Analysis of Techniques to Improve Protocol Processing Latency

David Mosberger Larry L. Peterson Patrick G. Bridges Sean O'Malley *

Department of Computer Science
The University of Arizona
Tucson, AZ 85716
{davidm,llp,bridges,sean}@cs.arizona.edu

Abstract

This paper describes several techniques designed to improve protocol latency, and reports on their effectiveness when measured on a modern RISC machine employing the DEC Alpha processor. We found that the memory system—which has long been known to dominate network throughput—is also a key factor in protocol latency. As a result, improving instruction cache effectiveness can greatly reduce protocol processing overheads. An important metric in this context is the *memory cycles per instructions* (mCPI), which is the average number of cycles that an instruction stalls waiting for a memory access to complete. The techniques presented in this paper reduce the mCPI by a factor of 1.35 to 5.8. In analyzing the effectiveness of the techniques, we also present a detailed study of the protocol processing behavior of two protocol stacks—TCP/IP and RPC—on a modern RISC processor.

1 Introduction

Communication latency is often just as important as throughput in distributed systems, and for this reason, researchers have analyzed the latency characteristics of common networking protocols, such as TCP/IP [15, 6, 14] and RPC [32]. This paper revisits the issue of protocol latency. Our goal is not to optimize a particular protocol stack, but rather, to understand the fundamental limitations on processing overhead.

*O'Malley's current address is Network Appliance, 319 N. Bernardo Ave., Mountain View, CA 94043. This work supported in part by DARPA Contract DABT63-91-C-0030, NFS grant NCR-9204393, and by Digital Equipment Corporation.

In doing so, this paper goes beyond the earlier work in three important ways:

- **Updated Study:** It studies protocol latency on a modern RISC architecture—the 64-bit DEC Alpha—and in doing so, updates earlier studies that were performed on the x86 architecture. This is important because the different design tradeoffs applied to CISC and RISC designs typically lead to qualitative differences in processing behavior.
- **New Techniques:** It describes and evaluates a new set of techniques that are designed to improve protocol latency. These techniques are targeted not so much at reducing the number of *instructions* executed to process each packet, but more at the number of *cycles* that each instruction takes.
- **Detailed Analysis:** It contains a level of detail not found in other studies. In particular, it reports on instruction-cache (i-cache) effectiveness, as well as on processor stall rates. The bottom-line is that we evaluate protocol latency in terms of *memory cycles per instruction* (mCPI), a metric that will become increasingly important as improvements in memory speed lag farther behind improvements in processor speed [5, 29].

It should be clear from these three points that memory bandwidth—and in particular, the memory cycles required by each instruction—is a central focus of this paper. In fact, the experimental results presented in this paper show that the difference between the worst- and best-case mCPI that we were able to measure is a factor of 3.9 for the TCP/IP stack, and a factor of 5.8 for an RPC stack. The techniques we propose are primarily targeted at improving the mCPI, although some also have a positive effect on the instruction count.

Because these techniques are aimed at improving the mCPI of networking software, they are necessarily fine-grain. To be more precise, they can all be characterized as compiler-based techniques. As such, one might ask if they are specific to networking code, or if they are applicable to general applications (e.g., SPECmark code). The answer is that while it is likely that these techniques are of some benefit to application programs, they are motivated by the unique

characteristics of networking software (specifically) and low-level systems code (more generally). For example, exception handling and other infrequently executed code often makes up a large portion of the critical execution paths in networking software. One of our techniques (*outlining*) exploits this fact. Also, execution in layered networking software often results in deep call chains and since each function call is typically an optimization barrier, in limited context available to the compiler's optimizer. A technique called (*path-inlining*) attacks these two problems. As a final example, networking software is designed to handle a wide range of situations, but once a connection is established, it is often possible to specialize the code for that particular connection. A technique called (*cloning*) addresses this issue.

Note that this works focuses on networking code as currently deployed, that is, for code written in C. We do not propose a new programming language or paradigm for protocol implementation, although we observe that some of the proposed techniques have also proven useful in alternative protocol implementation languages [4].

The paper is organized as follows. Section 2 sets the context in which this research was performed. In doing so, it expands earlier studies on TCP/IP latency with results for a modern RISC machine. Section 3 describes and discusses the latency improvement techniques which are then evaluated in Section 4. Section 5 offers some concluding remarks.

2 Preliminaries

This section sets the context in which this research was performed. It first describes the experimental testbed, and then updates earlier TCP/IP results reported in [6] with measured results for a modern RISC workstation. This update also provides evidence that the base case used in later sections is sound, that is, it is representative of the behavior of commercial TCP/IP implementations.

2.1 Experimental Testbed

The hardware used for all tests consists of two DEC 3000/600 workstations connected over an isolated 10Mbps Ethernet. These workstations use the 21064 Alpha CPU running at 175MHz [30]. The CPU is a 64-bit wide, super-scalar design that can issue up to two instructions per cycle. In practice, there are very few opportunities to dual issue pure integer code. For integer-only systems code, it is therefore more accurate to view the CPU as a single-issue processor.

The memory system features split primary i- and d-caches of 8KB each, a unified 2MB second-level cache (backup-cache, or b-cache), and 64MB of main memory. All caches are direct-mapped and use 32-byte cache blocks. For the i-cache, this implies that a cache block holds 8 instructions. Memory read accesses are non-blocking, which is important since it implies that there is not necessarily a direct relationship between the number of misses and the number of CPU stall cycles induced by these misses. This is because non-blocking loads make it possible to overlap memory accesses with useful computation. The memory system interface is 128 bits wide and the lmbench [20] suite reports a memory access latency of 2, 10, and 48 cycles for a d-cache, b-cache, and main-memory accesses, respectively. When executing straight-line code out of the b-cache, the CPU can sustain an execution rate of 8 instructions per 13 cycles [10].

Unless noted otherwise, all software was implemented in a minimal stand-alone version of the x-kernel [13]. The entire test runs in kernel mode (no protection domain crossings) and without virtual memory. The kernel is so small that it fits entirely into the b-cache and, unless forced (as in some of the tests), there are no b-cache conflicts. All code was compiled using a version of gcc 2.6.0 that was modified to support outlining [31]. While we started with the regular x-kernel distribution, we did apply some modifications in the process of porting it to the Alpha. These modification, which are described in detail elsewhere [22], are summarized below:

- **D-cache optimizations:** First, data structures were reorganized to minimize compiler-introduced padding and to co-locate structure members that are accessed together. Second, the kernel was adapted to use continuations [8] and stacks that are first-class objects so as to minimize the number of stacks in use and to allow managing them in a cache-friendly last-in-first-out manner. Third, the hash-table manager was changed to allow efficient visiting of all table-elements. This obviates the need for TCP to maintain some of its state in multiple representations, thus reducing TCP's d-cache footprint.

- **Conditional and careful inlining:** Conditional inlining is a technique that allows inlining a function provided that a subset of the function's actual arguments is constant. This allows generating inline code for the simple cases without forcing inlining for the complex cases where the resulting code inflation would be unacceptably large. Careful inlining limits the use of inlining to the cases that will result in improved performance even for latency sensitive code. This is in contrast to the blind inlining that is often used when optimizing execution in tight loops.

- **Fixing machine idiosyncrasies:** The LANCE [1] Ethernet adapter present in the test machines employs a DMA engine that results in memory being used sparsely: every two bytes of data is followed by a 2 byte gap. The Universal Stub Compiler [24] was used to allow accessing such memory efficiently and conveniently. Also, the Alpha architecture does not support sub-word load and store operations. The critical-path code size of TCP was dramatically reduced by changing a few type declarations to use full words instead of sub-words. The resulting increase in d-cache footprint was very modest so that the change was well warranted.

2.2 Test Protocols

As the goal of this research is to test a set of latency improving techniques on protocol stacks that are representative of networking code, we use two protocol stacks that differ greatly in design and implementation: a conventional TCP/IP stack and a generic RPC stack. TCP/IP was chosen primarily because of its ubiquitous nature that facilitates comparison with other work on latency-oriented optimizations. In contrast, the RPC stack is a model case for the x-kernel paradigm that encourages decomposing networking functionality into many small protocols.

The organization of the two protocol stacks is shown in Figure 1. The left side shows the TCP/IP stack. At the top is TCPTEST, a simple, ping-pong test program. Below are TCP and IP which are the x-kernel versions of the corresponding Internet protocols [28, 27]. The x-kernel implementation of TCP is based on BSD source code so, except for interface changes, they are identical. VNET is a virtual protocol [23] that routes outgoing messages to the appropriate network adapter. In BSD-derived implementations, VNET is normally part of IP. ETH is the device-independent half of the Ethernet driver, whereas LANCE is the device-dependent half for the network adapter present in the DEC 3000 machine.

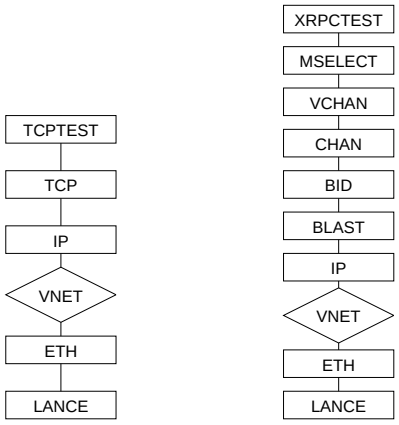

Figure 1: Test Protocol Stacks

The right side of Figure 1 shows the RPC stack. It implements a remote procedure call facility similar to Sprite RPC [25]. Since the x-kernel-paradigm encourages stacks with many small (minimal) protocols, RPC is considerably taller than TCP/IP. At the top is XRPCTEST, which is the RPC-equivalent of the ping-pong test implemented in TCPTEST. MSELECT, VCHAN, CHAN, BID and BLAST together provide the desired RPC semantics. A detailed description of these protocols can be found in [23].

2.3 Base Case

We first establish that the base case used in later sections is sound. To do this, we compare the x-kernel TCP/IP stack with the one implemented in DEC Unix v3.2c. While there is always room for improvement, the Unix implementation qualifies as well-optimized code.

Because of large structural differences between the two implementations, it is not meaningful to compare end-to-end numbers. For example, on Unix, network communication involves crossing the user-kernel boundary whereas in the x-kernel, all execution is in kernel mode. Instead, we compare TCP/IP input processing, which is almost identical for the two cases. The necessary data was collected by instrumenting both kernels with a tracing facility. This facility allows collecting instruction traces of actual working code.[1] Combined with execution times obtained using the CPU's cycle counter, this allows a detailed comparison of processing overheads. It also provides us with the opportunity to update [6] with results for a modern RISC workstation.

The results reported in Table 1 are for the case where a one byte TCP segment arrives on a bi-directional connection. This is in contrast to [6] where the focus was on unidirectional connections (as is the case for an ftp transfer, for example). Using a bi-directional connection is more realistic to measure latency since if data flows in only one direction, it is usually possible to send large packets, and for large packets, processing time is dominated by data-dependent costs [7]. In contrast, with a request-response style of communication, small, latency-sensitive messages are quite common.

The distinction between uni- and bi-directional data connections matters for two reasons. First, with a bi-directional connection, both end-hosts perform sender and receiver-related house-keeping. With a unidirectional connection, each host performs one function, but not the other. Second, the DEC Unix implementation uses header-prediction [6]. This is an optimization primarily targeted at improving latency. Unfortunately, TCP header-prediction works only for uni-directional connections. The result is that for a bi-directional connection—which is the case for which latency is most critical—header prediction slightly worsens latency. However, with less than a dozen additional instructions executed, the slow down is not significant.

Architecture:	**80386**	**Alpha**	
TCP/IP implementation:	**[6]:**	**Unix v3.2c:**	**Improved: x-kernel:**
# of instruction executed...			
... in `ipintr`:	57	248	
... in `tcp_input`:	276	406	
... from IP to TCP input:		262	437
... from TCP to socket input:		1188	1004
CPI:		4.3	3.3

Table 1: Comparison of TCP/IP Implementations

[1] The traces are available via anonymous ftp from `ftp://cage.cs.arizona.edu/pub/davidm/tcpip`.

The first row in Table 1 shows the number of instructions executed in function `ipintr` (IP input processing). The second row gives the number of instructions executed in `tcp_input` *after* the TCP control block has been found (i.e., after function `in_pcblookup` has returned). The IP count for the 80386 processor was taken directly from [6]. The TCP input processing count of 276 instructions was arrived at by adding both the sender side and the receiver side costs, as well as the common path cost reported in [6] (154 instructions for the common path, $15 + 17$ additional instructions for the receive side processing, and $9+20+17+44$ additional instructions for the sender side processing). The DEC Unix numbers were measured as described above. Since the x-kernel source code is organized differently, it is not possible to report corresponding x-kernel numbers.

First, consider the results for `tcp_input`. The DEC Unix trace is roughly 50% longer than the 80386 count. Such a code inflation is not uncommon when converting CISC code to RISC code, especially considering that the traced Alpha code does not have sub-word loads and stores available [3]. Second, comparing the `ipintr` results, we notice that IP input processing on the Alpha appears to be more than a factor of four longer than on the 80386. We believe this to be more an artifact of how the counting was performed rather than a real difference. For example, the DEC Unix implementation has the IP header checksum inlined. This artificially inflates the `ipintr` count by 42 instructions. Even though the checksum alone does not fully explain the large discrepancy, it serves to illustrate that it is probably not a good idea to count instructions executed in a specific function since such a count depends heavily on implementation details (e.g., amount of inlining). Instead, we suggest to count the number of instructions required to complete an entire task. This also makes it easier to compare different implementations of the same protocol stack.

Thus, to compare IP processing between Unix and the x-kernel, we counted the number of instructions executed between the point where an incoming packet enters IP and the point where it enters TCP. For BSD-derived implementations, this covers the instructions executed from the point where `ipintr` is called up to the point where `tcp_input` is called. The corresponding x-kernel functions are `ipDemux` and `tcpDemux`. Similarly, for TCP we counted the number of instructions executed between entering TCP and the point where data is delivered to the layer above TCP. For BSD-derived implementations, this is the code executed between the calls to `tcp_input` and `sowakeup`. The corresponding x-kernel calls are `tcpDemux` and `clientStreamDemux`. Equivalent counts for the 80386 are not available.

As the third and fourth rows in Table 1 illustrate, DEC Unix is doing a little better during IP processing while the x-kernel implementation is better during TCP processing. Overall, the two traces have almost the same length (1450 versus 1441). As the last row shows, the average number of cycles required to execute each instruction (CPI) is 3.3 for the x-kernel and 4.3 for Unix, so in terms of actual execution time, the x-kernel is quite a bit faster. The important point, however, is that the similarity in code-path length provides evidence for the claim that the x-kernel TCP/IP implementation is indeed representative of production-quality implementations.

3 Latency Reducing Techniques

This section describes three techniques that we evaluated as a means to reduce protocol-processing latency. Unlike many other optimization techniques that improve execution speed by reducing the number of instructions executed, these techniques are primarily targeted at reducing the *cost* for each instruction.

3.1 Outlining

As the name suggests, outlining is the opposite of inlining. It exploits the fact that not all basic blocks in a function are executed with equal frequency. For example, error handling in the form of a kernel panic is clearly expected to be a low-frequency event. Unfortunately, it is rarely possible for a compiler to detect such cases based only on compile-time information. In general, basic blocks are generated simply in the order of the corresponding source code lines. For example, the sample C source code shown on the left is often translated to machine code of the form shown on the right:

```
                                :
   :                          load      r0,(bad_case)
if (bad_case) {               jump_if_0 r0,good_day
  panic("bad day");           load_addr a0,"bad day"
}                             call      panic
printf("good day");         good_day:
   :                          load_addr a0,"good day"
                              call      printf
                                :
```

The above machine code is suboptimal for two reasons: (1) it requires a jump to skip the error handling code, and (2) it introduces a gap in the i-cache if the block size is larger than one instruction. A taken jump often results in pipeline stalls and i-cache gaps waste memory bandwidth because useless instructions are loaded into the cache. This can be avoided by moving error handling code out of the main line of execution, that is, by *outlining* error handling code. For example, error handling code could be moved to the end of the function or to the end of the program.

Outlining traditionally has been associated with profile-based optimizers [12, 26]. Profile-based optimizers are aggressive rather than conservative—any code that is not covered by the collected profile will be outlined. They also make it difficult to map the changes back to the source code level, so it is not easy to verify that a collected profile is indeed (sufficiently) exhaustive. Finally, relatively simple changes to the source code may require collecting a new profile all

over again. The main advantage of profile-based optimizing is that it can be easily automated.

Due to the above drawbacks and the unique opportunities present in networking and low-level systems code, our outlining approach is language-based and conservative. Being conservative, it may miss outlining opportunities and be less effective than a profile-based approach. However, systems code is unique in that it contains much code that can be outlined trivially. For example, it is not uncommon to find functions that contain up to 50% error checking/handling code. Just outlining these obvious cases can result in dramatic code-density improvements. Since the approach is language based, the source code explicitly indicates what portions of the code get outlined. That is, outlining gives full control to the systems programmer.

We modified the GNU C compiler such that if-statements can be annotated with a static prediction as to whether the if-conditional will mostly evaluate to TRUE or FALSE. If-statements with annotations will have the machine code for the unlikely branch generated at the end of the function. Unannotated if-statements are translated as usual. With this compiler-extension, the code on the left is translated into the machine code on the right:

```
   :                          :
if (bad_case @ 0) {        load      r0,(bad_case)
   panic("bad day");       jump_if_not_0 r0,bad_day
}                          load_addr a0,"good day"
printf("good day");        call      printf
   :                     continue:
                              :
                          return_from_function

                         bad_day:
                          load_addr a0,"bad day"
                          call      panic
                          jump      continue
```

Note that the if-conditional is followed by an `@0` annotation. This tells the compiler that the expression is expected to evaluate to FALSE most of the time. In contrast, the annotation `@1` would mark a mostly-TRUE expression. For portability and readability, these annotations are normally hidden in C pre-processor macros.

The above machine code avoids the taken jump and the i-cache gap at the cost of an additional jump in the infrequent case. Corresponding code will be generated for if-statements with an else-branch. In that case, the static number of jumps remains the same, however. It is also possible to use if-statement annotations to direct the compiler's optimizer. For example, it would be reasonable to give outlined code low-priority during register allocation. Our present implementation does not yet exploit this option.

As alluded to before, outlining should not be applied overly aggressively. In practice, we found the following three cases to be good candidates for outlining:

1. Error handling. Any kind of expensive error handling can be safely outlined. Error handling is expensive, for example, if it requires a reboot of the machine, console I/O, or similar actions.

2. Initialization code. Any code along the critical path of execution that is executed only once (e.g., at system startup) can be outlined.

3. Unrolled loops. The latency sensitive case usually involves so little data processing that unrolled loops are never entered. If there is enough data for an unrolled loop to be entered, execution time is typically dominated by data-dependent costs, so that the additional overheads due to outlining are insignificant.

We found that outlining alone does not make a huge difference in end-to-end latency. However, the code density improvements that it achieves are essential to the effectiveness of the next two techniques: cloning and path-inlining.

3.2 Cloning

Cloning involves creating a copy of a function. The cloned copy can be relocated to a more appropriate address and/or optimized for a particular use. For example, if the TCP/IP path is executed frequently, it may be desirable to pack the involved functions as tightly as possible. The resulting increase in code-density can improve i-cache, TLB, and paging behavior. The longer cloning is delayed, the more information is available to specialize the cloned functions. For example, if cloning is delayed until a TCP/IP connection is established, most connection state will remain constant and can be used to partially evaluate the cloned function. This achieves similar benefits as code synthesis [17]. Just as for inlining, cloning is at odds with locality of reference. Cloning at connection creation time will lead to one cloned copy per connection, while cloning at protocol stack creation time will require only one copy per protocol stack. By choosing the point at which cloning is performed, it is possible to tradeoff locality of reference with the amount of specialization that can be applied.

Cloning can be considered the next logical step following outlining—the latter improves (dynamic) instruction density within a function, while the former achieves the same across functions. Figure 2 summarizes the effects that outlining and cloning have on the i-cache footprint. The left column shows many small i-cache gaps due to infrequently executed code. As shown in the middle column, outlining compresses frequently executed code and moves everything else to the end of the function. The right column shows that cloning leads to a contiguous layout for clone A and clone B. Note that this particular example assumes that the clones and the original functions can share the outlined code. Whether this is possible depends on architectural details. For the Alpha, sharing is normally possible. Where sharing is not possible, cloning places a copy of the outlined code behind all the frequently executed code.

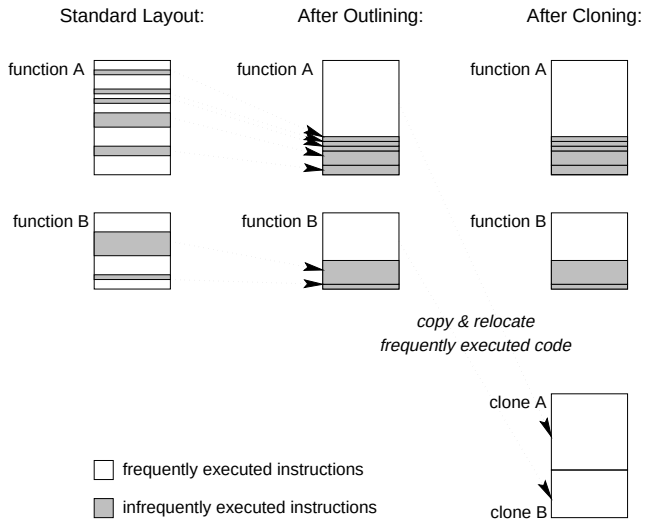

Figure 2: Effects of Outlining and Cloning

We implemented runtime cloning as a means to allow flexible experimentation with various function positioning algorithms. Cloning currently occurs when the system is booted (not when a connection is established) and supports only very simple code specialization. Code specialization is specific to the Alpha architecture and is targeted at reducing function call overheads. In particular, under certain circumstances, the Alpha calling convention allows us to skip the first few instructions in the function prologue. Similarly, if a caller and callee are spatially close, it is possible to replace a jump to an absolute address with a PC-relative branch. This typically avoids the load instruction required to load the address of the callee's entry point and also improves branch-prediction accuracy.

We experimented extensively with different layout strategies for cloned code. We thought that, ideally, it should be possible to avoid all i-cache conflicts along a critical path of execution. With a direct-mapped i-cache, the starting address of a function determines exactly which i-cache blocks it is going to occupy [19]. Consequently, by choosing appropriate addresses, it is possible to optimize i-cache behavior for a given path. The cost is that this fine-grained control of function-placement occasionally makes it necessary to introduce gaps between two consecutive functions (sometimes it is possible to fill a gap with another function of the appropriate length). Gaps have the obvious cost of occupying main memory without being of any direct use. More subtly, if i-cache blocks are larger than one instruction, fetching the last instructions in a function will frequently result in part of a gap being loaded into the i-cache as well, thereby wasting memory bandwidth.

We devised a tool employing simple heuristics that, based on a trace-file, computed a layout that minimizes replacement misses without introducing too many additional gaps. We call this approach *micro-positioning* because function placement is controlled down to size of an individual instruction. I-cache simulation results were encouraging—it was possible to reduce replacement misses by an order of magnitude (from 40, down to 4), while introducing only four or five new cold misses due to gaps.

However, when performing end-to-end measurements, a much simpler layout strategy consistently outperformed the micro-positioning approach. The simpler layout strategy achieves what we call a *bipartite layout*. Cloned functions are divided into two classes: *path* functions that are executed once and *library* functions that are executed multiple times per path invocation. There is very little benefit in keeping path functions in the cache after they executed, as there is no temporal locality unless the entire path fits into the i-cache. In contrast, library functions should be kept cached starting with the first and ending with the last invocation. Based on these considerations it makes sense to partition the i-cache into a path partition and a library partition. Within a partition, functions are placed in the order in which they are called. Such a sequential layout maximizes the effectiveness of prefetching hardware that may be present. This layout strategy is so simple that it can be computed easily at runtime—the only dynamic information required is the order in which the functions are invoked. In essence, computing a bipartite layout consists of applying the well-known "closest-is-best" strategy to the library and path partition *individually* [26].

Establishing the performance advantage of the bipartite layout relative to the micro-positioning approach is difficult since small changes to the heuristics of the latter approach resulted in large performance variations. The micro-positioning approach usually performed somewhat worse than a bipartite layout and sometimes almost equally well, but never better. It is not entirely obvious why this is so and it is impossible to make any definite conclusions without even more fine-grained simulations, but we have three hypotheses. First, micro-positioning leads to a non-sequential memory access pattern because a cloned function is positioned wherever it fits best, that is, where it incurs the minimum number of replacement misses. It may be this nearly random access pattern that causes the overall slowdown. Second, the gaps introduced by the micro-positioning approach do cost extra memory bandwidth. This hypothesis is corroborated by the fact that we have not found a single instance where aligning function entry-points or similar gap-introducing techniques would have improved end-to-end latency. Note that this is in stark contrast with the findings published in [11], where i-cache optimization focused on functions with a very high degree of locality. So it may be that micro-positioning suffers because of the memory bandwidth wasted on loading gaps. Third, the DEC 3000/600 workstations used in the experiments employ a large second-level cache. It may be the case that the initial i-cache misses also missed in the second-level cache. On the other hand, i-cache replacement misses are almost guaranteed to result in a second-level cache hit. Thus,

it is quite possible that 36 replacement misses were cheaper than four or five additional cold misses introduced by micro-positioning.

Despite the unexpected outcome, the above result is encouraging. In order to improve i-cache performance, it is not necessary to compute an optimal layout—a simple layout-strategy such as the bipartite layout appears to be just as good (or even better) at a fraction of the cost. We would like to emphasize that the bipartite layout strategy may not be appropriate if all the path and library functions can fit into the i-cache. If it is likely that the path will remain cached between subsequent path-invocations, it is better to use a simple linear allocation scheme that allocates functions strictly in the order of invocation, that is, without making any distinction between library and path functions. This is a recurrent theme for cache-oriented optimizations: the best approach for a problem that fits into the cache is often radically different from the best one for a problem that exceeds the cache size.

3.3 Path-Inlining

The third latency reducing technique is path-inlining. This is an aggressive form of inlining where the entire latency-sensitive path of execution is inlined to form a single function. Since the resulting code is specific for a single path, this is warranted only if the path is executed frequently. It is important to limit inlining to path-functions: by definition, library-functions are used multiple times during a single path executions, so it is better to preserve the locality of reference that they afford. Also, inlining library-functions would likely lead to an excessive growth in code size.

The advantage of path-inlining is that it removes almost all call overheads and greatly increases the amount of context available to the compiler for optimization. For example, in the x-kernel's VNET protocol, output processing consists of simply calling the next lower layer's output function. With path-inlining, the compiler can trivially detect and eliminate such useless call overheads.

While path-inlining is easy in principle, the practical problem is quite difficult. None of the common C compilers are able to inline code across module boundaries (object files). There are tools available that assist in doing so, but the ones that we experimented with were not reliable enough to be of much use. While it should not be very difficult to add cross-module inlining to an existing C compiler, in our case it appeared more effective to apply the require transformations manually.

We applied path-inlining to both the TCP/IP and the RPC stacks. In the TCP/IP case, this resulted in collapsing the entire stack into two large functions: one for input processing and one for output processing. Roughly the same applies for the RPC stack, although the split is slightly different: one function takes care of all the processing in protocols XRPCTEST, MSELECT, VCHAN as well as the output processing in CHAN and the protocols below it, whereas the other function handles all input processing up to the CHAN protocol.

3.4 Support for Paths

The second two techniques described in this section—cloning and path-inlining—depend on knowing the exact sequence of functions that a given packet is going to traverse. While this is usually the case for outbound packets, knowing the path an incoming packet is going to follow is problematic. This is because traditional networking code discovers the path of execution incrementally and as part of other protocol processing: a protocol's header contains the identifier of the next higher-level protocol, and this higher-level protocol identifier is then mapped into the address of the function that implements the appropriate protocol processing. In other words, processing incoming packets involves indirect function calls that cannot be inlined in general.

To make cloning and path-inlining work for this important case, it is necessary to *assume* that a packet will follow a given path, generate path-inlined and cloned code for that assumed path, and then, at run-time, establish that an incoming packet really will follow the assumed path. We have designed and implemented a new OS, called Scout, that extends the x-kernel by adding an explicit path abstraction [21]. An essential component of Scout is a packet classifier that determines which path a given packet will traverse [2, 18, 33, 9]. Thus, such a system provides exactly the information necessary to use cloning and path-inlining.

4 Evaluation

This section evaluates the techniques presented in the previous section. It first describes the specific test cases and then follows with a presentation of end-to-end latency results and a detailed, trace-based analysis of processing behavior. For brevity, no throughput measurements are presented but we note that none of the techniques had a negative effect on it. In fact, as is commonly the case, the latency improvements also resulted in a slightly higher throughput.

4.1 Test Cases

Recall that all measurements were performed in the environment described in Section 2.1. Both the TCP/IP and RPC stacks were measured in several configurations. The configurations were selected to allow us to gauge the effect of each technique. An exhaustive measurement of all possible combinations would have been impractical, so we focus on the following six version and supply additional data where needed.

- **STD:** This is the standard version from Section 2.3. It uses none of the latency-reducing optimizations described in Section 3.

- **OUT:** Like STD, but includes outlining.
- **CLO:** Like OUT, but includes cloning (using a bipartite layout).
- **BAD:** Like CLO, but cloning has been used to artificially *worsen* the i-cache behavior. While not strictly a worst-case scenario, this version is used to establish the potential of i-cache effects to influence processing latency. Specifically, for the TCP stack, version BAD results in 217 additional i-cache and 110 additional b-cache misses (relative to CLO, which has 483 i-cache and 678 b-cache misses). For the RPC stack, it results in 233 additional i-cache and 14 additional b-cache misses (relative to CLO with 488 i-cache and 845 b-cache misses).
- **PIN:** Like OUT, but includes path-inlining.
- **ALL:** Like PIN, but cloning (bipartite layout) has been used to improve i-cache behavior. That is, this version uses all techniques, and is expected to achieve the best performance.

Note that path-inlining and cloning require running a packet classifier on incoming packets since the optimized code is no longer general enough to handle all possible packets. Currently, the best software classifiers add an overhead of about $1 - 4\mu$s per packet [2, 9]. The results presented in this section do not include the time required to classify packets. This seems justified since systems commonly use classifiers to improve functionality, flexibility, and accountability [16, 21]. In a system that already makes use of a classifier, the benefits of cloning and path-inlining are truly the ones reported below. The same applies for a system that uses a hardware classifier or a protocol stack that contains an explicit path-identifier.

4.2 End-to-End Results

TCP and RPC latency was measured by ping-ponging packets with no payload between a server and a client machine. Since TCP is stream-oriented, it does not send any network packets unless there is data to be sent. Thus, the "no payload" case is approximated by sending 1B of data per message. For both protocol stacks, the tests result in 64-byte frames on the wire since that is the minimum frame size for Ethernet. The reported end-to-end latency is the average time it took to complete one roundtrip in a test involving 100,000 roundtrips. Time was measured with a clock running at 1024Hz, thus yielding roughly a 1ms resolution.

For the TCP/IP stack, the optimizations were applied to both the server and client side. Since the processing on the server and client side is almost identical, the improvement on each side is simply half of the end-to-end improvement. For the RPC stack, the optimizations were restricted to the client side. On the server side, the configuration yielding the best performance was used in all measurements (which happened to be the ALL version). Always running the same RPC server ensures that the reference point remains fixed and allows a meaningful analysis of client performance.

	TCP/IP		**RPC**	
Version	T_e [μs]	Δ [%]	T_e [μs]	Δ [%]
BAD	498.8± 0.29	+60.5	457.1± 0.20	+25.1
STD	351.0± 0.28	+12.9	399.2± 0.29	+9.2
OUT	336.1± 0.37	+8.1	394.6± 0.10	+8.0
CLO	325.5± 0.07	+4.7	383.1± 0.20	+4.8
PIN	317.1± 0.03	+2.0	367.3± 0.19	+0.5
ALL	310.8± 0.27	+0.0	365.5± 0.26	+0.0

Table 2: End-to-end Roundtrip Latency

Table 2 shows the end-to-end results. The rows are sorted according to decreasing latency, with each row giving the performance of one version of the TCP/IP and RPC stacks. The performance is reported in absolute terms as the mean roundtrip time plus/minus one standard deviation, and in relative terms as the per cent slow-down compared to the fastest version (ALL). For TCP/IP, the mean and standard deviation where computed based on ten samples; five samples were collected for RPC.

As the table shows, the BAD version of the TCP/IP stack performs by far the worst. With almost 500μs per roundtrip, it is over $173\mu s$ slower than version CLO, which corresponds to a slowdown of more than 53%. As explained above, the code in the two versions is mostly identical. The only significant difference is the layout of that code. This clearly shows that i-cache effects can have a profound effect on end-to-end latency.

Row STD shows that the standard x-kernel version of the protocol stacks has a much better cache behavior than BAD. That version is slower by about 12.9% for TCP/IP and 9.2% for RPC. There are two reasons why STD performs relatively well. First, earlier x-kernel experiences with direct-mapped caches led to attempts to improve cache performance by manually changing the order in which functions appear within the object files and by changing the link order. Because of such manual tuning, the STD version has a reasonably good cache behavior to begin with. Second, it also appears to be the case that the function usage pattern in the x-kernel is such that laying the functions out in the address space in what basically amounts to a random manner, yields an average performance that is closer to the best case than to the worst case. This is especially true since in the latency sensitive case, there are few loops that have the potential for pathological cache-behavior. Keep in mind, however, that case BAD is possible in practice unless the cache layout is controlled explicitly. The techniques proposed in this paper provide sufficient control to avoid such a bad layout.

Row OUT indicates that outlining works quite well for TCP/IP—it reduces roundtrip time by about 15μs when com-

pared to STD. Since both the client and the server use outlining, the reduction on the client side is roughly half of the end-to-end reduction, or 7.5μs. In contrast, at a 4.6μs savings, outlining makes a smaller difference to the RPC stack. This is because TCP consists of a few large functions that handle most of the protocol processing (including connection establishment, tear-down, and packet retransmission), whereas RPC consists of many small functions that often handle exceptional events through separate functions. In this sense, the RPC code is already structured in a way that handles exceptional events outside the performance critical code path. Nevertheless, outlining does result in significantly improved performance for both protocol stacks.

In contrast, row CLO indicates that cloning works better for RPC than for TCP. In the former case, the reduction on the client side is about 11.5μs whereas in the latter case the client-side reduction is roughly 5.3μs. This makes sense since TCP/IP absorbs most of its instruction locality in a few, big functions, meaning that there are few opportunities for self-interference. The many-small-function structure of the RPC stack makes it likely that the uncontrolled layout present in version OUT leads to unnecessary replacement misses. Conversely, this means that there are good opportunities for cloning to improve cache effectiveness.

Path-inlining also appears to work very well for the RPC stack. Since PIN is the same as version OUT with path-inlining enabled, it is more meaningful to compare it to the outlined version (OUT), rather than the next best version (CLO). If we do so, we find that the TCP/IP client side latency is about 9.5μs and the RPC client side about 27.3μ below the corresponding value in row OUT. Again, this is consistent with the fact that the RPC stack contains many more—and typically much smaller—functions than TCP. Just eliminating call-overheads through inlining improves the performance of the RPC stack significantly.

Finally, row ALL shows the roundtrip latency of the version with all optimizations applied. As expected, it is indeed the fastest version. However, the client-side reduction for TCP/IP compared to PIN is only about 3.1μs and the improvement in the RPC case is a meager 1.8μs. That is, with path-inlined code, partitioning library and path functions does not increase performance much further.

While end-to-end latency improvements are certainly respectable, they are nevertheless fractional on the given test system. It is important to keep in mind, however, that modern high-performance network adapters have much lower latency than the LANCE Ethernet adapter present in the DEC 3000 system [1]. To put this into perspective, consider that a minimum-sized Ethernet packet is 64 bytes long, to which an 8 byte long preamble is added. At the speed of a 10Mbps Ethernet, transmitting the frame takes 57.6μs. This is compounded by the relative tardiness of the LANCE controller itself: we measured 105μs between the point where a frame is passed to the controller and the point where the "transmission complete" interrupt handler is invoked. The LANCE overhead of 47.4μs is consistent with the 51μs figure reported elsewhere for the same controller in an older generation workstation [32]. Since the latency between sending the frame and the receive interrupt on the destination system is likely to be higher, and since each roundtrip involves two message transmissions, we can safely subtract $105\mu\text{s} \times 2 = 210\mu$s from the end-to-end latency to get an estimate of the actual processing time involved. For example, if we apply this correction to the TCP/IP stack, we find that version BAD is actually 186% slower than the fastest version. Even version STD is still 40% slower than version ALL.

Table 3 revisits the end-to-end latency numbers, adjusted to factor out the overhead imposed by the controller and Ethernet. While there will obviously be some additional latency, one should expect roundtrip times on the order of 50μs rather than the 210μs measured on our experimental platform.[2]

	TCP/IP		**RPC**	
Version	T_e [μs]	Δ [%]	T_e [μs]	Δ [%]
BAD	288.8	+186.5	247.1	+59.0
STD	141.0	+40.2	189.2	+21.7
OUT	126.1	+25.1	184.6	+18.7
CLO	115.5	+14.6	173.1	+11.3
PIN	107.1	+6.3	157.3	+1.2
ALL	100.8	+0.0	155.5	+0.0

Table 3: End-to-end Roundtrip Latency Adjusted for Network Controller

4.3 Detailed Analysis

The end-to-end results are interesting to establish global performance effects, but since some of the protocol processing can be overlapped with network I/O, they are not directly related to CPU utilization. Also, it is impossible to control all performance parameters simultaneously. For example, the tests did not explicitly control data-cache performance. Similarly, there are other sources of variability. For example, the memory free-list is likely to vary from test case to test case (e.g., due to different memory allocation patterns at startup time). While not all of these effects can be controlled, most can be measured.

Towards this end, we collected two additional sets of data. The first is a set of instruction traces that cover most of the protocol processing. The second is a set of fine-grained measurements of the execution time of the traced code. The instruction traces do not cover all of the processing since the tracing facility did not allow the tracing of interrupt handling. Other than that, the traces are complete. For the sake of brevity, we only summarize the most important results; see [22] for a more detailed discussion.

[2] Numbers is this range have been reported in the literature for FDDI and ATM controllers [7].

	TCP/IP				RPC			
	T_p [μs]	Length	iCPI	mCPI	T_p [μs]	Length	iCPI	mCPI
BAD	167.0± 1.75	4718	1.61	4.58	154.2± 0.47	4253	1.69	4.66
STD	89.6± 0.34	4750	1.72	1.58	85.1± 0.53	4291	1.78	1.69
OUT	84.1± 0.12	4728	1.61	1.50	81.0± 0.16	4257	1.68	1.65
CLO	77.2± 0.36	4684	1.61	1.28	71.0± 0.29	4227	1.69	1.25
PIN	69.9± 0.48	4245	1.57	1.31	57.7± 0.18	3471	1.66	1.25
ALL	66.1± 0.48	4215	1.57	1.17	49.2± 0.12	3468	1.67	0.81

Table 4: Protocol Processing Costs

4.3.1 Processing Time Measurements

Given the execution traces and timings, it is possible to derive a number of interesting quantities: protocol processing time, CPI, and, most importantly, the memory CPI (mCPI). Protocol processing time was measured with the CPU's cycle counter. Thus, the CPI is obtained by dividing the cycle count by the trace length. The memory CPI can then be calculated by subtracting the instruction CPI (iCPI). The iCPI is the average number of cycles spent on each instruction assuming a perfect memory system. That is, the stall cycles in this quantity are purely due to data-dependencies and limitations in the CPU. The iCPI was derived from the instruction trace by feeding it into a CPU simulator. The simulator is somewhat crude with respect to branches as it simply adds a fixed penalty for each taken branch. Other than that, the simulator has almost perfect accuracy.

The trace-based data is shown in Table 4. Columns T_p shows the measured processing time in micro-seconds. As before, this is shown as the sample mean plus/minus the sample standard deviation. The column labeled Length gives the trace length as an instruction count. Columns mCPI and iCPI are the memory and instruction CPI values, respectively.

Looking at the iCPI columns, we find that both the TCP/IP and RPC stacks break down into three classes: the standard version has the largest iCPI, the versions using outlining (BAD, OUT, CLO) have the second largest value, and the path-inlined versions have the smallest value. This is expected since the code within each class is largely identical. Since the CPU simulator adds a fixed penalty for each taken branch, the decrease in the iCPI as we go from the standard version to the outlined versions corresponds directly to the reduction in taken branches. Interestingly, outlining improves iCPI by almost exactly 0.1 cycles for both protocol stacks. This is a surprisingly large reduction considering that path-inlining achieves a reduction of 0.04 cycles at the most. We expected that the increased context available in the inlined versions would allow the compiler to improve instruction scheduling more. Since this does not seem to be the case, the performance improvement due to path-inlining stems mostly from a reduction in the critical-path code size. Note that even in the best case, the iCPI value is still above 1.5. While some of this can be attributed to suboptimal code generation on the part of gcc 2.6.0, a more fundamental reason for this large value is the structure of low-level systems code: the traces show that there is very little actual computation, but much interpretation and branching.

The mCPI columns in the table show that, except for the RPC case of ALL, the CPU spends well above 1 cycle per instruction waiting for memory (on average). Comparing the mCPI values for the various versions, we find that the proposed techniques are rather effective. Both protocol stacks achieve a reduction by a factor of more than 3.9 when going from version BAD to version ALL. Even when comparing version ALL to STD we find that the latter has an mCPI that is more than 35% larger. In terms of mCPI reduction, cloning with a bipartite layout and path-inlining are about equally effective. The former is slightly more effective for the TCP/IP stack, but in the RPC case, both achieve a reduction of 0.4 cycles per instruction. Combining the two techniques does have some synergistic effects for TCP/IP. The additional reduction compared to outlining or path-inlining alone is small though, on the order of 0.11 to 0.14 cycles per instruction. Of all the mCPI values, the value 0.81 for the ALL version of the RPC stack clearly stands out. Additional data presented in [22] leads us to believe that the value is an anomaly: just small changes to the code lead to mCPI values more in line with the other results. This serves as a reminder that while the proposed techniques improve cache behavior, it is nearly impossible to achieve perfect control for any reasonably complex system.

4.3.2 Outlining Effectiveness

The results presented so far are somewhat misleading in that they underestimate the benefits of outlining. While it does achieve performance improvements in and of itself, it is more important as an enabling technology for path-inlining and cloning. Thanks to outlining, the amount of code replication due to path-inlining and cloning is greatly reduced. Together with this size reduction goes an increase in dynamic instruction density, that is, less memory bandwidth is wasted loading useless instructions. This is demonstrated quantitatively with the results presented in Table 5. It shows that without outlining (STD version), around 20% of the instructions loaded into the cache never get executed. For both protocol stacks, outlining reduces this by roughly a factor of 1.4. It is illustra-

tive to consider that a waste of 20% corresponds to 1.8 unused instructions per eight instructions (one cache block). Outlining reduces this to about 1.3 unused instructions. This is a respectable improvement, especially considering that outlining was applied conservatively.

	Without Outlining		**With Outlining**	
	i-cache unused	Size	i-cache unused	Size
TCP/IP	21%	5841	15%	3856
RPC	22%	5085	16%	3641

Table 5: Outlining Effectiveness

The table also shows that outlining results in impressive critical-path code-size reductions. The Size columns show the static code size (in number of instructions) of the latency critical path before and after outlining. In the TCP/IP stack, about 1985 instructions could be outlined, corresponding to 34% of the code. Note that this is almost a full primary i-cache worth of instructions (8KB). Similarly, in the RPC stack 28% of the 5085 instructions could be outlined. This reinforces our claim that outlining is a useful technique not only because of its direct benefits, but also as a means to greatly improve cloning and path-inlining effectiveness. Minimizing the size of the main-line code improves cloning flexibility and increases the likelihood that the entire path will fit into the cache.

5 Concluding Remarks

Networking system designers have known for some time that memory bandwidth plays a critical role in end-to-end throughput. This paper argues that memory bandwidth also is a major player in protocol processing latency. It demonstrates this with measurements performed on a modern 64-bit workstation. While the quantitative results certainly are machine-specific, it is reasonable to expect that the qualitative conclusions will generalize to most other high-performance RISC-based systems.

Beyond this basic result, the paper describes three techniques that can be applied to networking code to improve the latency situation. These techniques have two benefits. First, they significantly improve execution speed by reducing the mCPI. Fundamentally, this reduction is achieved by (a) increasing the dynamic instruction stream density, (b) reducing the number of cache conflicts, and (c) reducing the critical-path code size. The impact of these techniques will grow rapidly as the gap between processor and memory speeds widens. For example, this research was conducted on a machine with a 175MHz Alpha processor, a 100MB/s memory system, and a 10Mbps Ethernet. We now also have in our lab low-cost machines with a 300MHz processor, an 80MB/s memory system, and 100Mbps Ethernet. Second, even though case BAD reported in Section 4 was constructed artificially, sub-optimal configurations are possible and not uncommon in practice. For example, the measured mCPI for the DEC Unix v3.2c TCP/IP stack is 2.3, which is significantly worse than the 1.58 mCPI measured for the standard x-kernel. The proposed techniques make it relatively easy to avoid such bad cache behavior. That is, they help improve the predictability of a system.

Acknowledgments

We would like to thank Allen Brady Montz for implementing a GNU C extension that was used in earlier versions of this work. Several people at DEC provided valuable insight into the Alpha architecture and the 21064 implementation. In particular, we are grateful for Jeffrey Mogul's help. We also thank the anonymous reviewers for providing invaluable feedback.

References

[1] AMD. *Am7990: Local Area Network Controller for Ethernet*.

[2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, 1994.

[3] D. Bhandarkar and D. W. Clark. Performance from architecture: Comparing a RISC and CISC with similar hardware organization. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 310–319.

[4] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. In *Proceedings of SIGCOMM '96 Symposium*, to appear Aug. 1996.

[5] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 120–133, 1993.

[6] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overheads. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[7] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):35–43, July 1993.

[8] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In

Proceedings of the Thirteenth ACM Symposium on Operating System Principles, pages 122–36. Association for Computing Machinery SIGOPS, Oct. 1991.

[9] D. R. Engler, F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 251–266, 1995.

[10] A. Eustace. Personal communication, Oct. 1994.

[11] R. Gupta and C.-H. Chi. Improving instruction cache behavior by reducing cache pollution. In *Proceedings Supercomputing '90*, pages 82–91. IEEE, 1990.

[12] R. R. Heisch. Trace-directed program restructuring for AIX executables. *IBM Journal of Research and Development*, 38(9):595–603, Sept. 1994.

[13] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[14] V. Jacobson. A high performance TCP/IP implementation. Presentation at the NRI Gigabit TCP Workshop, Mar. 18th–19th 1993.

[15] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of SIGCOMM '93 Symposium*, volume 23, pages 259–268, San Fransico, California, Oct. 1993. ACM.

[16] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 244–255, Asheville, North Carolina, Dec. 1993. ACM.

[17] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY 10027, Sept. 1992.

[18] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX Conference*, San Diego, CA, Jan. 1993. USENIX.

[19] S. McFarling. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, Apr. 1989.

[20] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Technical Conference*, pages 120–133, 1996.

[21] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. Technical Report 96-05, University of Arizona, Tucson, AZ 85721, 1996.

[22] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. Technical Report 96-03, University of Arizona, Tucson, AZ 85721, 1996.

[23] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[24] S. W. O'Malley, T. Proebsting, and A. B. Montz. USC: A universal stub compiler. In *Proceedings of SIGCOMM '94 Symposium*, pages 295–306, London, UK, Aug. 31st – Sept. 2nd 1994.

[25] J. K. Ousterhout, A. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, pages 23–35, Feb. 1988.

[26] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 16–27, White Plains, NY, June 1990.

[27] J. Postel. RFC-791: Internet Protocol. Available via ftp from ftp.nisc.sri.com, Sept. 1981.

[28] J. Postel. RFC-793: Transmission Control Protocol. Available via ftp from ftp.nisc.sri.com, Sept. 1981.

[29] B. Prince. Memory in the fast lane. *IEEE Spectrum*, 31(2):38–41, Feb. 1994.

[30] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Burlington, Massachusetts, 1992. Order number EY-L520E-DP.

[31] R. M. Stallman. *Using and Porting GNU CC*, 1992. Manuscript provided by the Free Software Foundation to document `gcc`.

[32] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[33] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *1994 Winter USENIX Conference*, pages 153–165, 1994.