Analysis of Techniques to Improve Protocol Processing Latency

David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley [1]

TR 96-03

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. In particular, 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 up to a factor of 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.

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1] This work supported in part by ARPA Contract DABT63-91-C-0030, by Digital Equipment Corporation.

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 network protocols, such as TCP/IP [KP93, CJRS89, Jac93] and RPC [TL93]. 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. 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, while on the surface it seems that protocol latency should scale with processor speed, this is not necessarily the case. Like throughput, protocol latency is influenced by the processor's memory bandwidth.
- **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 behind improvements in processor speed.
- **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.

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 targetted at improving the mCPI, although they also have a positive effect on the instruction count.

Because these techniques are aimed at improving the mCPI of network software, they are necessarily fine-grain. To be more precise, they can all be characterized as compiler-based techniques, and as such, one might ask if they are specific to network code, or if they are applicable to programs in general (i.e., SPECmark code). The answer is that while it is possible that these techniques are of some benefit to application programs, they are motivated by the unique characteristics of network software (specifically) and low-level systems code (more generally). For example, exception handling code often makes up a large portion of network software. One of our techniques (*outlining*) exploits this fact. As a second example, network software, since it is often layered, typically involves rather deep call chains. A second technique (*path-inlining*) is targetted at this situation. As a final example, network 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 connection. A final technique (*cloning*) addresses this issue.

The paper is organized as follows. Section 2 revisits earlier studies on TCP/IP latency, but this time on RISC processors. This study is interesting in its own right, but also serves as the foundation for the techniques and analysis presented in later sections. Section 3 then describes the latency improvement techniques, and Section 4 reports on an experimental evaluation of these techniques. Note that these techniques are applied to existing protocol implementations written in C; we are not proposing a new programming language. Finally, Section 5 offers some concluding remarks.

2 Starting Point

The work reported in this paper is done in the context of the x-kernel [HP91], a framework that allows flexible experimentation with networking protocols. This section briefly describes the x-kernel and the protocols that we evaluate. In doing so, this section has two goals: (1) to establish a base case that we use to evaluate the techniques introduced in the next section, and (2) to update eariler studies of protocol latency.

2.1 Test Protocols

We begin by emphasizing that the goal of this research is not to create the world's fastest implementation of a given protocol stack, but rather to test a set of latency improving techniques on protocol stacks that are representative of networking code in general. To that end, we tested the techniques both on a TCP/IP and an RPC stack.

TCP/IP is an interesting test-case because of its ubiquitous nature. Applying the techniques to TCP/IP also facilitates comparison with other work on latency-oriented optimizations. The left part of Figure 1 shows the protocol-stack that was used to test TCP/IP. At the top, TCPTEST is a simple, ping-pong-style test program. Below are TCP and IP which are the x-kernel versions of the corresponding Internet protocols [Pos81b, Pos81a]. 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 [OP92] that routes outgoing messages to the appropriate network adaptor. In BSD-derived implementations, VNET's functionality is part of IP. ETH is the device-independent half of the Ethernet driver, while LANCE is the device driver for the LANCE network adaptor that is present in the DEC 3000/600 workstations. The network adaptor connects to the CPU via the TURBOchannel bus [AMD, DEC93].

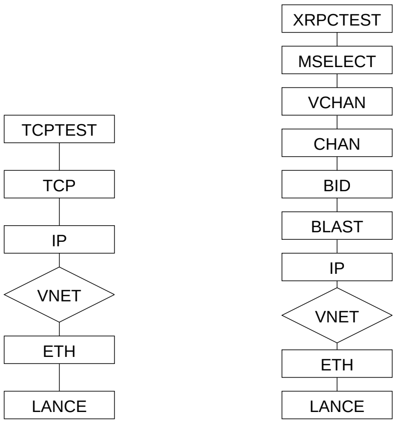

Figure 1: Test Protocol Stacks

The RPC stack implements a remote procedure call facility similar to Sprite RPC [OP92]. This test stack is a model for the x-kernel paradigm that encourages decomposing networking functionality into many small protocols. Consequently, as shown in Figure 1, the RPC stack is deeper than the TCP/IP stack. At the top of the RPC stack we find protocol XRPCTEST which is again a simple ping-pong style test program. The test program sends zero-sized RPC requests to the RPC server which responds with a zero-sized reply. MSELECT, VCHAN, CHAN, BID and BLAST together provide RPC semantics. A detailed description of these protocols can be found in [OP92]. For the purpose of this discussion, it is sufficient to know that the client side of XRPCTEST performs a call into MSELECT to send a request to the server. This call propagates all the way down to the LANCE driver where the request packet is transmitted to the server. After the packet is sent, the calling thread is blocked in CHAN, waiting for the server's reply. The reply packet triggers an interrupt in the LANCE driver, which propagates up the protocol stack to CHAN. CHAN then signals (unblocks) the waiting thread and returns. Eventually, the awakened thread resumes execution and returns to XRPCTEST.

2.2 RISC-Motivated Changes

The x-kernel was originally developed on a 32-bit CISC machine, and as a result, there were several interesting changes that we had to make to produce better latency-sensitive code for a modern RISC machine with a multi-level memory system. Similarly, porting TCP to the 64-bit Alpha processor required some changes. We now summarize these changes to the x-kernel and TCP.

2.2.1 Improving Data Cache Behavior

Strictly speaking, caches are not a RISC invention. However, since many RISC CPUs push CPU clock frequencies to the extreme, main memory accesses impose a much higher relative penalty than other architectures. Making networking code more cache friendly, therefore, is an important step towards reducing processing latency. Whereas the focus is on the i-cache, there were also a few changes to improve the d-cache side. In particular, the x-kernel data structures were reorganized to minimize compiler introduced padding. This is important on the Alpha since pointers and long integers take up 8 bytes, and since such variables must be aligned to their size. For example, placing a pointer behind a byte-sized field normally results in in a 7 byte gap since the compiler needs to ensure proper alignment of the pointer. While reorganizing the structures we were also careful to spatially co-locate structure fields that are used together in close temporal proximity.

The memory object that is used most often during execution is the program stack. To optimize d-cache behavior, we modified the tread manager to make extensive use of continuations [DBRD91]. We also converted stacks to first-class objects. Instead of being statically attached to a given thread, stacks are now dynamically attached to a thread upon demand. Together with continuations, this has the effect that latency sensitive path invocations will normally execute on the same stack. This optimization does not work when a thread blocks with useful state on the stack. To improve this case, stacks are managed with a last-in-first-out policy. This maximizes the chance that a newly attached stack is already in the d-cache.

Another d-cache optimization is to store the same information only once. For example, TCP timer management requires efficient traversal of all open connections. Being BSD-based, the original x-kernel implementation employed a list of open connections for this purpose. However, in the x-kernel there is also a hash-table that is used to demultiplex incoming TCP segments. This hash-table fundamentally contains the same information as the list of open connections. So, if it were possible to efficiently traverse all elements in a hash-table, there would be no need to maintain the separate list of open connections. Unfortunately, hash-tables operate best if they are sparsely populated. Traversing the whole table therefore is relatively inefficient because most of the table-entries (hash-buckets) are empty. Not only is it slow, but it also results in a needlessly large d-cache footprint.

To remedy this situation, we modified the hash-table implementation to contain a list of non-empty hash-table buckets. With this list, the elements in the hash table can be visited simply by following the list of non-empty buckets. Unfortunately, managing the list imposed unacceptably high overheads because removing an element either required a list-traversal or a doubly-linked list. Fortunately, it is possible to evaluate list-removals lazily: when a bucket becomes empty, we can just leave it on the list. The list is then cleaned up the next time we have to traverse the list anyway (i.e., on the next visitation of the elements), at which time removing empty buckets is trivial since it is easy to keep track of the previous non-empty bucket as the list is being traversed. Detailed measurements showed that the modified hash-table manager is substantially faster for hash-table traversals, without significantly affecting the time to insert elements. All other operations—in particular, hash-table lookup—are unaffected. The speedup for hash-table traversals is roughly inversely proportional to the fraction of non-empty buckets in the table. For example, traversing a table with 10% of the buckets populated will be roughly an order of magnitude faster. Thus, at the expense of one pointer per hash-table bucket, it was possible to completely remove the list of open connections in TCP.

2.2.2 Avoiding Unnecessary Work

After the d-cache optimizations, we analyzed the code path that is executed when TCP/IP processes a message. In doing so, we found several opportunities to avoid unnecessary work.

First, execution traces indicated that TCP performs integer multiplication and division on both the inbound and output paths. On the inbound path, the operations are used to update the congestion window. In a latency-sensitive environment (e.g., low-latency LAN), losses are rare and the congestion window is usually fully open. By testing for this common case, it is possible to avoid the expensive integer operations all together. Similarly, on the output side, TCP checks whether it is necessary to send a window update by computing 35 percent of the maximum possible window. By computing 33 percent instead, it is possible to replace the integer multiplication and division by a simple shift and add. This change does not affect the operational properties of TCP noticeably.[1] Integer multiplication and division are quite slow on most processors. This is compounded by the fact that the Alpha architecture does not provide an integer division instruction [Sit92]. Thus, not only is the operation slow, but the function that implements it also takes up instruction space. Removing the large division routine from the critical path of execution reduced the cache-footprint and proved to be important for some of the techniques that will be presented in the next section.

The second opportunity to avoid unnecessary work is x-kernel specific, but similar opportunities may arise in other implementations. In the x-kernel, message buffers are pre-allocated for interrupt handlers. For incoming packets, typical processing involves taking a message buffer from the pool of pre-allocated buffers, shepherding the message through protocol processing, and upon return, refreshing (reallocating) the message buffer so that it can be put back into the buffer pool. Originally, a message buffer was refreshed by first destroying it. Destroying a message buffer may or may not result in memory being freed, depending on whether there are other references to the same message. In most cases, incoming messages are consumed immediately. This means that by the time protocol processing is finished, there are no other reference to the message remaining and destroying the message buffer will always result in freeing exactly as much memory as is needed for the new buffer. Thus, in the vast majority of cases, it is possible to avoid a call to *free()* and *malloc()* by simply detecting this case short-circuiting the two calls.

2.2.3 Careful Inlining

Inlining is one of the most commonly applied optimizations. Fundamentally, it trades temporal locality for better code quality. In an environment where instruction-bandwidth is at a premium, this presents a non-trivial trade-off. For example, suppose there is a frequently called function occupying F i-cache blocks. If that function is inlined, the caller may grow only by $F' < F$ blocks because the compiler's optimizer can exploit call-site specific information and there

[1] In fact, some versions of TCP, for example the one that is shipped with NetBSD, compute 50 percent instead.

is no call overhead. Notice that even if the inlined code is not smaller by itself, inlining can increase the size of the caller by less than F blocks because function calls are normally optimization barriers.

A frequently called function is likely to reside in the i-cache when it is called. If it is inlined, however, the locality of reference may be so small that executing the inlined version will result in F' additional cache misses. In such a case, inlining is beneficial only if the inlined version is faster with F' cache misses than the genuine function with F cache hits. In practice, we found the following four cases to be safe for inlining:

1. The function has only one call-site.
2. The inlined version is not larger than the number of instructions that would be required in the call-site to perform the function call.
3. Call-site specific information simplifies the functions so much that it will execute faster even if the inlined version causes additional i-cache misses.
4. The inlined code itself is used frequently enough that i-cache misses can be amortized over multiple activations.

It is our contention that inlining is frequently misused to avoid replacement misses in the small associativity caches commonly found in high-performance RISC architectures. This may be tolerable when optimizing for throughput, but our experience clearly suggests that this is the wrong approach to improve latency. As discussed in the next section, replacement misses can instead be avoided through direct means.

While searching for good inlining opportunities, we found that in many cases, inlining would be beneficial when one or more of the actual arguments are constant (i.e., such that the third case in the above list applies). For example, the x-kernel hash-table manager supports a one-entry cache to exploit locality of network messages [Mog92]. Ideally, a hash-table lookup that results in a cache-hit should require only the few instructions needed to compare the key being looked up with the key of the cached entry. However, the hash-table manager supports a general interface that allows for unaligned keys and various key-sizes. This introduces the dilemma that the common usage pattern is simple enough to be inlined, but the general case is complicated enough to make inlining counter-productive.

Fortunately, GNU C [Sta92] supports a builtin (intrinsic) function that evaluates to TRUE if and only if the argument is a compile-time constant. This mechanism is powerful enough to construct a pre-processor macro that expands into an inline key-comparison if the alignment and size of the keys are appropriate, but results in a simple function call if inlining is not warranted. Obviously, this is an ad-hoc solution that is rather awkward to use. However, the concept of *conditional inlining* appears to be an elegant solution to achieving high performance without sacrificing modularity. On the Alpha architecture, we found that an inlined hash-table lookup with a cache-hit takes about three times fewer instructions than the general function. Moreover, because only the initial key-comparison is inlined, code expansion is minimal.

2.2.4 Fixing Machine Idiosyncrasies

The first two generations of Alpha implementations did not support byte and short-word (16-bit) sized memory-accesses. Consequently, changing `char` and `short` variables to word-sized integers typically results in a reduction of code-size. Normally, this reduction is not very significant, but for TCP, changing about a dozen declarations in the C structure that represents connection state resulted in a surprising code-size reduction. In fact, we found that this change was the one with the largest savings in instruction count.

Another machine peculiarity involves the interface to the network adaptor. The network driver communicates with the LANCE chip via a shared region in the main memory. This shared region holds receive and transmit frame buffers as well as descriptors for the frame buffers. The LANCE chip has a 16-bit bus interface, while the TURBOchannel to which it is connected is 32 bits wide. This has the unfortunate effect that shared memory is used sparsely—for descriptors, every 16 bits of shared memory are followed by a 16-bit gap. For buffers, 16 bytes of shared memory are followed by a 16 byte gap. To keep things simple, most LANCE drivers for TURBOchannel-based machines update descriptors in the shared memory by copying them first into dense memory, applying the necessary modifications, and then writing back the entire descriptor to sparse memory. Descriptors are ten bytes long. Every update therefore involves copying 20 bytes, even if only a single bit changes. In principle, there is no reason that this copying cannot be avoided: descriptors can be updated in the sparse memory directly. However, C has no concept of "sparse" or "dense" memory and coding this directly is at least error prone enough that, to the best of our knowledge, nobody has ever bothered to do so. Fortunately, the Universal Stub Compiler (USC) is ideally suited for this task—the layout of the descriptor can be described independently of the structure of the sparse memory space [OPM94]. USC can generate inlined functions that allow direct sparse memory access to any field in the descriptor. USC provides an efficient solution to the sparse-memory problem presented by the machine, and as a side-effect, the resulting code is much easier to understand and maintain.

2.2.5 Summary of Changes

Since we are interested in latency, it is useful to summarize the changes that affect code size. Table 1 gives a breakdown of the dynamic instruction count savings for the TCP/IP processing path. As the first row shows, the simple TCP change that replaced some byte and short variables by words achieved the biggest saving. The second rank is occupied by the change in the way messages are refreshed at the end of protocol processing. As the second row in the table shows, this resulted in a reduction of 208 instructions. Interestingly, this improvement does *not* affect end-to-end latency in the TCP/IP latency test. This is because the message is refreshed only after the current packet has been sent to the network adapter. The computation therefore is overlapped with communication. However, the change does reduce CPU utilization, which is also desirable.

Technique	Instructions saved
Change bytes and shorts to words in TCP state:	324
More efficiently refresh message after processing:	208
Use USC in LANCE to avoid descriptor copying:	171
Inlined hash-table cache test:	120
Various inlining:	119
Avoid integer division:	90
Other minor changes:	39
Total:	1071

Table 1: Dynamic Instruction Count Reductions

2.3 TCP/IP Performance Comparison

After applying the optimizations described above, we felt that the code was in good enough shape to serve as a base case for evaluating the techniques presented in the next section. The performance numbers in Table 2 indeed demonstrate significant improvements. End-to-end roundtrip latency was reduced by more than 26μs and almost 20% fewer instructions were executed during protocol processing. Notice that the number of cycles per instruction (CPI) does not change significantly.

	Original:	Improved:
Roundtrip latency [μs]:	377.7	351.0
Instructions executed:	5821	4750
Processing time [cycles]:	18941	15688
CPI:	3.26	3.30

Table 2: Performance Comparison of Original and Improved x-kernel TCP/IP Stack

To verify that the TCP/IP implementation is roughly equivalent to a production-quality implementation, we present a few performance numbers that compare the improved x-kernel implementation with the DEC Unix v3.2c implementation. These numbers are based on instruction traces that we collected.[2] This comparison also provides an interesting update of the CISC numbers presented in [CJRS89]. Specifically, we traced and measured the time it takes to perform IP and TCP processing for an incoming one byte TCP segment on a connection with bi-directional data flow. Notice that [CJRS89] focused on the case where data flows only in one direction (as is the case for an ftp transfer, for example). We feel using a bi-directional connection is more realistic since if data flows in only one direction, it is usually possible to batch it together and send reasonably large packets, and for large packets, processing time is dominated by data-dependent costs. In contrast, with a request-response style of communication, batching is not possible, and 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 hosts perform sender and receiver-related house-keeping when receiving a packet. In contrast, with a unidirectional connection a host performs either sender or receiver-related house-keeping but not both. Second, the DEC Unix TCP implementation uses header-prediction [CJRS89]. This is an optimization targeted at improving latency. However, all the header-prediction implementations we know of work only for uni-directional connections. The result is that rather than improving latency, header prediction slightly worsens latency on a connection with a bi-directional data flow. Fortunately, with less than a dozen additional instructions executed, the slow down is not very large.

Architecture:	**80386**	**Alpha**	
TCP/IP implementation:	**[CJRS89]:**	**DEC Unix v3.2c:**	**Improved x-kernel:**
Number of instruction executed...			
... in `ipintr`:	57	248	
... in `tcp_input`:	276	406	
... between IP input and TCP input:		262	437
... between TCP input and socket input:		1188	1004

Table 3: Comparison of TCP/IP Implementations

Table 3 displays the results. The first row shows the number of instructions executed while the processor executes in function `ipintr` (IP input processing). The second row displays the number of instructions executed in `tcp_input` *after* the TCP control block has been found (i.e., after function `in_pcblookup` returns). Since the x-kernel source code is organized differently, it is not possible to list corresponding number for the x-kernel.

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

The cost of 276 instructions for TCP input processing on the 80386 was arrived at by adding both the sender side and the receiver side costs, as well as the common path cost reported in [CJRS89] (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). First, we notice that IP processing in DEC Unix appears to be more than a factor of four longer than for the 80386. We believe this more 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 that depends heavily on implementation details (e.g., amount of inlining). Instead, we suggest to count the number of instructions required to complete a specific task; more on this below. For `tcp_input`, the DEC Unix trace is roughly 50% longer than the 80386 count. While somewhat on the large side, such a code inflation is not uncommon when converting CISC code to RISC code, especially considering that the traced Alpha code does have sub-word loads and stores available.

As alluded to before, a better performance measure is to count the total number of instructions executed to complete a particular task. This is what we did to compare the x-kernel and the DEC Unix implementations. In particular, 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, to establish TCP processing costs 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 were not available. As the third row in Table 3 illustrates, DEC Unix is doing a little better during IP processing, and the x-kernel implementation is better during TCP processing. Overall, the two traces have almost the same length (1450 versus 1441). However, the x-kernel code executes quite a bit faster as separate measurements yielded a CPI of 4.26 for DEC Unix and a CPI of 3.3 for the x-kernel. The important point is that the x-kernel implementations appears perfectly competitive with a common and reasonably well optimized implementation of TCP/IP.

3 Latency Reducing Techniques

This section describes three techniques that we have evaluated in an attempt to reduce protocol-processing latency. Unlike the techniques presented in the previous section, which had the effect of reducing the number of instructions executed along the protocol stack, these techniques are primarily targetted at the cost—in terms of memory cycles—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_zero     r0, good_day
    panic("Hit a snag...");               load_address     a0, "Hit a snag..."
}                                         call             panic
printf("Good day...");            good_day:
   :                                      load_address     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 i-cache 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 mainline 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 [Hei94, PH90]. Unfortunately, profile-based optimizers suffer from the problem of being aggressive rather than conservative: any code that is not covered by the collected profile will be outlined. This is aggravated by the fact that it is difficult to back-map the optimizer's changes to the source code, thereby making it difficult to verify that a collected profile is indeed (sufficiently) exhaustive. Also, relatively simple changes to the source code may require collecting a new profile all over again. This may be acceptable for user-level code, but is certainly less than ideal for system software, such as networking code.

In contrast, 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, we have measured system software that contains up to 50% error checking/handling code; just outlining in these obvious cases can result in dramatic code-density improvements. 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:

```
                                            :
   :                                      load             r0, (bad_case)
if (bad_case PREDICT_FALSE) {             jump_if_not_zero r0, bad_day
    panic("Hit a snag...");               load_address     a0, "Good day..."
}                                         call             printf
printf("Good day...");            continue:
   :                                        :
                                          return_from_function
                                  bad_day:
                                          load_address     a0, "Hit a snag..."
                                          call             panic
                                          jump             continue
```

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 mechanisms.
2. Initialization code. Code 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 technique: cloning.

3.2 Cloning

Cloning involves making a copy of a function. The cloned copy can be relocated to a more appropriate address and/or optimized for a particular use of that function. For example, if the TCP/IP path is executed frequently, it may be desirable to pack the involved functions as tightly as possible. It is usually not necessary to clone outlined code. 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 [Mas92]. 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.

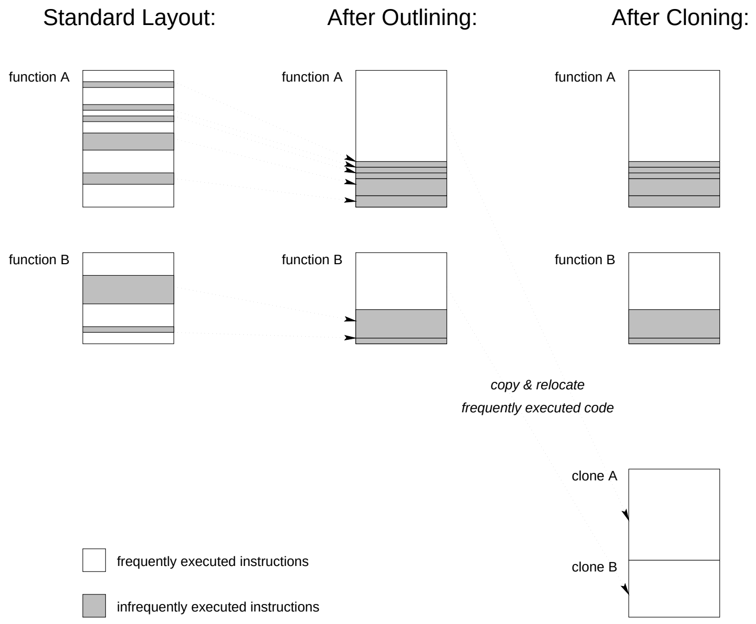

Figure 2: Effects of Outlining and Cloning

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 effect 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. In this particular example, we assume the clones can share outlined code with the original functions.

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 [McF89]. Consequently, by choosing appropriate addresses, it is possible to optimize i-cache behavior for a given path. The cost is that occasionally it is 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 precious i-cache 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 measurement, 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 per path-invocation and *library* functions that are executed multiple times per path. 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 should be placed in the order in which they are called. 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* [PH90].

Establishing the performance advantage of this layout scheme relative to the micro-positioning approach is difficult since small changes to the heuristics of the latter approach results in large performance changes. The micro-positioning approach usually performs somewhat worse than a bipartite layout and sometimes almost equally well, but never better. One may wonder why this is so. It is impossible to make any definite conclusions without detailed simulations of the CPU and the memory system, 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 in the micro-positioned code that causes the overall slowdown. Second, the gaps introduced by the micro-positioning approach do cost extra i-cache bandwidth. We have not found a single instance where aligning function entry-points or similar gap-introducing techniques would have improved end-to-end latency. This is in stark contrast with the findings published in [GC90], 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 i-cache 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 are cheaper than four or five additional cold misses introduced by micro-positioning.

Despite the unexpected outcome, this 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, unfortunately, a recurrent theme for cache-oriented optimizations—the best solution when the problem fits into the cache is radically different from the best solution when the cache is a scarce resource.

3.3 Path-Inlining

The third latency reducing technique is path-inlining. This is an agressive form of inlining where the entire latency-sensitive path of execution is inlined into a single function. As explained in Section 2.2.3, this is warranted only if the inlined code is executed very frequently. Obviously, functions that are used repeatedly still should not be inlined since it is better to preserve the locality of reference and also since otherwise the size of the resulting path could suffer from exponential growth.

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 avoid such useless call overheads.

Path-inlining is relative easy as long as no indirect function calls are involved. This is usually the case for the outbound side of network processing, although routing can complicate this case. On the inbound side, 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 higher-level protocol. This higher-level protocol identifier is then mapped into the address of the function that implements the appropriate protocol processing. In short, inbound processing is full of indirect function calls. To make path-inlining work for this important case, it is necessary to *assume* that a packet will follow a given path, generate path-inlined code for that assumed path, and then at run-time, establish that an incoming packet really will follow the assumed path. The last part requires employing a packet classifier [BGP+94, MJ93, YBMM93, EKJ95].

While path-inlining is easy in principle once indirect function calls have been taken care of, 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.

4 Evaluation

This section evaluates the techniques presented in the previous section. If first describes the experimental methodology and summarizes the test cases we measured, and then reports both end-to-end measurements and the results of a detailed analysis.

4.1 Experimental Setup

The hardware consists of two DEC 3000/600 workstations connected over an isolated Ethernet. These workstations use the 21064 Alpha CPU running at 175MHz [Sit92]. 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. The d-cache is write-through and allocates on read misses only, while the b-cache is write-back and allocates on either miss type. To improve write performance, the CPU uses a 4-deep write buffer. Each write buffer can hold one cache block. The CPU is a 64-bit wide, super-scalar design that can issue up to two instructions per cycle. The memory system interface is 128 bits wide.

To achieve maximum control over the experiments, the software was implemented in a minimal stand-alone version of the x-kernel [HP91]. 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. The protocol stack uses the protocols described in section 2. All code was compiled using a version of gcc 2.6.0 that supports outlining.

The metrics that are ultimately of most interest are end-to-end latency, throughput, and CPU utilization. End-to-end timings were measured with a timer running at 1024Hz, yielding roughly a 1ms resolution. We verified that none of the techniques negatively affected throughput, and in fact, they slightly improved throughput performance. For the sake of brevity, we do not report any of the throughput results in this paper. To measure CPU utilization and to gain a better understanding of the effects of each technique, we also collected execution traces and measured the execution time of the traced code using the CPU's builtin cycle counter.

4.2 Test Cases

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. Measuring all possible combinations of the techniques would have been impractical, so we focused on the following six cases; we supply additional data where appropriate.

STD: This configuration includes none of the optimizations described in Section 3, but does include the 64-bit specific improvements outlined in Section 2.

OUT: Like STD, but includes outlining.

CLO: Like OUT, but includes cloning (using the bipartite layout).

BAD: Like CLO, but cloning has been used to artificially *worsen* the i-cache behavior.

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.

It is important to note that path-inlining requires running a packet classifier on incoming packets since the optimized code is no longer general enough to handle all possible packets. Currently, the best packet classifiers add an overhead of about $1 - 4\mu$s per packet on the tested hardware platform [BGP+94, EKJ95]. However, to separate packet classification performance from the techniques under study here, no packet classifier was used. In this sense, the PIN and ALL measurements should be interpreted as the performance obtained with a zero-overhead packet classifier. Using path-inlining without a packet classifier works fine in the test environment since we used an isolated Ethernet network that had no traffic other than the one generated by the test programs.

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. In both cases, this results in 64-byte frames on the wire since that is the minimum allowed size for Ethernet. The end-to-end latency reported is the average time it took to complete one roundtrip in a test involving 100,000 roundtrips. 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 the server and client sides 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 a meaningful analysis of client-performance can be performed.

4.3 End-to-End Results

Table 4 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.

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

Table 4: End-to-end Roundtrip Latency

Not surprisingly, version BAD performs worst. With almost 500μs per roundtrip, it is over $173\mu s$ slower than the version CLO, which corresponds to a slowdown of more than 53%. As alluded to before, the two versions are identical except for their cache behavior. On the client side of TCP/IP, version BAD suffers an additional 217 i-cache and 110 b-cache misses compared to version CLO, while the RPC client has an additional 233 i-cache and 14 b-cache misses. Clearly, i-cache effects can have a profound effect on end-to-end latency.

As row STD shows, however, the standard x-kernel version of the protocol stacks has much better cache behavior than version BAD. That version is slower by only 12.9% for TCP/IP and by 9.2% for RPC. The reason that STD performs relatively well is two-fold. First, it appears to be the case that the function usage pattern is such that laying the functions out in the address space in a random manner yields an average performance closer to the best case than to the worst case. Second, earlier x-kernel experiences with direct-mapped caches have led to attempts to improve cache performance by changing the link-order of the object files that form the kernel. Because of this manual tuning, the STD version does not suffer from cache thrashing to start with. Keep in mind, however, that case BAD is possible in practice; part of the motivation for this work is to provide automatic tools that avoid such pessimal layouts.

Row OUT indicates that outlining works quite well for TCP/IP—it reduces roundtrip time by about 15μs when compared to STD. Since both the client and the server uses 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 not surprising. 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), while RPC consists of many small functions and exceptional events are often handled by calling another function. In a sense, the RPC code is already structured in a way that handles exceptional events outside the main line of the code. Nevertheless, outlining does improve performance significantly in both cases.

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 would lead 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 we 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, based on end-to-end latency, separating library code from the path-inlined code does not appear to be very important.

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 adaptors have much lower latency than the LANCE Ethernet adaptor present in the DEC 3000 system [AMD]. 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 Ethernet ($10 \cdot 10^6$bps), 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 [TL93]. 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 5 revisits the end-to-end latency numbers, adjusted to factor out the overhead imposed by the controller. While the controller will obviously add some latency, one should expect RTTs on the order of 50μs[3] rather than the 210μs measured on our experimental platform.

	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 5: End-to-end Roundtrip Latency Adjusted for Network Controller

4.4 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 at once. 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 we cannot control these effects, we can measure many of them. 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 are not complete since the tracing method did not allow us to trace interrupt handling; they cover all protocol processing code except for the network driver interrupt handling and context switching.

4.4.1 Cache Statistics

With the execution traces and a simulator of the DEC 3000/600 memory hierarchy it is possible to compute the cache statistics presented in Table 6. It lists the i-cache, d-cache, and b-cache performance as the number of misses to the cache (column **Miss**), the total number of accesses to the cache (column **Acc**), and the number of replacement misses (column **Repl**). Notice that the middle three columns combine the d-cache and write-buffer performance since the d-

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

cache is used on the read path and the write-buffer on the write path only. The write-buffer performs write-merging, so a merged write is counted like a cache-hit, whereas a write that caused a write to the b-cache is counted as a cache-miss.

		i-cache			d-cache/wr-buffer			b-cache		
		Miss	Acc.	Repl	Miss	Acc	Repl	Miss	Acc	Repl
TCP/IP	BAD	700	4718	224	459	1862	31	863	1390	110
	STD	586	4750	72	492	1845	56	800	1286	0
	OUT	547	4728	69	462	1841	40	731	1183	0
	CLO	483	4684	27	455	1862	34	678	1074	0
	PIN	484	4245	66	406	1668	27	630	1015	0
	ALL	414	4215	10	401	1682	28	596	913	0
RPC	BAD	721	4253	176	556	1663	19	995	1544	14
	STD	590	4291	31	547	1635	14	1004	1379	0
	OUT	542	4257	26	556	1629	19	951	1313	0
	CLO	488	4227	7	547	1664	13	845	1213	0
	PIN	402	3471	14	453	1310	19	694	972	0
	ALL	374	3468	0	450	1330	13	662	931	0

Table 6: Cache Performance. **Miss**: number of accesses that missed in the cache. **Acc**: Total number of cache accesses. **Repl**: Number of replacement misses.

The rightmost column in the table shows that, except for the BAD versions, none of the kernels cause replacement misses in the b-cache. Since the entire kernel is small enough to fit into the b-cache, this means that all code executes out of the b-cache unless there are conflicts with data accesses performed outside of the traced code.

A more important observation is that the cache simulations confirm that cloning with a bipartite layout does indeed help avoid i-cache replacement misses. For example, applying cloning to version OUT reduced the number of i-cache replacement misses in the TCP/IP stack from 69 to 27. Interestingly, path-inlining alone does not get rid of many replacement misses. The table shows that the PIN version still suffers from 66 such misses. This is because there is nothing that prevents library code from clashing with path code. The RPC case is analogous to TCP/IP. In fact, the savings are even larger: compared to version OUT, cloning alone reduces the number of replacement misses by a factor of 3.7, and together with path-inlining, not a single replacement miss remains.

4.4.2 Processing Time Measurements

We now consider the execution time of the traced code. Together with the length of the instruction trace, we can determine the average number of *cycles per instruction* (CPI) by simply multiplying the execution time by the CPU clock frequency (175MHz) and dividing the product by the trace length. Furthermore, feeding the trace into a CPU simulator, we can compute the CPI of the traced code assuming a perfect memory system. This is usually referred to as the *instruction CPI* (iCPI). Subtracting the iCPI from the CPI yields the *memory CPI* (mCPI)—the average number of cycles that an instruction stalls waiting for the memory system. This data is shown in Table 7. Column 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 labelled **Length** gives the trace length as an instruction count. Columns **mCPI** and **iCPI** are the memory and instruction CPI values, respectively.

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

Table 7: Protocol Processing Costs

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 identical. Since the CPU simulator adds a fixed penalty for each taken branch, the decrease in the iCPI as we go from the outlined versions to the standard version can be attributed to the reduction in taken branches. Interestingly, outlining improves iCPI by almost exactly 0.1 cycles for both protocol stacks. This is a suprisingly 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. It is, however, possible that a compiler with a better instruction scheduler would have achieved a larger iCPI reduction.

As the mCPI columns show, the CPU spends well above 1 cycle per instruction waiting for memory (on average). The only exception is version ALL in the RPC stack, where mCPI suddenly drops to 0.81; we return to this later. Comparing the mCPI values for the various versions, we find that the proposed techniques are rather effective. Both protocol stacks achieve a reduction 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 since version ALL has the smallest mCPI of all versions. 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.

4.4.3 Performance Improvement Comparison

We are now in a position where we can compare the end-to-end results with the processing execution times and the trace-based cache simulation results. First, we would like to verify that the outlining and cloning improvements are really primarily due to i-cache rather than d-cache effects. The fact that the mCPI values are much greater than 0 indicates that the memory system is the bottleneck. As all versions run out of the b-cache (except for the BAD versions), the processing time improvement is dominated by changes in the number of b-cache accesses (column ΔN_b in Table 8). Thus, we would like to know the percentage I of b-cache access reduction due to the i-cache, as opposed to the d-cache/write-buffer.[4] If the number of b-cache accesses is reduced purely due to the i-cache, the percentage would be

[4] In computing this percentage, it is important to keep in mind that the number of b-cache accesses due to the i-cache is given by the number of b-cache accesses minus the number d-cache/write-buffer misses. This is typically greater than the number of i-cache misses since a miss may lead to another i-cache block being prefetched, thus resulting in two b-cache accesses.

100%. If the reduction is purely due to the d-cache/write-buffer, it would be 0%. A value greater than 100% indicates that d-cache/write-buffer performance got worse, but that the i-cache was able to compensate for those losses so that, overall, the number of b-cache access was still reduced.

Table 8 lists this percentage for both the TCP/IP and RPC protocol stack in the respective column labeled I. Notice that in all but one case more than 90% of the b-cache access reductions obtained through outlining and cloning are due to the i-cache. The exception is where outlining is applied to the standard x-kernel version of the TCP/IP stack. As shown in row STD→OUT, the i-cache can take credit for only 71% of the reduction in b-cache accesses, but in all other cases, d-cache effects did not lead to significantly overestimating the benefit of a technique.

	TCP/IP					**RPC**				
	I [%]	ΔT_e [μs]	ΔT_p [μs]	ΔN_b [1]	ΔN_m [1]	I [%]	ΔT_e [μs]	ΔT_p [μs]	ΔN_b [1]	ΔN_m [1]
BAD→CLO	97	86.7	89.8	316	110	99	74.0	83.2	331	14
STD→OUT	114	7.4	5.5	103	0	71	4.6	4.1	66	0
OUT→CLO	91	5.3	6.9	109	0	94	11.5	10.0	100	0
OUT→ PIN	70	9.5	14.2	168	0	67	27.3	23.3	341	0
PIN→ALL	93	3.2	3.8	102	0	95	1.8	8.5	41	0

Table 8: Comparison of Latency Improvement

Next, we can compare the end-to-end latency improvements with the improvements in processing time. This is shown in columns ΔT_e and ΔT_p in Table 8. The data shows that the improvements are generally consistent with each other. The end-to-end improvement may be bigger than the processing time improvement if portions of the untraced code executed faster as well. The converse can occur if, for example, most of the improvement occurs in a section of the code whose execution overlaps network communication. The only place where these figures deviate significantly is in the RPC case, when going from the path-inlined version (PIN) to the version that was also cloned (ALL). In this case, the processing time improvement is 8.5μs, but the observed end-to-end improvement is only 1.8μs. While attempting to resolve this discrepancy, we found that only slight changes to the code led to processing time improvements much closer to the observed end-to-end numbers. We suspect that through bad luck, most of the untraced code collided in one part of the i-cache, leaving most of the traced code cached over multiple activations. Thus, the traced code executed almost entirely out of the i-cache, whereas the untraced code experienced many replacement misses. Unfortunately, we cannot verify this hypothesis since none of the available measurement techniques are completely free of intrusion. To be on the safe side, the discussions in this paper consider the end-to-end results as the relevant ones.

Finally, we can cross check whether the time improvements are consistent with the reductions in the number of b-cache accesses. If we divide the processing time improvements (ΔT_p) in the second through fifth row by the difference in the number of b-cache accesses (ΔN_b) we get an average b-cache latency in the range from 5.6 to to 17.5 cycles. (This ignores the RPC improvement for the PIN→ALL case, and uses a conversion factor of 175cycles/μs.) These values appear reasonable considering that a b-cache access takes 10 cycles to complete. It would be unrealistic to expect exactly 10 cycles per miss since this simple model ignores many of the finer aspects of the DEC 3000's memory system. Also, we cannot apply the same reasoning to the BAD→CLO improvements since we did not measure how many of the b-cache blocks remain cached across multiple invocations of the path. Nevertheless, the table includes the data for the sake of completeness. Notice that the PIN→ALL change in the RPC stack yields an 8.5μs processing time improvement, but the difference in the number of b-cache accesses is only 41. Clearly, the improvement cannot be due to the decrease in the number of b-cache accesses since that would mean that each access cost on the order of 36 cycles—almost four times the theoretical latency. This supports the hypothesis that the processing time improvement is due to effects outside of the traced code.

4.4.4 Outlining Effectiveness

There are several ways to evaluate the effectiveness of outlining. For example, Table 7 shows that outlining reduces processing times by about 5% compared to the standard version. Another measure is the percentage of instructions that remain unused in each cache block. Those instructions take up i-cache bandwidth without ever being used by the latency-critical path of execution. Table 9 indicates that outlining reduces the amount of wasted i-cache bandwidth significantly. Both the TCP/IP stack and the RPC stack originally left about 21%, or 1.8 instructions out of 8, unused. With outlining, only about 1.3 instructions per block are not used. 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 9: Outlining Effectiveness

Outlining is even more dramatic when measuring the amount of code that was outlined. In Table 9, the columns labeled **Size** 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. Similarly, in the RPC stack 28% of the 5085 instructions could be outlined. Thus, we consider outlining a useful technique not only because of its direct benefits, but primarily as a means to greatly improve cloning. As cloning works at the function level, minimizing the size of the main-line code—which is the only part that is cloned—improves flexibility and increases the likelihood that the entire path will fit into the cache.

5 Concluding Remarks

Network designers have known for a long time that memory bandwidth plays a critical role in end-to-end network throughput rates. This paper argues that memory bandwidth also has a measurable impact on protocol latency, and it describes three techniques that can be applied to network code to improve this situation. These techniques are primarily targetted at improving the number of memory cycles required by each instruction; we also described a collection of techniques that can be used to simply reduce the number of instructions required for each packet.

Beyond this basic result, there are two important things to take away from this work. First, even though case BAD reported in the results corresponds to a pessimally configured system, sub-optimal configurations are possible in practice. For example, we measured the mCPI of the DEC Unix TCP/IP stack to be 2.3, which is significantly worse than the 1.17 mCPI measured in our optimally configured system. One contribution of this work is a set of compiler-based techniques that can be applied to any system to ensure that such bad cases do not happen. If nothing else, these techniques improve the predictability of the system. Second, the impact of mCPI reducing techniques is becoming increasingly important as the gap between processor and memory speeds widen. For example, this research was conducted on a 175MHz Alpha-based processor with a 100MB/s memory system. We now also have in our lab a low-cost 266MHz processor with a 66MB/s memory system.

References

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

[BGP+94] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. PathFinder: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, 1994.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overheads. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall 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, October 1991.

[DEC93] DEC. *TURBOchannel: Hardware Specification*. Digital Equipment Corp., 1993. Order number EK-369AA-OD-007B.

[EKJ95] Dawson R. Engler, Frans Kaashoek, and James O'Toole 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.

[GC90] Rajiv Gupta and Chi-Hung Chi. Improving instruction cache behavior by reducing cache pollution. In *Proceedings Supercomputing '90*, pages 82–91. IEEE, 1990.

[Hei94] R. R. Heisch. Trace-directed program restructuring for AIX executables. *IBM Journal of Research and Development*, 38(9):595–603, September 1994.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[Jac93] Van Jacobson. A high performance TCP/IP implementation. Presentation at the NRI Gigabit TCP Workshop, March 18th–19th 1993.

[KP93] Jonathan Kay and Joseph 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, October 1993. ACM.

[Mas92] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY 10027, September 1992.

[McF89] Scott McFarling. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, April 1989.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX Conference*, San Diego, CA, January 1993. USENIX.

[Mog92] Jeffrey C. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.

[OP92] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[OPM94] 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, August 31st – September 2nd 1994.

[PH90] 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.

[Pos81a] J. Postel. RFC-791: Internet Protocol. Available via ftp from ftp.nisc.sri.com, September 1981.

[Pos81b] J. Postel. RFC-793: Transmission Control Protocol. Available via ftp from ftp.nisc.sri.com, September 1981.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Burlington, Massachusetts, 1992. Order number EY-L520E-DP.

[Sta92] Richard M. Stallman. *Using and Porting GNU CC*, 1992. Manuscript provided by the Free Software Foundation to document `gcc`.

[TL93] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[YBMM93] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. July 1993.