sets the corresponding `A-`bit, which initiates the user-level tracking and prevents future `poll` invocations. On return, the handler calls the original `tcp_poll`.

The `connect` handler performs the same actions as the `poll` handler. The `release` handler reverses the actions of the `connect/poll` handlers.

The event handlers update the `R-`, `W-`, and `E-`bits like the original `poll`, set the `R-`bit of the *proxy socket*, and unblock any waiting threads.

Exploiting uselect in Squid. In order to use `uselect`, Squid is changed as follows. During initialization, before creating the accept socket, Squid invokes `uselect_init`; as a result, the accept socket is tracked at user level. In each processing cycle, Squid invokes `uselect` instead of `select` to determine the states of all of its sockets. Finally, when trying to prevent starvation of the accept socket during a processing cycle, Squid uses `get_socket_state` instead of `select` to check the ready-to-read state of this socket.

Overall, `uselect` enables Squid to eliminate a significant number of systems calls with very few code modifications. Furthermore, `uselect` reduces the overhead of the remaining `select` system calls through the use of the proxy socket.

3 Data-Stream Splicing

The data-stream splicing mechanism proposed and evaluated in this paper enables a Web proxy to forward data between its server and client connections in the kernel, with support for content caching, persistent connection, and pipelined requests. The mechanism helps reduce the number of context switches and data copy operations incurred when serving cache misses, POST requests, and connection tunnels.

In its basic functionality, the mechanism establishes, in the socket layer, a data path between two data streams, such that packets received on one stream are forwarded on the other stream immediately, in interrupt context. On servers with zero-copy networking stacks and adapter support for checksum computation, the payload of forwarded packets is not touched by the proxy CPU.

The proposed mechanism extends previous proposals [22, 37, 39] with support for the following functionality:

- request pipelining and persistent connections;
- content caching decoupled from client aborts;
- efficient splicing for short transfers.

The new socket-level splicing mechanism can establish bi- and unidirectional data paths. Figure 3 illustrates the corresponding data flows. In the bidirectional mode, the traditional model for in-kernel splicing [21, 22, 37], data received on either of the two connections is forwarded to the peer endpoint. Connection close events are also forwarded. Splicing terminates when both connections are in `CLOSE` state. Bidirectional splice is typically used for SSL tunneling; it cannot support request pipelining and persistent connections.

The unidirectional mode addresses these limitations. Data coming from one endpoint, the *source*, is forwarded on the peer connection, towards the *destination*, while data coming from *destination* is provided to the application and not forwarded to the *source*. Connection close events are not forwarded. Splicing terminates (1) after transferring a specified number of bytes, or (2) when the *source* closes the connection.

Unidirectional mode is typically used to support request pipelining and persistent connections for HTTP GET and POST. For instance, for an HTTP GET, the *source* is the origin server, and the *destination* is the client; the server response is forwarded inside the kernel from the server to the client connection, while additional client requests are handled by the application and, if necessary, forwarded to the server. After unsplice, the same client connection can be used to transfer cached objects or it can be spliced with a connection to a different server.

Optionally, in unidirectional mode, a copy of the transferred payload is provided to the application. This mode, called *KeepCopy*, enables a Web proxy to populate its cache while exploiting kernel-level data forwarding. The application receives the content via the traditional `read` interface. The input stream is not terminated if the client aborts its connection, thus the cache operation can be completed.

Our experience with Web proxy workloads led us to develop a splice API that minimizes system call overheads, even for short transfers. The approach is to allow the application to combine several system calls that are typically issued in sequence by the application, and to eliminate some of the remaining system calls. Specifically, the basic splice interface, which specifies the connections, the type of splicing, and the termination condition, can be combined with: (1) a write to the client connection, used for the HTTP headers and the first content segment, and (2) a read from the server connection in *KeepCopy* mode. In addition, an application can save a system call when not interested in acquiring the amount of forwarded data returned by the unsplice command. Namely, with the *AutoRelease* option set in the splice request, the kernel releases the splicing context upon termination, eliminating the need for an explicit unsplice command from the application. Also, in *KeepCopy* mode, the application can specify a minimum input size, which is used to reduce the number of read system calls.

Implementation. The implementation of data-stream splice, about 4000 lines of C code, is structured as a loadable module. The module maintains a description and state for each spliced connection. Connection descriptors are maintained in a hash table with hash entries computed from the addresses of related TCP sockets.

The application uses `ioctl` calls to issue *Splice* and *Unsplice* commands to the kernel. Parameters and results are represented in a *SpliceRequest* data structure. For *Splice*, this data structure identifies the two connections (*fd0*, *fd1*), the type of splicing (e.g., bi- or unidirectional, termination type, with or without *KeepCopy* and *AutoRelease*), and the data to send on *fd1* before splicing is initiated. For unidirectional splice, *fd0* is the source and *fd1* is the destination. The data structure also includes parameters used in one or more of the splicing modes, such as the payload limit, when termination is based on

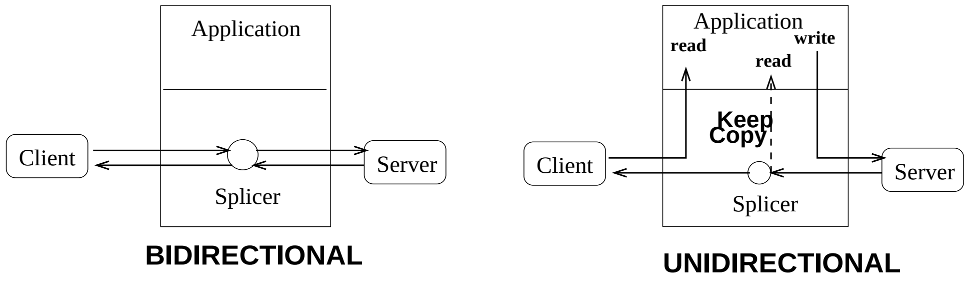

Figure 3: Types of splice interactions.

the transferred amount. For *KeepCopy*, the application can indicate the minimum input size, (*min2read*), the maximum amount of data forwarded and not yet read (*max2cache*), and an input buffer, to be filled by the *Splice* command if data is already available. *max2cache* helps prevent slow applications from overloading the kernel buffer cache.

After the *Splice* command, the application waits for notification of splicing termination, provided by the kernel as a `POLLOUT` event on the destination connection. This occurs when the transfer reaches the payload limit or the FIN packet is processed, and, for *KeepCopy*, when the application reads the last segment of the forwarded payload. If termination is due to connection abort, an error condition is signaled to the user.

For short transfers, the splicing can terminate in the *Splice* call. In this case, the application is informed by an appropriate return code and the splice context is released.

The application uses the *Unsplice* command to release the splicing context after termination notification or when it wants to abort the splice, such as on timeout or application shut-down. The *Unsplice* request specifies the two connections and it returns the number of bytes forwarded in each direction.

In the kernel, the splicing is established if the connections are in the `ESTABLISHED` or `CLOSE_WAIT` states. Upon splicing, the two TCP sockets are assigned new `data_ready`, `write_space`, `error_report`, and `state_change` handlers and a new `inet_stream_ops` data structure, with the `recvmsg`, `sendmsg`, and `poll` entries replaced.

The new `data_ready` and `write_space` handlers, invoked when data and ACKs are received, perform most of the data forwarding. Packet forwarding is started at end of the *Splice* call and it is driven by data and ACK packet arrivals on the two connections. Forwarding never stalls except for *KeepCopy* transfers when the amount of data forwarded and not read reaches *max2cache*. In this case, forwarding is restarted at the end of `recvmsg`.

For lower overheads, the TCP checksum is computed incrementally, from the old value, when forwarding an entire packet; it is computed from scratch only when forwarding a fraction of the packet, as it might occur at the beginning and the end of splicing.

The `recvmsg` and `sendmsg` handlers in the new `inet_stream_ops` are replaced with error-returning handlers when the corresponding operations are not permitted for the spliced socket. For instance, in bidirectional splicing, both handlers are replaced for both sockets. For unidirectional splicing, write is not permitted for the destination socket, and read is not permitted for the source socket unless splicing with *KeepCopy*.

For *KeepCopy*, the application uses the `read` system call, supported by the `recvmsg` handler, to retrieve a copy of the forwarded data. A local copy is created by cloning the packets, i.e., the `sk_buff`'s, and adding them to a list in the splice descriptor of the source socket. The new `poll` returns a `POLLIN` event when the local copy reaches the specified minimum input size (*min2read*) or when no more data is to be forwarded. The new `recvmsg` of the source socket transfers the data from the cloned `sk_buff`'s to the application buffer.

Using the mechanism in Squid. The original Squid 2.4 implementation handles cache misses as follows. The *http* module reads from a server connection in a 85KByte buffer allocated on the call stack. The data is transferred to the *store* module, which copies it in 4KByte memory blocks and notifies the *client-side* module of the new data arrival. The *client-side* processes the reply, generating the HTTP headers. If it can fill a 4KByte block, it copies data from the *store* into a send buffer, and it registers for a ready-to-write notification on the client socket. When the notification is received, the block is written, the completion routine updates the state, and the client registers to receive the next block from the *store*.

In order to exploit the data-stream splice mechanism for GET requests, we made the following changes. The *http* reads the first segment of a server reply in a 2896-byte buffer, attempting to consume only the first 2 MTUs. This size is chosen because: (1) a multiple of MTU size minimizes checksum overheads by enabling incremental checksum computation; (2) the included content and the proxy HTTP headers are below the 4Kbyte limit that can be sent in one operation by the *client-side*; and (3) a large fraction (approx. 40%) of the objects in proxy workloads can be received in one read operation.

After the read, if HTTP headers indicate that more data is expected, the *http* initializes a splice descriptor before sending the content to the *store* and does not prepare for a new read from the server. The splice descriptor is attached to the `StorageEntry` data structure, which is reachable from *http* and *client-side* request descriptors.

For a request with a splice descriptor, the *client-side* does not wait to fill the first 4KByte block before preparing for output. When it receives the ready-to-write notification on the client socket, it invokes the *Splice* command to write the available content and to initialize the in-kernel transfer; it does not register with the *store* to receive the rest of the content.

The *Splice* parameters define a unidirectional transfer, with the server connection as source and the client connection as destination. Splicing termination is set for a payload limit, if *ContentLength* is defined, or, otherwise, for the close of the server connection. The *AutoRelease* flag is set. If the content is cacheable, the *KeepCopy* flag is set and the related parameters are defined.

If the *Splice* command returns without error or indication of termination, the client socket is registered for (1) ready-to-write notification, to process the splicing termination, and for (2) ready-to-read notification, to receive the next request from the client. In *KeepCopy* mode, the server socket is registered for ready-to-read notification. Splice-related flags are set in both client and server socket descriptors in the global `fd_table` to ensure that spliced connections are handled appropriately in the close procedure, such as when called from timeout handlers. If *Splice* returns an error, the transfer resumes at application level.

The Squid handler invoked upon splicing termination performs the following actions. First, it checks if splicing termination was abnormal, checking the server socket EOF condition for *KeepCopy*, and the socket error otherwise. Next, it invokes the *client-side* procedure for write completion, providing the size of the spliced transfer. This restarts the activity on the client connection, activating the output for the next pending request or closing the connection. Finally, the server connection is added to the pool of persistent connections or closed, according to the request parameters. If splicing termination is abnormal, both connections are closed.

On a timeout, the handler issues an *Unsplice* command, calls the *client-side* write completion procedure, and closes both connections. We have also extended Squid to splice HTTP CONNECTs and POSTs. Details are available in [38].

Overall, with the proposed splice mechanism, a Web proxy can move the 'data path' for its cache misses (i.e., HTTP body transfers) into the kernel. This releases resources for the 'control path' (i.e., HTTP header processing) and cache management, which remain at user level.

4 Experimental Environment

The experimental evaluation of the two mechanisms proposed in this paper is conducted with Squid, a popular Web proxy application, and with Polygraph [40], a benchmarking tool widely used by the industry.

4.1 Hardware, System Software, and Apps

Our testbed comprises five nodes: three Polygraph nodes (two clients and one server), the Squid proxy, and a wide-area network emulator. Except for the WAN emulator which runs PicoBSD 0.445, all nodes run the Linux 2.4 kernel. An additional node, running the monitoring and data collection applications, is attached to the testbed. Table 1 describes the hardware configurations.

Table 1: Hardware configuration of the testbed.

	CPU Type	Speed (MHz)	Memory (MBytes)
PolyClient 1	PentiumIII	550	128
PolyClient 2	PentiumIII	667	256
PolyServer	PentiumPro	2x200	224
Web Proxy	PentiumIII	1000	512
WAN Emulator	PentiumPro	200	128

Except for the Polygraph server, all nodes are attached to a Gigabit Ethernet switch, the Alteon ACEswitch 180. The Polygraph server is attached via the dual-homed WAN emulator. The proxy node uses an Alteon Gigabit Ethernet adapter; the other nodes use Fast Ethernet adapters. The network links and the WAN emulator are never overloaded during the experiments. The network configuration is similar to that used in Polygraph Cache-Offs [23], with clients and servers on different subnets.

Polygraph. Web Polygraph 2.7.6 [40] is a benchmarking tool for caching proxies and other Web intermediaries. Polygraph includes high-performance HTTP clients and servers, realistic traffic generation and content simulation, and standard workloads. These standard workloads, including the Polymix-4 used in our evaluation, are widely accepted in the web caching community.

Each client and server node (agent) runs one or more 'robots', each robot handling one connection at a time, and possibly sending several requests in a connection. A client robot maintains a predefined request rate (e.g., 0.4 req/s). The overall request rate is determined by the number of client nodes, number of robots in a client node, and per-robot request rates.

In our experiments, we use Polygraph 2.7.6, modi-

fied to allow client and server applications to open up to 12,000 concurrent connections instead of the original 1024 limit; the kernel connection limit is set accordingly.
Proxy Application. The proxy application is Squid 2.4, extended to exploit the `splice` and `uselect` interfaces proposed in this paper, as well as the `/dev/epoll` interface used for comparison with `uselect`. Squid is a typical example of an event-driven application that can manipulate a very large number of communication streams. Squid is built around an infinite `select/poll`-loop. In each cycle, the application performs input and output operations on its active sockets, depending on connection and request processing states.

We modified Squid to support up to 64K file descriptors, more than the 1024 preset limit in Linux. In addition, we made several changes to improve its scalability under high load. These changes are orthogonal to the mechanisms evaluated in this paper and are deemed necessary because of our interest in driving the application at higher loads than previously published evaluations [40] on comparable hardware. For instance, we reduced the number of calls for memory allocation by extending the use of pre-allocated buffers.

Squid can use several models of disk cache management. In our experiments, we use the *diskd* and the *null* models. The *diskd* model uses daemons to perform the (blocking) disk I/O operations, one daemon for each disk. Squid communicates with a daemon through two message queues and a shared memory region; the message queues are used for operation descriptors and completion notifications, and the shared memory is used for the data blocks subject to I/O operations.

The *null* model emulates an infinite size, 0-overhead disk cache. There is no disk I/O and the list of cacheable objects read from the server is maintained in memory.

For `/dev/epoll`, we use *ep_patch-2.4.18-0.32*[20]. `/dev/epoll` is an efficient event-delivery mechanism which uses a shared memory between application and kernel to eliminate the data copy of notification results. However, it does not eliminate the system calls for event notification retrieval, and requires system calls for socket registration and deregistration. In order to use this interface, we define a new Squid procedure for connection-state tracking similar to the one used for `poll`, and we extend the `fd_table` to include flags for read and write availability. The new procedure performs the following steps. First, it traverses the file descriptor table, identifying the sockets in which the application is interested to read or write. For each of these sockets, the event types of interest are saved. Also, sockets not registered with `epoll` are registered at this time, indicating interest in all types of events; their new read- and write-availability flags are set. Second, `epoll` is invoked, and if there is any returned list of events, this is used to set the read/write availability flags of the indicated sockets. Third, the list of sockets in which the application has interest is traversed, and I/O operation(s) are performed if the corresponding availability flags are set. Availability flags are cleared when the corresponding I/O operations return blocking indications (i.e., `errno` is EAGAIN). Sockets are unregistered with `epoll` just before they are closed. The two traversals in steps one and three are also performed when using `uselect`, `select`, and `poll`.
WAN Emulation. To simulate WAN conditions, we use the same tools and settings as the Polygraph Cache-Offs. The WAN emulation tool is DummyNet [36]. For the proxy-server link, this tool introduces round-trip packet delays of 80 ms and packet losses of 0.05%. No delays or losses are introduced on the client-proxy links.

4.2 Experimental Methodology

The goal of our experimental study is to evaluate the performance of the proposed mechanisms and compare them with related mechanisms. The evaluation focuses on performance metrics like CPU utilization and response time. Towards this end, we use (1) microbenchmarks, in which the workload includes fixed size objects and the Web proxy does not perform disk I/O operations, and (2) realistic experiments, in which the workload is Polymix-4, a Polygraph workload representative for Web proxy caches, and the proxy stores cached objects on disks. Taking a high-level view at the benefits of these mechanisms, we do not attempt to quantify the individual components of the associated overhead reduction, such as data copies and context switches.

In microbenchmarks, we vary: (1) object cacheability, (2) hit ratio, (3) object size, (4) request rate, and (5) number of concurrent connections. In an experiment, all requests are HTTP GETs for objects of identical size and cacheability type (i.e., either cacheable or non-cacheable). The set of object sizes includes 4, 8, 12, 25, 64, and 128KBytes. The selection is related to the distribution of file sizes in Polymix-4, in which, with approximation, 4KBytes is the 50-th percentile, 8KBytes is the 75-th percentile, and 25KBytes is the 95-th percentile. Hit ratios are either 0% or (almost) 100%. Squid uses the *null* disk manager. Each data point represents three or more samples. For each sample, we collect statistics for 15min, after a 10min warm-up.

In the Polymix-4 experiments, we vary only the request rate. These experiments are similar to the Fourth Polygraph Cache-Off benchmarking [23], but with shorter phases. Namely, each experiment starts with an empty cache and takes 4h 30min. The fill phase runs at 160req/s for 90min. The first peak 'daily' load phase takes 25min, and the measurement peak 'daily' load phase takes 120min; the rest of the time is spent in ramp-up and down phases. Squid uses the *diskd* disk manager, with 4 disks, each with 3GByte of caching space. Each data point represents one sample.

In all experiments, request rates are selected such that client and server nodes are never overloaded.

For each experiment, we collect the statistics produced by the Polygraph agents, by `vmstat` and `tcpstat` running on the proxy node, and by Squid. Polygraph statistics include request rates and response times. `vmstat` provides information on CPU utilization, and `tcpstat` provides information on TCP transfers. Squid provides various statistics, such as rate of cache hits and number of open sockets.

All the plots of performance metrics for a single configuration include 90-percent confidence intervals, calculated with the T-student distribution. Note that for some experiments, the confidence intervals are very small, hardly visible on the plots. This is due to the workload model and the large number of requests in each run.

Polygraph Parameters. In all experiments, except for the parameters specified above, the Polygraph testbed is configured as for the Polymix-4 workload. Among its parameters we mention: client robot request rate of 0.4 req/s, and server delays normally distributed with a 2.5s mean and 1s deviation. The number of requests that a robot sends in a connection is drawn from a Zipf(64) distribution. Similarly, the server uses a Zipf(16) distribution to close active connections. The Polymix-4 workload has 58% hit rate, and about 70% cacheability ratio.

Squid Configuration. In all experiments, Squid runs with the default configuration, except for a few changes. No access log is maintained and no access control is performed, as in the Squid evaluation at the Third Polygraph Cache-Off [23]. The memory cache is 100MBytes (vs. 175MBytes used at the Cache-off). The *diskd* disk manager spins, waiting for request completion, if a daemon request queue is longer than 4K items, and it starts dropping file open requests when the queue exceeds 2K items. When using data-stream splice, all invocations use *AutoRelease*, 64K `max2cache`, and 16K `min2read`.

5 Experimental Evaluation

5.1 Microbenchmark: User-level Connection Tracking

Two microbenchmarks are used to compare the performance of the proposed `uselect`, with the `select` and `poll` system calls, and with the `/dev/epoll` event notification mechanism. The first microbenchmark evaluates the scalability with the number of active connections for a fixed file size, and the second microbenchmark evaluates the impact of file size for a fixed rate. For both microbenchmarks, the hit ratio is almost 100%,

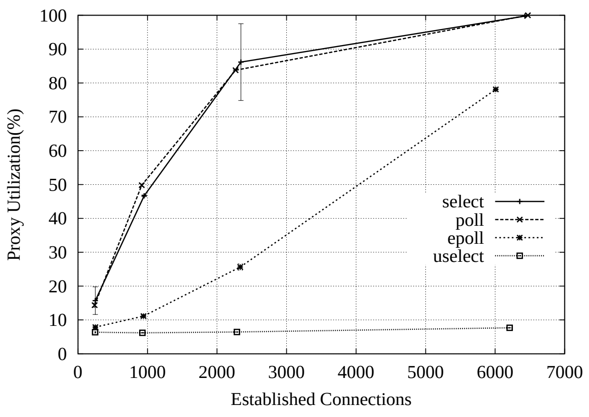

Figure 4: Proxy CPU utilization: 100%Hits, 100req/s, 8KByte files, null store.

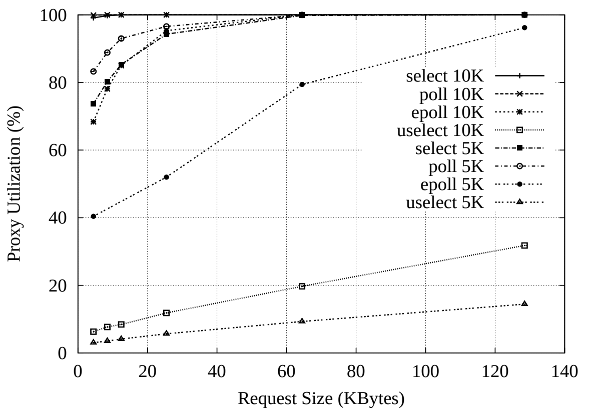

Figure 5: Proxy CPU utilization: 100%Hits, 5K robots at 50req/s and 10K robots at 100 req/s, null store.

which emulates the behavior of an origin Web site.

For the first microbenchmark, the Polygraph clients maintain a rate of 100 req/s, and a variable number of robots: 250, 1000, 2500 and 10,000. The file size is fixed at 8KBytes, which represents the ≈75-th percentile of the Polymix-4 object size distribution. Figure 4 presents the proxy CPU utilization versus the mean number of concurrent established connections as reported by Polygraph, which may be lower than the total number of robots. The plots illustrate that, for this level of request rate, with `uselect`, the system load is independent of the number of active connections, while with `/dev/epoll` and `select`, the system loads are very sensitive to the number of active connections. For instance, the load difference between 1000 and 2500 connections is 0% for `uselect`, 14% for `dev/epoll`, and 55% for `select`.

For the second microbenchmark, the Polygraph clients maintain 5,000 and 10,000 robots, each with a fixed rate of 0.01 req/s, resulting in overall request rates of 50 req/s and 100 req/s, respectively. File size varies

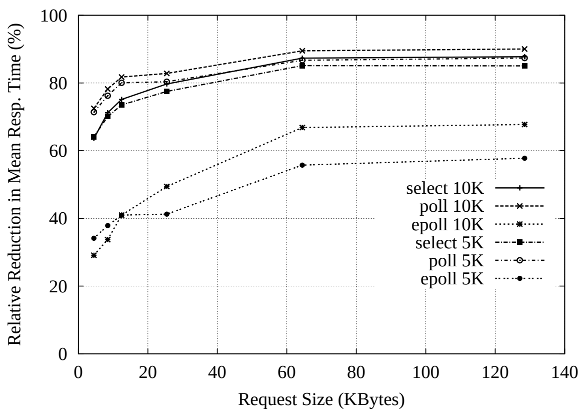

Figure 6: Rel.reduction of response time w/ uselect: 100%Hits, null store, 5K robots and 10K robots.

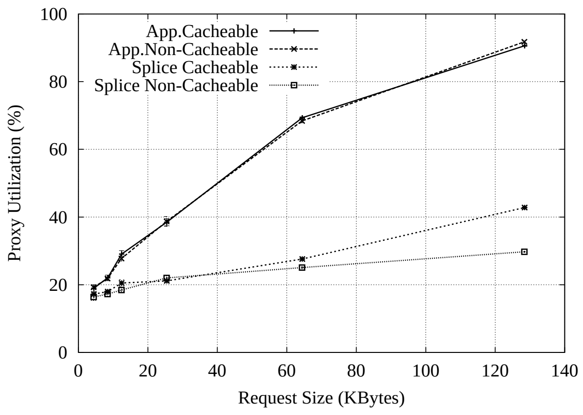

Figure 7: Proxy CPU utilization: 0%Hits, 40 req/s, null store.

from 4 to 128KBytes. We could not experiment with significantly more than 10K robots because of memory limitations on the proxy.

Figure 5 presents the variation of proxy CPU utilization. The plots illustrate that `uselect` is significantly more scalable than the other three mechanisms. For 5K active connections, the relative reduction in CPU utilization is 85-96%. For the larger load, the relative reduction is 68-94%. Moreover, the plot illustrates that `uselect` can handle 10K connections with lower overheads than `/dev/epoll` can handle 5K connections. The plot illustrates also that `/dev/epoll` is more scalable than `select` and `poll`, and that, at this loads, `select` is slightly more scalable than `poll`.

The lower CPU overheads achieved with `uselect` translate in significant response time reductions. Figure 6 presents the relative reductions computed as $100(1 - A.r/B.r)$, where $X.r$ is mean response time measured for configuration X. For 5K connections, the reduction relative to `/dev/epoll` is 29-58%, relative to `select` is 62-85%, and relative to `poll` is 70-85%.

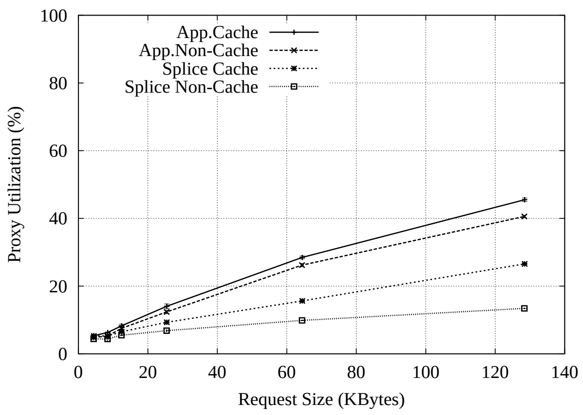

Figure 8: uselect proxy CPU utilization: 0%Hits, 40 req/s, null store.

5.2 Microbenchmark: Data-stream Splicing

Two microbenchmarks are used to evaluate the performance of data-stream splicing relative to application-level forwarding. The first microbenchmark evaluates the dependence on transfer lengths, and the second microbenchmark evaluates the dependence on system load.

For the first microbenchmark, Polygraph clients maintain a fixed rate of 40 req/s and a hit ratio of 0% (i.e., all requests handled by Squid require transfers from the server). Across experiments, we vary the object size and cacheability (i.e., cacheable or non-cacheable). When the object is cacheable, the splice mode is *KeepCopy*, thus the application performs read operations to bring the spliced content in its cache; no reads are performed for non-cacheable objects.

Figure 7 presents the proxy CPU utilization when the request rate is fixed at 40 req/s. This rate level is chosen to avoid reaching overload on both proxy and network. The plot illustrates that socket-level splice can result in significant overhead reductions. These reductions increase with the object size. Also, for large objects, reductions are larger for non-cacheable than for cacheable objects, for which *KeepCopy* is used. This is due to the fewer system calls executed for serving non-cacheable objects, difference which is relevant only for large objects. For non-cacheable objects, the relative reduction varies from 15% for 4K files, to 68% for 64K and 128K files. For cacheable objects, the relative reduction is 10-60%. For these experiments, the reduction in response time is relatively insignificant (up to 1.7%) because the mean response times is large (2.7-3.4s) due to server think time.

The performance benefit of data-stream splicing remains relevant also when using `uselect`. Figure 8 presents the CPU utilization when the application uses `uselect` instead of `select` to handle the same

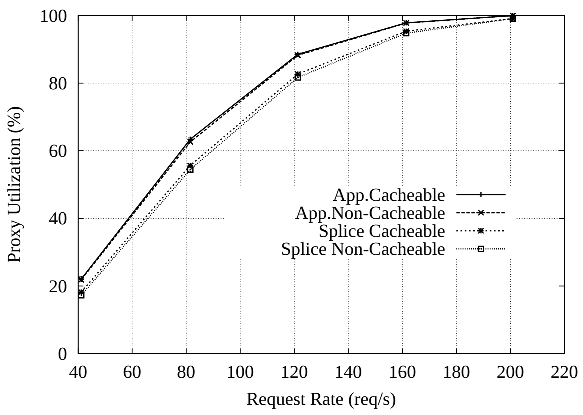

Figure 9: Proxy CPU utilization: 0% Hits, 8KByte files, null store.

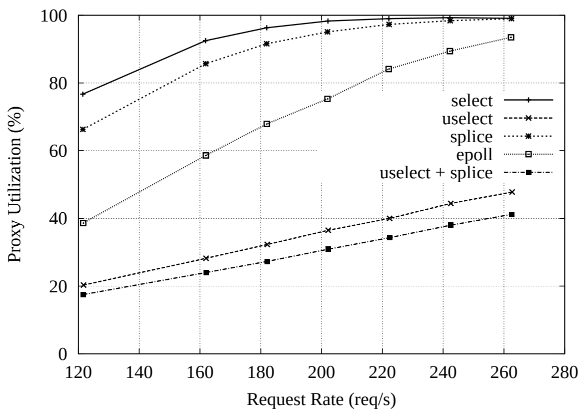

Figure 11: Proxy CPU utilization: Polymix-4, diskd store.

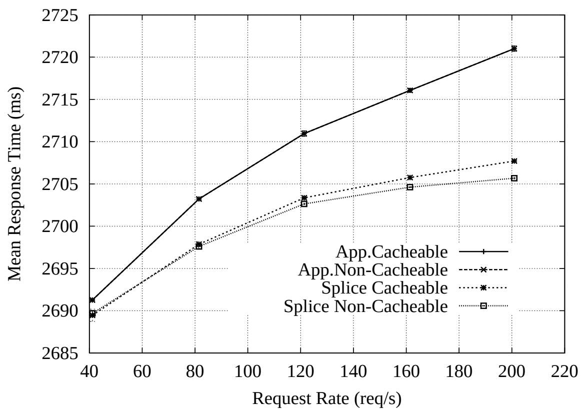

Figure 10: Mean Response Time: 0% Hits, 8KByte files, null store.

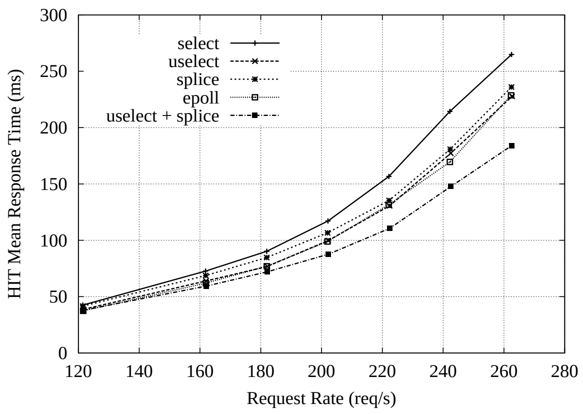

Figure 12: Hit Mean Response Time: Polymix-4, diskd store.

workload. The reduction relative to `uselect` with application-level forwarding is up to 45% for cacheable objects, and up to 65% for non-cacheable objects.

These experiments illustrate that data-stream splicing is a necessary mechanism in the context of the heavy-tail distribution of Web content sizes, because it helps Web proxies to significantly reduce the performance perturbations caused by serving atypically large files.

For the second microbenchmark, Polygraph clients generate request rates between 40 and 200 req/s and the file size is fixed at 8KBytes.

Figures 9 and 10 present the proxy CPU utilization and the mean response time, respectively; note that the plots for application-level forwarding overlap. These plots demonstrate that the reduction in CPU overhead enabled by splice increases with the request rate. However, as the system approaches overload, this reduction has a smaller impact on CPU utilization, but a larger impact on response time. For instance, for non-cacheable objects, the difference in CPU utilization decreases from 8% (at 120 req/s) to 0.9% (at 200 req/s), while the response time difference increases from 2ms to 14ms.

5.3 Polymix-4 Workload

In order to estimate the benefits of the proposed mechanisms in a real Web proxy deployment, we experiment with Polymix-4, a workload that provides a realistic mix of file sizes, cacheability types, and HTTP request types; it generates a realistic hit ratio, connection persistency model, and server think-time. This experiment evaluates `uselect`, `select`, `/dev/epoll` with application-level forwarding, and splicing with `select` and with `uselect`. Request rates vary from 120 to 260; swapping impacts the results at higher rates.

Figure 11 presents the proxy CPU utilization and Figures 12 and 13 present the mean response times for hits and misses, respectively. `uselect` provides reductions in CPU utilization similar to those in the microbenchmarks, 50-70% relative to `select` and 50% relative to `/dev/epoll`. These experiments also demonstrate that the `uselect` implementation can handle effectively

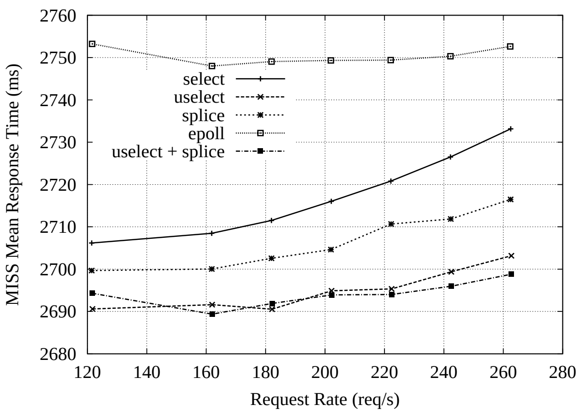

Figure 13: Miss Mean Response Time: Polymix-4, diskd store.

both connect and accept operations; the microbenchmarks exercise mostly the accept operation.

Similarly, `splice` provides reductions in CPU overheads comparable to the microbenchmarks, considering that in Polymix-4 cache misses represent only about 40% of the requests and that file size distribution is biased towards small files (the 75-th percentile is 8KByte).

The CPU overhead reductions provided by `uselect`, `splice`, and their combination translate in reductions of mean response times for both hits and misses compared to the configuration using `select` and application-level forwarding. Reductions are larger at higher request rates. More specifically, the reductions for hits with `uselect` are 4-35ms, with `splice` are 1-28ms. Using both mechanisms, the reductions are 4-80ms, a 10-30% relative improvement. The reductions for miss with `uselect` are 15-30ms, and with splice are 6-16ms. Using both mechanisms yields reductions similar to `uselect` alone, which is a 0.5-1.3% relative improvement. We note that `/dev/epoll` provides reductions for hit response times similar to `uselect`, but the miss response times are the highest among the tested configurations, 20-50ms more than with `select`.

5.4 Discussions

Overall, the experimental evaluation presented in this paper illustrates that both user-level connection tracking and data-stream splice help lower the overheads and improve the scalability of Web servers. The former mechanism benefits any Web server that handles a very large numbers of concurrent connections, while the latter mechanism is mostly limited to Web proxy caches.

One lesson that we learned while experimenting with data-stream splice is that it is detrimental to forward data packets in process context. Our earlier implementations attempted to forward as many packets as possible in the *Splice* call, and upon the return from the `read` system call, for the *KeepCopy* mode. This approach has a negative performance impact because holding socket locks in process context interferes with the highly optimized Linux TCP/IP stack: packets received while forwarding in process context incur larger processing overheads.

As expected, the benefits of `uselect` and `splice` are not cumulative. This is because part of the overhead reduction achieved with `splice` is due to a reduction in the number of `select` system calls.

From our experience of modifying Squid to use `/dev/epoll`, we learned that it is not straightforward to change a complex application designed to use select/poll-type connection state tracking to use event notification-based mechanisms, like `/dev/epoll`. It can be very difficult to identify all the code regions in the application built on programmer's assumptions about the state-tracking model. The same argument applies to a complex Web application implemented around an event notification mechanism, such as `/dev/epoll`. The connection state tracking mechanism proposed in this paper enables effective user-level implementations of mechanisms like `select/poll` and `/dev/epoll`; applications can enjoy the performance benefits with minimal modifications, just by linking to the library that implements the desired API.

6 Related Work

Recent research on Web server performance has focused on optimizing the operating system functions on the critical path of request processing. In this paper, we focus on the same problem domain, proposing two mechanisms that Web servers, and in particular Web proxy servers, can use in a flexible manner to reduce connection handling and data forwarding overheads. Both mechanisms address the overheads of context switching and data copy between application and kernel domains, one for connection state tracking and the other for data forwarding in TCP streams.

A large body of research has focused on improving the scalability of connection-state tracking mechanisms, critical for event-driven architectures. Traditional mechanisms for connection-state tracking, `select` and `poll` exhibit poor scalability. Optimizations can reduce the in-kernel overhead of collecting socket status information [4], but cannot reduce context switching and data copy overheads. Event delivery mechanisms with batch notifications represent an alternative to `select/poll` [5, 6, 19, 33, 34, 35]. This paradigm supports implementations that are more efficient, and reduces the overhead of data copy between user and kernel space, in particular when the number of active connections is a small fraction of the total number of open connections. The `/dev/epoll` proposed in [34] further reduces data copy overheads by using a shared memory region between kernel and application for passing event notifications. Similarly, *ECalls* [33] uses shared memory for application-to-kernel and kernel-to-application notification. Overall, these proposals help reduce the system call overheads, but cannot reduce the number of invocations. Event delivery mechanisms with individual notifications, like I/O completion ports (IOCP) [12], incur a larger volume of system calls, but benefit a thread-based architecture by implementing a thread dispatching policy that minimizes thread context switches.

The connection state tracking mechanism introduced in this paper enables significant reductions of the number and overheads of system call invocations. By using connection state elements propagated by the kernel in a shared memory region, the application can acquire the information necessary for connection state tracking without context switching to the kernel domain. In the Linux implementation, connection state is propagated at user space automatically, after `connect` or the first `select/poll`; no system call other than `connect` and `accept` would be required if a file descriptor were assigned to the socket prior to the invocation of the `accept` handler. Existing APIs like `select`, event delivery [5, 34],x and IOCP [12] can be re-implemented to exploit the mechanism and achieve significant overhead reductions.

Numerous studies on TCP and server performance demonstrate that achievable transfer bandwidths are limited by the overhead of copying data between kernel and user-space buffers [7, 17]. Previous research has proposed several in-kernel splicing mechanisms of data streams produced by devices/files and sockets [11, 31]. In-kernel splicing of TCP connections has been proposed, as well. Some of the solutions [3, 9, 13, 15, 16] do not make the splicing interface available at application level. These solutions are integrated with kernel-level modules for HTTP request distribution and are implemented either between the TCP and socket layers [3, 15, 16] or in the IP layer [9, 13]. Existing solutions that can be exploited at application level are implemented in the IP layer [21, 22, 39] or in the socket layer [37], and are restricted in their ability to effectively serve the full range of connection characteristics and request types handled by a Web proxy. For instance, the IP-level implementations cannot handle pipelined requests and client aborts. The mechanism in [37] cannot handle cacheable content and persistent connections.

The data-stream splice mechanism proposed in this paper enables a Web proxy application to exploit kernel-level forwarding for all types of requests that involve transfers between its server and client connections. Similar to [37], the mechanism is implemented at socket level but with extended functionality. Drawing from the socket level implementation, the mechanism has several advantages over the IP-level implementations. First, the mechanism allows the splicing of TCP connections with different maximum segment sizes or TCP options and fosters faster loss recovery [37]. Second, it allows for more efficient support for persistent connections (e.g., the mechanism in [39] unsplices at the first data received in the client connection) and for caching the transferred content (e.g., the mechanism in [22] aborts the caching procedure if the client aborts the connection). These advantages offset the relatively small increase in forwarding overheads vs. IP-level splicing due to the transport-layer processing on incoming and outgoing paths. We submit that IP-level solutions need to re-implement substantial segments of the TCP stack in order to support a flexible API, similar to the one proposed in this paper.

7 Conclusion

This paper proposes to enhance a general-purpose operating system with mechanisms that reduce the system overheads of applications such as Web servers, which handle large numbers of concurrent connections, and of applications such as Web proxies, which forward large volumes of data.

Improved scalability with the number of active connections is enabled by user-level connection tracking. Promoting a new implementation paradigm, this mechanism is the first to provide notifications of connection state changes without incurring any context switches and data copies between application and kernel domains. With a `select` API, this mechanism demonstrates CPU overhead reductions of 52-72% relative to `select` and 50% relative to `/dev/epoll`. In the future, we plan to implement an event notification API similar to [5].

Lower data forwarding overheads are enabled by data-stream splicing. Implemented in the socket layer, this mechanism is the first to enable effective in-kernel forwarding for the whole range of transfers performed by a Web proxy cache, supporting persistent connections, request pipelining, and content caching. Experiments demonstrate up to 12% reductions in Squid's forwarding overheads.

Acknowledgments. Authors thank John Tracey and Erich Nahum for providing the GigaBit equipment used in the evaluation. Also, authors thank Anees Shaikh for his comments on earlier versions of this paper.

References

[1] T. Anderson, H. Levy, B. Bershad, E. Lazowska, "The Interaction of Architecture and Operating System Design", *ASPLOS, 1991*

[2] Apache Software Foundation, "Apache http server

project", *http://www.apache.org/*

[3] H. Balakrishnan, V. Padmanabhan, S. Seshan, R. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links", *ACM SIGCOMM, 1996*

[4] G. Banga, J. Mogul, "Scalable kernel performance for Internet servers under realistic loads", *USENIX OSDI 1998*

[5] G. Banga, J. Mogul, P. Druschel, "A scalable and explicit event delivery mechanism for UNIX", *USENIX Annual Technical Conference, 1999*

[6] A. Chandra, D. Mosberger, "Scalability of Linux Event-Dispatch Mechanisms", *USENIX Annual Technical Conference, 2001*

[7] J. Chase, A. Gallatin, K. Yocum, "End-System Optimizations for High-Speed TCP", *IEEE Communications, 39(4), Apr. 2001*

[8] E. Cohen, H, Kaplan, J, Oldham, "Managing TCP connections under persistent HTTP", *World Wide Web Conference, 1999*

[9] A. Cohen, S. Rangarajan, H. Slye, "On the Performance of TCP Splicing for URL-aware Redirection", *USENIX Symposium on Internet Technologies and Systems, 1999*

[10] T.Dierks, C. Allen, "The TLS Protocol, Version 1.0", *IETF, Network Working Group, RFC 2246*

[11] K. Fall, J. Pasquale, "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability", *USENIX Winter, 1993*

[12] J. Hart, "Win32 System Programming", *Addison Wesley Longman, 1997*

[13] G. Hunt, G. Goldszmidt, R. King, R. Mukherjee, "Network Dispatcher: A Connection Router for Scalable Internet Services", *World Wide Web Conference, 1998*

[14] S. Iyer, A. Rowstron, P. Druschel, "Squirrel: A decentralized peer-to-peer web cache", *ACM Symposium on Operating Systems Principles, 2001*

[15] IBM Corporation, "IBM Netfinity Web Server Accelerator V2.0", *http://www.pc.ibm.com/ us/solutions/ netfinity/ server_accelerator.html*

[16] P. Joubert, R. King, R. Neves, M. Russinovich, J. Tracey, "High-Performance Memory-Based Web Servers: Kernel and User-Space Performance", *USENIX Annual Technical Conference, 2001*

[17] J. Kay, J. Pasquale, "Profiling and Reducing Processing Overheads in TCP/IP", *IEEE/ACM Transactions on Networking, 4(6) p.817-828, 1996*

[18] B. Krishnamurthy, C. Wills, "Improving Web Performance by Client Characterization Driven Server Adaptation", *World Wide Web Conference, 2002*

[19] J. Lemon, "Kqueue: A generic and scalable event notification facility", *FREENIX Track: USENIX Annual Technical Conference, 2001*

[20] D. Libenzi, "Improving (network) I/O performance", *http://www.xmailserver.org/linux-patches/nio-improve.html*

[21] D. Maltz, P. Bhagwat, "MSOCKS: An Architecture for Transport Layer Mobility", *INFOCOM, 1998*

[22] D. Maltz, P. Bhagwat, "Improving HTTP Caching Proxy Performance with TCP Tap", *IBM Research Report RC 21147, Mar. 1998*

[23] The Measurement Factory, "Publich Benchmarking Results", *http://www.measurement-factory.com/results*

[24] E. Nahum, T. Barzilai, D. Kandlur, "Performance Issues in WWW Servers", *ACM SIGMETRICS, 1999*

[25] E. Nahum, M. Roşu, S. Seshan, J. Almeida, "The Effects of Wide Area Conditions on WWW Server Performance", *ASM SIGMETRICS, 2001*

[26] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, R. Tewari, "Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks", *World Wide Web Conference, 2002*

[27] NLANR, "Squid Web Proxy Cache", *http://www.squid-cache.org/*

[28] NLANR, "IRCache Home", *http://www.ircache.net*

[29] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?", *Summer 1990 USENIX Conference*

[30] V. Pai. P. Druschel, W. Zwaenepoel, "Flash: An Efficient and Portable Web Server", *USENIX Annual Technical Conference, 1999*

[31] J. Pasquale, E. Anderson, K. Fall, J. Kay, "High-Peformance I/O and Networking Software in Sequoia 2000", *Digital Technical Journal, 1995*

[32] V. Paxon, "End-to-end Internet packet dynamics", *IEEE/ACM Transactions on Networking, 7(3), June 1999*

[33] C. Poellabauer, K. Schwan, R. West, "Lightweight Kernel/User Communication for Real-Time and Multimedia Applications", *Workshop on Network and Operating Systems Support for Digital Audio and Video, 2001*

[34] N. Provos, C. Lever, "Scalable network I/O in Linux", *Technical Report CITI-TR-00-4, University of Michigan, Center for Information Technology, 2000*

[35] N. Provos, C. Lever, S. Tweedie, "Analyzing the Overhead Behavior of a Simple Web Server", *Technical Report CITI-TR-00-7, University of Michigan, Center for Information Technology, 2000*

[36] L. Rizzo, "dummynet", *http://info.iet.unipi.it/ ∼luigi/ip_dummynet*

[37] M. Rosu, D. Rosu, "An Evaluation of TCP Splice Benefits in Web Proxy Servers", *World Wide Web Conference, 2002*

[38] M. Rosu, D. Rosu, "Kernel Support for Faster Web Proxies", *IBM Research Report RC 22669, Dec. 2002*

[39] O. Spatscheck, J. Hansen, J. Hartman, L. Peterson, "Optimizing TCP Forwarder Performance", *IEEE/ACM Transactions on Networking, 8(2), April 2000, also Dept. of CS, Univ. of Arizona, TR 98-01, Feb.1998*

[40] Web Polygraph, "Workloads", *http://www.web-polygraph.org*

[41] M. Welsh, D. Culler, E. Brewer, "SEDA: An Architecture for Well-Conditioned scalable Internet Services", *ACM Symposium on Operating Systems Principles, 2001*

[42] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, H. Levy, "On the scale and performance of cooperative Web proxy caching", *ACM Symposium on Operating Systems Principles, 1999*

[43] Zeus Technology Limited, "Zeus Web Server", *http://www.zeus.co.uk*