Towards Enabling Web Proxy Control of TCP Splice Transfer Rates

Jiantao Kong [*]
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
jiantao@cc.gatech.edu

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

Abstract

This paper addresses the problem of CPU resource contention between a Web proxy cache and the TCP Splice kernel service, which this application employs for serving cache misses. TCP Splice improves the performance of a proxy cache by reducing the CPU utilization and latency for cache misses. However, TCP Splice implementations based on packet forwarding in the IP or socket levels can delay the serving of cache hits when high bursts of packets move through the in-kernel infrastructure. In these implementations, the TCP Splice activity has higher priority than the application, and the application has no means of controlling the pace of spliced transfers. In this paper, we propose an alternative paradigm for TCP Splice implementation that enables application control, while providing reasonably large reductions in CPU overheads.

1 Introduction

Previous research has proved that the TCP Splice kernel service can significantly reduce transfer overheads in Internet servers like firewalls, mobile gateways, and content-based routers [5, 4, 2, 1, 7]. Web proxy cache servers can also benefit from exploiting TCP Splice, but the benefits may be limited by the resource contention between the kernel-level spliced transfers and the cache-served transfers [7].

More specifically, the TCP Splice implementations proposed by previous research can cause an increase of response times for cache hits. These implementations, based on either IP or socket-layer mechanisms, perform most of the activity in interrupts. Thus, the forwarding of packets in spliced connections receives higher priority than the application itself. This causes delays of the application-level activities, including the serving of cache hits, when high bursts of packets move through the TCP Splice infrastructure.

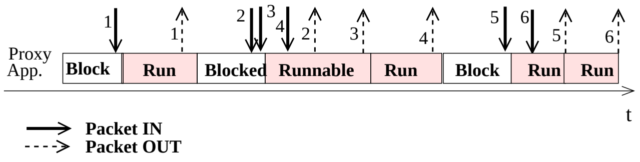

Figure 1: Timeline with application-level splicing.

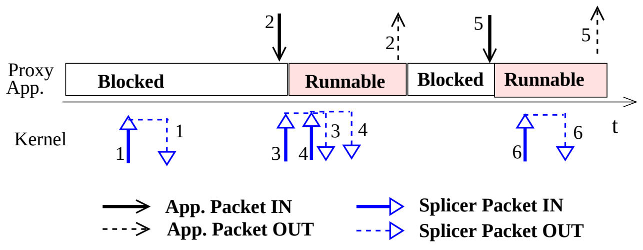

Figure 2: Timeline with interrupt-driven TCP Splice.

To illustrate this effect, Figures 1 and 2 represent the execution timeline when the application performs application-level splicing and when it exploits an interrupt-driven TCP Splice, respectively. We consider an event-driven application, such as the Squid Web proxy cache, using a connection-state tracking mechanism, such as `select`. The application alternates between the *runnable* state, in which it serves all the events signaled by the most recent invocation of this mechanism, and the *blocked* state, in which it waits for I/O events. When exploiting TCP Splice, the application is not notified for the I/O events related to

[*]Work done during Summer Internship at IBM T.J.Watson Research Center

spliced connections (see Figure 2). These are handled by the TCP Splice module as soon as they occur. Thus, packet arrivals on the spliced connections interrupt the application and delay its execution until these packets are forwarded on peer connections. The additional service delay becomes relevant under high bursts of spliced traffic or when the service time for cache hits is very small (e.g., mostly memory hits).

We submit that proxy applications, like Web proxy caches, cannot fully benefit from the TCP Splice kernel service unless they can control the pace of their spliced transfers according to their overall performance goals. Due to their interrupt-driven paradigm, the previously proposed implementations have no simple extensions that could enable application-level control.

The goal of our work is to identify a TCP Splice implementation paradigm that enables the integration of various models of application-level control while providing significant overhead reductions relative to application-level splicing. Further, our quest is to identify control methods that are effective in the context of highly variable workloads and connection characteristics (e.g., hit rates, transfer bandwidths), such as experienced by typical Web proxy caches.

In this paper, we present the *kThread TCP Splice*, an implementation paradigm in which the spliced traffic is forwarded by a kernel thread. This thread can stop forwarding when the application is runnable and can resume upon application signals. A more refined control of the forwarding activity is achieved by combining application-supplied statistics about user-level workloads with the forwarding thread's own observations of the spliced transfers.

Preliminary experimental evaluation demonstrates that the kernel thread-based TCP Splice reduces transfer overheads relative to application-level splicing, while it improves response times for cache hits relative to interrupt-driven TCP Splice. The evaluation is performed on Linux, 2.4 kernel, with Squid 2.4, as the proxy cache application.

The remainder of this paper is organized as follows. Section 2 presents the design of the kThread TCP Splice. Section 3 presents experimental results that illustrate the benefits of application-level pacing of TCP Splice transfers, as enabled by the kThread implementation. Section 4 discusses our plans for improving and extending the proposed TCP Splice control mechanism. Section 5 briefly presents the related work.

2 kThread TCP Splice

The kernel-thread based implementation of TCP Splice proposed in this paper, *kThread TCP Splice*, is derived from an interrupt-driven, socket-level implementation in Linux (2.4 kernel) henceforth called *Interrupt TCP Splice*. To facilitate the understanding of our design choices for kThread TCP Splice, we start with a brief presentation of the Interrupt TCP Splice.

Interrupt TCP Splice. This implementation is structured as a device driver module. The module maintains the description and state for each pair of spliced connections.

The application accesses the service through two system calls (`ioctl`'s), *splice* and *unsplice*. *Splice* initializes the splicing of a connection pair, indicating the type of transfer (e.g., bi- or unidirectional, with or without local copy, terminated by transfer size or connection close). The application is notified about the completion of a spliced transfer with a `POLLPRI` event. *Unsplice* separates two spliced connections.

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.

Most of the packet forwarding is performed by the `data_ready` and `write_space` handlers, which are called when packets are received. Packet forwarding is also performed in the *splice* system call to deplete the receive queue; for small files, splicing might complete at this time.

kThread TCP Splice. The goal of the kThread TCP Splice is twofold. First, the implementation enables the application, during its runnable intervals, to indicate to the TCP Splice infrastructure when to forward packets. Second, the implementation enables the TCP Splice infrastructure to forward packets while the application is blocked; the forwarding should stop, or be limited by some application-specified parameters, as soon as the application becomes runnable.

Towards this end, kThread TCP Splice extends the infrastructure of Interrupt TCP Splice with a kernel thread and a *forward* system call (ioctl). Different

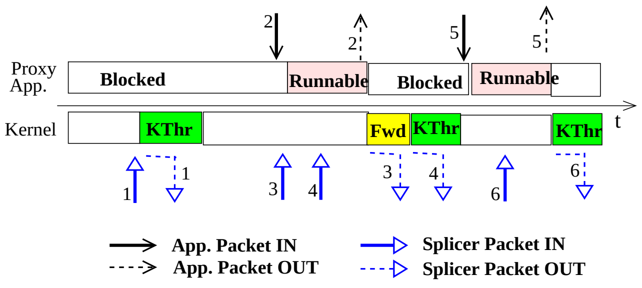

Figure 3: Timeline with KThread TCP Splice.

from Interrupt TCP Splice, the forwarding is performed by the kernel thread, when the application is blocked, and by the application thread itself in the *forward* call, when the application is active. Figure 3 illustrates the execution timeline when the application exploits kThread TCP Splice, for the same sequence of I/O events as in Figure 2.

In this implementation, the `data_ready` and `write_space` handlers do not forward packets. They wake-up the kernel thread when the application is blocked.

By using the *forward* call, the application ensures that the receive queues of the spliced connections are drained in a timely fashion even when it stays runnable for longer intervals, such as during periods with a high volume of memory cache hits. The application may choose to forward all of the existing packets, or only those received before the current active interval. The latter approach, illustrated in Figure 3, is more conservative mimicking the forwarding pattern of application-level splicing.

The kernel thread is activated by the socket handlers and by the `forward` command when there are connections with non-empty `receive_queue`'s. The thread will traverse the list of active connections, forwarding the packets in their `receive_queue`.

The kernel thread checks regularly the state of the application (e.g., after each handled connection). When the application is found to be runnable, the thread checks the *yield condition*. The kernel thread blocks when the yield condition holds true. The same condition is used for the wake-ups issued from the `data_ready`/`write_space` handlers. The yield condition can vary with the implementation and application requirements. For instance, in a conservative approach, the condition is always true, stopping the thread from forwarding when the application is active. Another approach, called *proportional rate*, is to maintain a proportion between the amounts of data forwarded from the kernel and from the application-level that is equal to the ratio of the corresponding number of served requests. Towards this end, application and kernel collect statistics. Application statistics are conveyed to the kernel module as a parameter of the *forward* call. At fixed sampling interval (e.g., 2 sec), the kernel module uses these statistics to adjust the bounds in the yield condition.

To summarize, the kThread TCP Splice eliminates the context switches and data copy incurred by application-level transfers, and provides a framework in which the application can control the pace of splice transfers relative to the other activities of the application. The experimental results presented in the next section demonstrate that overheads of the kThread implementation are very close to those of the interrupt-based implementation.

3 Experimental Evaluation

The experimental results presented in this section address the performance of kThread TCP Splice with conservative and proportional-rate controls relative to application-level splicing and Interrupt TCP Splice.

The experimental testbed comprises four nodes, a client (660MHz, 256M), a proxy (200MHz, 128M), a server (1.8GHz, 512M), and a router (200MHz, 128M). The client application is generating best-effort traffic in multiple concurrent request streams. The object size is fixed for an experiment. Object cacheability and expected hit ratio are defined by experiment parameters. The number of requests in a connection is Zipf($\alpha = 1, N = 64$). The proxy application is Squid 2.4 modified to use TCP Splice for requests requiring transfers from the server. The Squid cache is memory-based. The server is an HTTP-server emulator, using client-specific HTTP headers to determine the object size and its cacheability. The number of replies in a connection is Zipf($\alpha = 1, N = 16$). The router emulates network delays using NISTNet.

For the experiments presented in this section, the number of client streams is 128. Hit ratio is 50%. All hits are for a unique object and the misses are for non-cacheable objects. Object sizes are 5, 10, 15, 20, 33, 64, 100, and 128 KBytes. The emulated network delay between proxy and server is 40 ms; there is no delay between client and proxy, and no loss on either link. Each data point is the average of three samples; the confidence level is 95%.

In the following plots, the label ‘App.’ corresponds to application-level splice, ’Interrupt’ corresponds to Interrupt TCP Splice, ‘KThread Conservative’ corresponds to KThread TCP Splice with conservative control, and ’KThread Proportional Rate’ corresponds to KThread TCP Splice with proportional rate control.

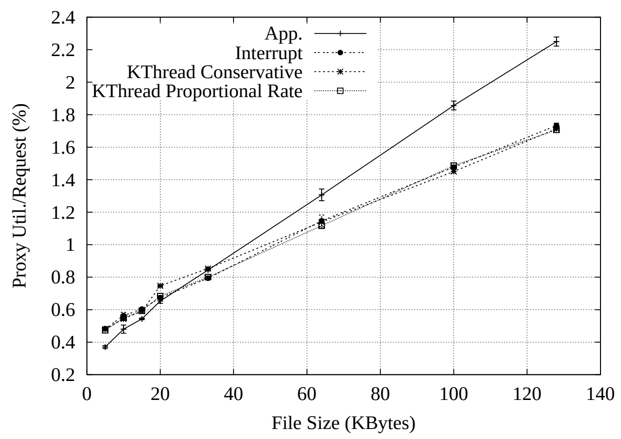

Figure 4: Proxy utilization per request.

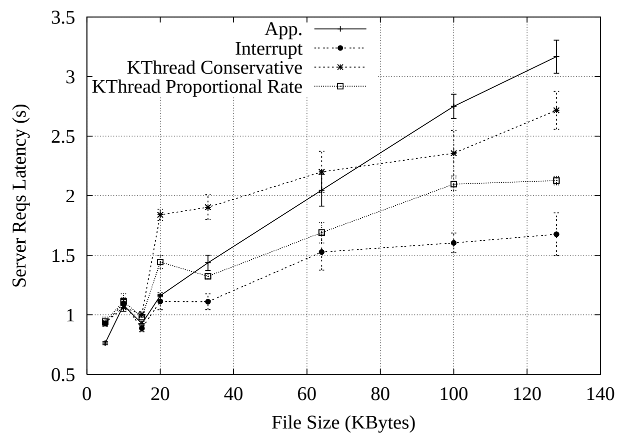

Figure 5: Average latency for spliced requests.

Figure 4 presents the proxy utilization per-request for the four types of splicing. This is computed as the ratio of average CPU utilization by total request rate. The plots demonstrate that kernel-level splicing reduces the proxy utilization per request, in particular for large file size. (Recently, we fixed Interrupt TCP Splice such that, for small files, CPU utilization per request and latency are lower than for application-level splicing; we are working on integrating these changes in kThread TCP Splice.) In addition, the kThread implementation with proportional rate control has quasi the same overheads as the interrupt-based implementation.

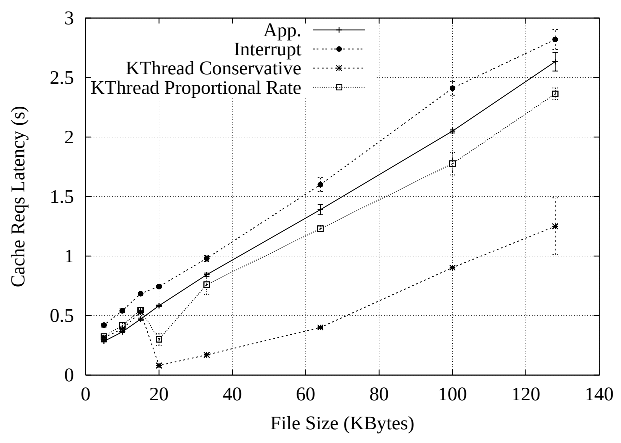

Figure 6: Average latency for cache requests.

Figures 5 and 6 present the average response latency for requests served from the server, by splicing, and from the local cache, respectively. The plots illustrate that, in comparison to application-level splicing, the interrupt-based implementation enables significant reductions of the response latency for spliced transfers. kThread TCP Splice enables the application to control the impact of spliced transfers on the response latency of cache hits. With conservative control, the service of cache hits has a higher priority than spliced requests. Therefore, the latency of cache hits is significantly lower (50-80%) than for application-level splicing, while the latency for spliced transfers increases (5-30%) for medium sized files. The proportional-rate control results in a more balanced handling of the two types of requests. Thus, the latency for both cache hits and spliced transfers is better than with application-level splice. The sudden drop in the latency of cache hits is related to the size of socket send buffers and influences packet forwarding and application scheduling.

4 Future Work

The TCP Splice control mechanisms considered in this paper are very simple. However, the framework offered by the forwarding kernel thread enables us to define more elaborated controls, such as rate enforcing.

In future work, we plan to identify better mechanisms for balancing resources between user- and kernel-level activities of applications like Web Proxy servers. We plan to consider solutions that take into account TCP connection parameters, such as window sizes and RTTs.

The experimental results illustrate that our current implementations of TCP Splice are not very efficient for small file transfers. We have identified the cause of inefficiency and we are currently experimenting with alternative implementations.

For future experimental evaluations, we plan to use typical proxy workloads, like those considered for the Polygraph Cache-offs. The best effort, fixed file-size workloads used in the preliminary experimental evaluation allowed us to observe the behavior of the TCP Splice implementations in a more predictable setting.

5 Related Work

An early in-kernel splicing mechanism was proposed for splicing data streams produced by devices/files and sockets [3]. To control the related impact on other activities in the system, this mechanism bounds the number of outstanding device/file I/O operations to a system-specific constant. The target of our work is to employ dynamic bounds, which change with the application workload.

Previous research has proposed several solutions for in-kernel splicing of TCP connections. All these solutions have interrupt-driven implementations, either in the IP layer [5, 6, 2, 4] or in the socket layer [1, 7]. None of these solutions has considered the problem of enabling application-level control over the pace of spliced transfers. For some of these solutions, this is not a concern because they are tailored for highly specialized systems, like HTTP (content-based) routers and mobile gateways [2, 4, 1].

6 Conclusions

This paper addresses the problem of CPU resource contention between a Web proxy cache application and the TCP Splice kernel service, which this application employs for serving cache misses. This paper proposes a new implementation paradigm for the TCP Splice kernel service that enables the application to manage this contention, by controlling the pace of the spliced transfers. Experimental evaluation demonstrates that by using simple control mechanisms, the proposed implementation improves on the response times of cache hits with respect to both application-level splicing and interrupt-driven socket-level splicing.

References

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

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

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

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

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

[6] 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*

[7] M. Rosu, D. Rosu, "An Evaluation of TCP Splice Benefits in Web Proxy Servers", *WWW11, 2002*