A Fast Prolac TCP for the Real World

by

David Rogers Montgomery, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by . . . .
M. Frans Kaashoek
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Fast Prolac TCP for the Real World

by

David Rogers Montgomery, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I verify that a TCP specification written in Prolac, a language designed for implementing network protocols, offered performance comparable to Linux 2.0.36. The thesis consists of three major sections: first, I describe how an existing Prolac TCP specification was interfaced with the Linux 2.0.36 kernel. Next I discuss measurement and testing procedure. Last, end-to-end times and excution profile data are analyzed to explain performance differences between Prolac and Linux.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

Most sincere thanks to Katie, Eddie, Frans, Jeremy and my parents.

Contents

List of Tables

Chapter 1

Introduction

Network protocols are difficult to implement. Extensions and performance concerns often cause modules to bleed across interfaces. The result is typically a monolithic piece of software that is not easily understood or modified. This limits innovation and experimentation.

Prolac (*pro*tocol *la*nguage for *c*ompilation) is a specialized language for implementing real network protocols. The language aims for increased simplicity and high performance. To achieve simplicity, it offers rules, modules and inheritance. These allow the programmer to cleanly structure the code in a top-down fashion; by abbreviating common code, essential behavior is highlighted. To achieve high performance, Prolac allows the programmer to specify levels of optimization at fine granularity. Prolac compiles to C code, which can be run through an optimizing compiler for additional gains.

Prior to this thesis, Prolac was used to write a proof-of-concept implementation of the TCP network protocol. This prototype runs as an application in user space; it communicates with itself using UNIX pipes, or with other TCP implementations using Berkeley Packet Filters and a network. The prototype Prolac TCP consists of twenty files totaling approximately three thousand non-empty lines of code. By comparison, the TCP code from the 4.4BSD kernel requires six files totaling almost five thousand non-empty lines of code. This comparison demonstrates Prolac's ability to decompose a network implementation into smaller, more understandable units.

Unfortunately, the prototype does not validate Prolac's claims to high performance. It runs two to three times slower than the in-kernel TCP implementation in 4.4BSD. Some of the slowdown can be attributed to the fact that the prototype runs in user mode. A program in user mode must compete for CPU time with other processes; usually, kernel code is only preempted to service low level interrupts. The user level TCP is also separated from the network hardware by the kernel boundary; because kernel buffers cannot be accessed from user-level without copying their contents, this imposes another performance penalty. To eliminate these penalties, we elected to run the Prolac TCP within the kernel.

This thesis presents the implementation, measurement and analysis of a kernel-level TCP protocol in Prolac. Once the Prolac TCP was in place, we measured its performance using simple throughput and latency tests. The Linux TCP code was measured with the same tests to establish a baseline for evaluating Prolac's performance. The results show that Prolac TCP provides end-to-end performance comparable to Linux TCP. Further analysis pinpoints the inefficiencies in the Prolac implementation and provides insight into design compromises in the Linux kernel.

This project also tested Prolac as a language. We discovered several compiler bugs, and added a new language feature, exceptions.

The remainder of the thesis explains the kernel implementation, measurement and analysis in detail. Chapter 2 explains how the prototype was adapted to the Linux kernel. Chapter 3 describes the measurement infrastructure and the test applications. We present our results with analysis in Chapter 4 and offer conclusions in Chapter 5.

Chapter 2

Implementation

The Prolac TCP stack was implemented as a Linux kernel module. To interface the Prolac TCP specification with Linux, we modified the specification to use the Linux socket buffer structure. The Prolac TCP exposes two different interfaces: one to the underlying IP layer and one to the application layer. The IP layer delivers packets from the network device to the TCP protocol, while application programs read and write buffers of data, as well as controlling connection status.

Linux provides a facility to install code into a running kernel; the code is called a loadable kernel module. Once installed, a kernel module has the same benefits and protections as the kernel, but can be uninstalled at any time. This flexibility is useful, especially during debugging; testing new code requires only unloading the current module and installing the new version. The debug-recompile-test cycle using loadable kernel modules is much quicker than the traditional kernel debugging method, which requires recompiling the entire kernel and rebooting the machine. For this reason, we elected to implement the Prolac TCP as a loadable kernel module.

2.1 Interfaces

Prolac TCP interacts with the IP layer below it and user processes above. We modified the prototype Prolac TCP specification to use the Linux socket buffer structure. This structure represents a network packet in all levels of Linux network code, regardless of protocol. Because network protocols are frequently stacked, a packet may visit multiple protocols for input or output processing. A shared data type reduces the number of times that data must be copied, an important performance consideration.

Socket buffers are used for input and output in the TCP protocol. When sending a packet, Prolac TCP creates a socket buffer from user data, fills it with user data and a TCP header, then passes it to the IP layer for transmission. When a packet is received over the network, the device driver places the contents of the packet in an socket buffer, then sends it up to the IP layer. The IP layer then dispatches it to the appropriate protocol.

2.1.1 The IP layer

With the Prolac module is loaded, two TCP stacks are resident in the kernel. The Prolac stack, however, registers with IP under a different protocol number. A packet switch was implemented in the IP layer to direct packets to the appropriate protocol; the switch applies several simple criteria to determine whether the packet should be sent to the Prolac TCP. If the packets matches the criteria, the switch modifies the protocol field in the IP header to match Prolac's protocol number. (The Prolac receive function corrects the protocol number before beginning packet processing.)

Prolac TCP must respond to packets arriving asynchronously over a network interface. If a kernel module needs to be called from inside the kernel, it must register itself after loading. Usually, code must be added to the kernel to support implement a registration mechanism. Fortunately, the Linux INET module makes provisions for just this situation; the Prolac TCP stack is able to use the existing function inet_add_protocol to register itself as a network protocol.

2.1.2 The application layer

We originally planned to extend the Prolac prototype to support the Linux socket structure as well. This would allow Prolac TCP to be used with unmodified user applications.

The Linux socket code, however, is not well modularized. Linux sockets combine the functions of the BSD Transmission Control Block and BSD socket. Some of the internal TCP state is exposed to the Linux socket code and used for decision making. Given such a cluttered interface, we opted to use system calls to provide rudimentary socket operations until we could further analyze Linux sockets.

The Linux kernel has no support for dynamically creating system calls; they must be defined when the kernel is compiled. To implement the Prolac socket operations, we used an extra level of indirection. We modified the kernel to implement the desired system calls. Each system call has an associated function pointer, which is initially null. When the Prolac TCP module is loaded, it updates these pointers with the address of internal functions. Subsequent system calls are then handed off to the kernel module for processing. When the Prolac module unloads, it disables the system calls by setting these pointers back to null.

These system calls provide a rudimentary interface similar to the BSD-socket API. A user can connect to network port and address, write and read data, and close the connection. Prolac sockets, however, do not allow blocking I/O, so an additional three system calls provide a polling interface. A user process waiting for a socket event calls the associated polling function repeatedly until the event occurs. The polling interface greatly simplifies the Prolac TCP protocol, but is unrealistic; a future version of Prolac TCP should incorporate blocking I/O.

2.2 Kernel modifications

Once a kernel module is installed, it can call any function and modify any variable exported by the kernel. The module can drastically transform the behavior of the kernel if certain control variables are exported. For example, we wanted to measure the performance of both Prolac and Linux TCPs under the stress of an unreliable connection. To simulate a bad connection, we added code to randomly drop incoming or outgoing packets in the IP layer. The rate of packet loss in each direction is controlled by an exported variable, which the Prolac module sets when loading and clears when unloading. While there are other ways to implement tunable kernel parameters, this way proved to be the easiest.

We marked the interfaces of the TCP layer with calls to start or stop a timer. The Prolac kernel module hooks these calls to record data about the processing that occurs in the TCP layer. The Prolac module implements a data log, which stores ordered tuples of pairs. To download the data, a separate application connects to the module using a system call.

2.3 Language modifications

Making the transition to kernel level required upgrading Prolac's support for exceptions. The user-level Prolac TCP uses the C setjmp/longjmp facility to simulate exceptions. Setjmp and longjmp provide a sort of bookmarking ability. A call to setjmp records the stack frame and program counter of a program; a subsequent longjmp restores the stack frame and program counter, effectively unrolling the stack up to the setjmp call. (Global state changes are not reversed.)

Setjmp and longjmp are sufficient for user-level because there are relatively few exceptions that must be handled. Running in the kernel demands a higher level of rigor about errors, introducing more exception conditions. To accommodate this, we recommended adding exception handling to Prolac. When this was implemented, we were able to significantly improve the TCP specification by using them, a clear demonstration of the value of using a language under real-world conditions.

Chapter 3

Measurement

We used three standard UNIX services (chargen, discard and echo) to benchmark the Linux and Prolac TCP implementations. Each service tests a different aspect of the TCP protocol; together, the services fully exercise a TCP stack. The tests provide a useful end-to-end measurement of Linux and Prolac TCP performance, evaluating it from a user's perspective. In conjunction with performance counters, the tests also provide more subtle measurements of protocol efficiency.

3.1 The benchmarks

3.1.1 Chargen

The chargen service sends an infinite stream of character data to the client. The speed at which the client TCP can deliver this data to the user determines the protocol's read bandwidth. The test client for the chargen service reads data from the connection in fixed size amounts. The size of these reads affects the read bandwidth measured for the TCP stack. With larger read sizes, the data transfer time begins to dominate protocol overhead, which is constant across read size.

With a reliable network, performance should be high, as packets are simply appended to the socket queue by TCP and removed from the head of the queue by the user. When we introduce network lossage, however, chargen begins to test the protocol's reassembly behavior.

3.1.2 Discard

The discard service acts as infinite sink for data–it simply discards all packets sent to it. The speed at which the protocol can send user data to the discard service determines the protocol's write bandwidth. The discard client writes data in fixed amounts to the service. Again, the size of these writes determines the ratio of protocol overhead to copy overhead.

The discard service strenuously exercises the TCP protocol. Because TCP is sending data, it must maintain a queue of packets for retransmission. Every incoming acknowledgement could potentially remove some of these packets. The TCP stack must also set timeouts to detect and respond to lost packets. When network unreliability is introduced, these timeouts become essential for maintaining performance.

3.1.3 Echo

While both chargen and discard measure bandwidth, the echo service can be used to measure latency. Echo reads data from a connection and immediately returns it to the sender, "echoing" the stream of data. We use the echo server to test latency by bouncing a packet containing a single integer between client and server; each time the client receives the packet, it increments the contents and send it to the server. Because we are transferring only four bytes of data, this test measures how quickly the TCP implementation can generate and respond to packets.

3.2 Experiments

The client applications interact with the server through the measured TCP. For Linux TCP, the clients use the standard socket interface, while for Prolac, the clients use the special system calls.

3.2.1 Parameters

The chargen, discard and echo services were located on 200 MHz Pentium Pro running OpenBSD with low load. The test protocol was installed on an IBM ThinkPad 560 with a 133 MHz Pentium processor running Linux 2.0.36. Packets were transmitted over a lightly utilized 10 Mbit/s Ethernet.

3.2.2 End-to-End measurement

End-to-end performance is measured using the UNIX gettimeofday call, which provides microsecond resolution. We start the timer after the connection is established but before data is transferred. Connection establishment is excluded from in end-to-end measurement because initiating a connection requires many operations outside the TCP protocol's control, like ARPing and route selection.

For a similar reason, we exclude the time to close a connection for both echo and chargen clients. The discard client, however, requires including connection closing to ensure accurate measurements. Because TCP can buffer large amounts of data, the last write may return well ahead of the last byte being transmitted. We wish to measure the time required to actually transfer all the data, so we wait for the connection to close, indicating all data has been received at the server. Echo and chargen are complete after the client has read a certain number of bytes, so we do not wait for connection close.

3.2.3 Execution measurement

We measure protocol performance by instrumenting calls to the TCP input/output routines with Pentium performance counters. These record internal processor events, like instruction reads or data cache misses. The performance counters allowed us to measure the efficiency of the TCP protocol at the code level. For example, a large number of instruction reads combined with a small number of code cache misses indicates that the code spends much of its time in tight loops.

To clarify the comparison between Prolac and Linux TCPs, we also recorded the impact of other kernel level operations. Because Linux TCP uses finer grain locking, it performs more lock/unlock operations and receives more interrupts than the Prolac TCP. We established the impact of locking by measuring both the number of cycles required to disable interrupts and the number of times Linux and Prolac performed the operation. To determine whether Prolac benefits from coarse locking, we ran Linux TCP with the same level of locking for comparison.

Chapter 4

Performance Analysis

Using the tests described in the previous section, we generated end-to-end time measurements as well as processor event counts. These measurements show that Prolac TCP performs comparably to Linux TCP in most situations. When the numbers differ, we explain the difference using details of the implementation.

4.1 End-to-end results

For end-to-end data throughput tests, we transferred 1,024 Kbytes in 1 Kbyte chunks. The end-to-end latency tests sent and received a four byte packet 1000 times. The results are summarized in Table 4.1. Compared to Linux, Prolac's throughput was worse, while latency was better.

Prolac's read and write throughputs are about 90 Kbytes/s slower than Linux. Because the throughput tests use larger packets than the latency test, data transfer inside the protocol consumes a larger fraction of the time. Prolac's lower read throughput is caused by an extra data copy introduced in the system call interface. Write throughput is reduced because Prolac's output routine is less efficient than Linux.

	Latency (μs)	Write throughput (KB/s)	Read throughput (KB/s)
Linux TCP	810	939	1,074
Prolac TCP	677	849	987

Table 4.1: End-to-end performance results

	Data accesses	Data cache misses
Linux TCP	1406	140
Prolac TCP	5534	899

Table 4.2: Discard test results, input

Prolac's output routine assembles outgoing packets from buffered user data. User data is copied twice: once when it is placed on the send queue, and again when Prolac assembles the outbound packet. Prolac then makes another pass over the outgoing data to compute the checksum. Each byte is read at least three times; additional retransmissions cause further copies.

Linux, by comparison, copies data from the user only once, placing it in a socket buffer and computing the checksum simultaneously. When retransmitting, Linux sends packets directly from the send queue, only updating the ack and window fields of the TCP header. By generating packets this way, Linux reads each byte only once.

The impact of multiple data copies is confirmed by the processor event counts shown in Table 4.2. During the discard test, Prolac input processing has four times more accesses to data memory than Linux, and six times as many data-cache misses. These numbers are counted under the input phase because Prolac generally sends a packet in response to an incoming ack.

In the echo test, we see Prolac TCP achieves lower latency than Linux TCP. As noted before, because the amount of data being transferred is so small, the test concentrates on protocol overhead and network latency. Since both tests were run within seconds of each other, the network latency may be assumed constant; we therefore conclude that Prolac's protocol overhead is less than Linux.

4.2 Execution analysis

Analysis of the processor event measurements shows where Prolac saves time compared to Linux; see Table 4.3 for a summary. The data show that Prolac outperforms Linux during input; the extra copies make its output function slightly less efficient and appear as higher data-read/d-cache-miss entries. In most other aspects, the code is comparable.

	Cycles		
	Input (ack)	Input (data)	Output
Linux TCP	8408	4472	15994
Prolac TCP	6255	3100	17819
	Code reads		
	Input (ack)	Input (data)	Output
Linux TCP	367	301	531
Prolac TCP	372	271	692
	Instruction cache misses		
	Input (ack)	Input (data)	Output
Linux TCP	170	90	232
Prolac TCP	141	71	272
	Data reads		
	Input (ack)	Input (data)	Output
Linux TCP	651	464	1137
Prolac TCP	614	524	1405
	Data cache misses		
	Input (ack)	Input (data)	Output
Linux TCP	19	2	53
Prolac TCP	27	2	103

Table 4.3: Echo test results

4.3 Analysis

Prolac's performance can be attributed to a combination of better code and/or less computation. Clearly, proving that Prolac generates better code is difficult without a side-by-side binary comparison. Aside from better code, Prolac could benefit from locking less or maintaining less socket state than Linux.

4.3.1 Locking

Prolac protects large sections of code from concurrency by disabling interrupts, while Linux uses small, atomic actions on the queues shared by the upper and lower halves of networking code. (The upper half services applications, while the lower half handles incoming packets.) We'd expect Linux to perform more locks and unlocks, and receive more interrupts during a given action. Our kernel measurements show that Linux disables interrupts seven times per round trip; Prolac disables interrupts only three times on average. Disabling interrupts requires 18 cycles, so this cannot account for the average difference of 1700 cycles between the Linux and Prolac.

	Time(s)	Cycles	I-cache misses
Linux TCP	.37	28669	492
Linux TCP (no interrupts)	.39	29418	524

Table 4.4: Comparing coarse and fine locking during echo test

Next, we consider whether disabling interrupts throughout input and output functions could improve Prolac's efficiency. We tested this hypothesis by disabling interrupts during both tcp_rcv and tcp_sendmsg functions. If interrupts were affecting the behavior of Linux TCP, it would be from loading code to service them. Also, cycles spent servicing interrupts would be charged to Linux TCP.

By suspending interrupts, we expect see increased cash hit rates and decreased cycle counts. The data we collected show no substantial improvement in cycle counts or i-cache misses for Linux with interrupts disabled. (see Table reftab:atomic-latency.) In fact, the latency test ran slightly slower, probably due to network driver interrupts being suppressed.

4.3.2 Sockets

While Prolac does not fully implement the Linux socket interface, it maintains very similar state. The only aspect neglected by Prolac is allowing user processes to block. When the Linux TCP receives new data, it wakes any processes sleeping on that socket. This could easily account for the 30 code-read difference when receiving data.

The difference in ack processing is harder to explain. Prolac actually performs more code reads than Linux, but requires far fewer cycles. The most apparent difference between Linux and Prolac ack processing involves retransmission timers. When TCP is sending packets, it sets a retransmission timer to detect and respond to packet loss. If all outstanding packets are acked, the timer is cleared.

Linux maintains a separate timer for each socket; when packets are in flight, the timer is set to expire shortly after TCP expects to receive an ack from the other end. When the ack arrives, all packets are accounted for and Linux clears the timer.

In OpenBSD, and therefore Prolac, a single timer is shared among all sockets. The timer expires every half second, and sockets that are waiting to retransmit decrement a wait count. If an ack arrives, the socket simply clears a flag, indicating it is no longer retransmitting. Because the timer is shared among all sockets, it is never cleared.

The approach taken by Linux TCP has many advantages. Each socket gets its own timer with one millisecond resolution, instead of 500 ms. Furthermore, if no sockets are retransmitting, all timers are inactive; Prolac wastes cycles by constantly polling sockets for retransmits. Unfortunately for Linux, the latency test is the worst case for individual timers. Timers are constantly being added when a packet is sent, then deleted before the next user write.

4.4 Summary

The data captured during our experiments proved useful during analysis. The inefficiency of Prolac's packet sending routines during the discard test argues compellingly for queueing actual packets for retransmission like Linux. The code level measurements also led us to the culprit for Linux's higher latency: the creation and deletion of socket timers.

Overall, the experimental results suggest that Prolac TCP can provide comparable performance to the Linux implementation. The code level measurements also indicate areas where optimization would reap the most gains.

Chapter 5

Conclusion

From the end-to-end and code level data in the previous chapter, it is clear that the syntactic niceties of Prolac do not results in a large performance penalty.

Our own experience using Prolac confirmed its claims about ease of use. Several extensions to the TCP prototype were implemented in very little time because of Prolac's decentralized, modular structure. At the same time, that decentralization can be daunting; because Prolac encourages computation to be expressed in many small rules, the number of interactions between modules can be overwhelming. Still, this initial discomfort is soon overcome.

The obvious next step would be to fully implement Linux TCP functionality using Prolac. The result would be a fully featured TCP implementation that maintains Prolac's comprehensibility and extensibility. This TCP specification could then be released to the public, allowing Prolac to foster innovation in protocol research.

Prolac offers many advantages over C code for specifying network protocols. With the measurement framework and techniques described in Chapter 3 in place, the Prolac compiler could be extended with additional optimizations to further improve performance.