Protocol Latency: MIPS and Reality

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

TR 95-02

Abstract

This paper describes several techniques designed to improve protocol latency, and reports on their effectiveness when measured on a modern RISC processor—the DEC Alpha. We found that memory bandwidth—which has long been known to dominate network throughput—is also a key factor in protocol latency. The techniques are designed to increase the effectiveness of the instruction-cache and result in reduced processor stall rates.

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 [TL93], and for this reason, researchers have analyzed the latency characteristics of common network protocols like TCP/IP [KP93, CJRS89, Jac93]. These studies have shown that, despite the rich functionality offered by TCP/IP, the processing overheads are actually quite low.

This paper revisits the issue of protocol latency. Our goal is not to optimize a particular protocol stack (we also study TCP/IP because of its rich functionality), but rather, to understand the fundamental limitations on processing overhead. In doing so, this paper goes beyond the earlier work in three important ways. First, it studies protocol latency on a modern RISC architecture—the 64-bit DEC Alpha. 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. Second, the paper 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. Third, the paper proposes 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 rather the number of *cycles* that each instruction takes.

It should be clear from these three points that instruction bandwidth is a central focus of this paper. In fact, the experimental results show that instruction-bandwidth is the bottleneck that keeps the CPU idle almost 70% of the time, even when the entire test program fits into the machine's second-level cache. Thus, the proposed techniques were primarily chosen based on their potential to improve the rate at which instructions are executed. Despite this bias, however, the techniques also affect instruction count. This allows us to gain insight into which dimension—number of instructions or cycles per instruction (CPI)—has the greater potential to decrease latency.

The paper is organized as follows. Section 2 describes several latency improvement techniques. We implemented and evaluated all of the techniques in a well-controlled experimental environment, with the results reported in Section 3. Finally, Section 4 discusses future work and gives some concluding remarks.

2 Techniques

It is well-known that modern networking code seldom suffers from a single, dominant latency bottleneck. Rather, latency accumulates through dozens of small overheads that, in isolation, are seemingly insignificant. This implies that latency cannot be improved significantly by a single stroke of genius as is often possible when maximizing throughput.

This section presents and discusses a number of techniques that we have evaluated in an attempt to reduce protocol-processing latency in the *x*-kernel. We chose a TCP/IP stack as our test-case because it is sufficiently complex to be interesting, and it facilitates a comparison with other latency-oriented optimizations. Figure 1 shows the protocol-stack that was used throughout this work. TCPTEST is a simple, ping-pong-style test program. TCP and IP are the *x*-kernel implementations of the corresponding Internet protocols. VNET is a virtual protocol [OP92] that routes outgoing messages to the appropriate network adaptor. (In other implementations, VNET's functionality is part of IP). ETH is the device-independent half of the Ethernet driver, while LANCE is a device driver for DEC 3000/600 workstations.

The latency-improvement techniques can be broadly classified into two groups: infrastructure-specific and protocol-specific optimizations. Changes to the infrastructure are of more global value because they will apply independent of the particular protocol stack being used. Nevertheless, we tested several protocol-specific changes in order to get an idea of the potential for latency improvements in that group. We first discuss infrastructure-specific optimizations; the

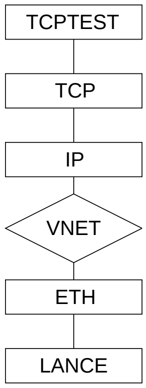

Figure 1: Test Protocol Stack

TCP/IP-specific techniques are described in the last subsection.

2.1 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 becomes a non-trivial trade-off. For example, suppose there is a frequently called function occupying F instruction-cache (i-cache) blocks. If that function is inlined, it may occupy only $F' < F$ blocks because the compiler's optimizer can exploit call-site specific information and there is no call overhead. A frequently called function is likely to reside in the i-cache when it is called. On the other hand, if it is inlined, locality of reference may be so small that executing the inlined version will result in F' cache misses. Thus, 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. 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 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 on machines with low-associativity caches. This may be tolerable when optimizing for throughput, but our experience clearly suggests that this is the wrong approach to improve latency. As discussed below, replacement misses should 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 makes extensive use of a hash-table manager [HMPT89, MO93]. The hash-table supports a one-entry cache to exploit locality of networking [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 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:

```
    :
if (bad_case) {
    panic("Hit a snag...");
}
printf("Good day...");
    :
```

```
            :
          load          r0, (bad_case)
          jump_if_zero  r0, good_day
          load_address  a0, "Hit a snag..."
          call          panic
good_day:
          load_address  a0, "Good day..."
          call          printf
            :
```

The above machine code is suboptimal for two reasons: (1) it requires a taken jump to skip around 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 (i.e., by outlining error handling). 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, therefore, 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 below on the left will be translated to the machine code shown on the right:

```
   :
if (bad_case PREDICT_FALSE) {
    panic("Hit a snag...");
}
printf("Good day...");
   :
```

```
          :
        load            r0, (bad_case)
        jump_if_not_zero r0, bad_day
        load_address    a0, "Good day..."
        call            printf
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 time 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 hardly makes a difference in end-to-end latency. In early measurements, roundtrip time decreased by about 2 μs. However, the code density improvements that it achieves are essential to the effectiveness of the next technique, namely cloning.

2.3 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 session is established, most session state will remain constant and can be used to partially evaluate the cloned function.

This achieves similar benefits as code synthesis [Mas92]. Obviously, just as for inlining, cloning is at odds with locality of reference. Cloning at session creation time will lead to one cloned copy per session, 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.

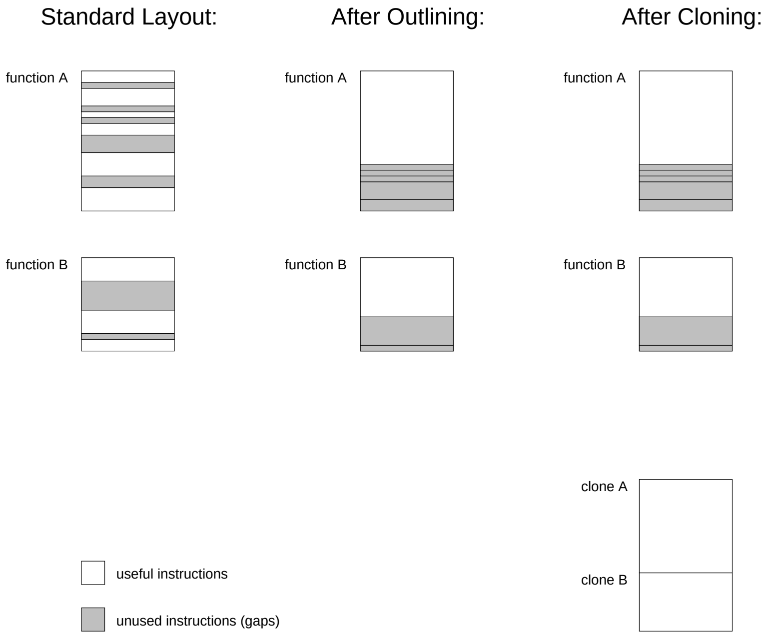

Figure 2: Effects of Outlining and Cloning

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 at the protocol stack level (not at the session level) 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 once they execute, as there is no temporal locality unless the entire path fits into the i-cache, which is not the case with TCP/IP. 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 the are called. Sequential layout maximizes the effectiveness of prefetching hardware that may be present; it is most likely that memory systems are optimized for an access pattern that is sequentially increasing. 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].

One may wonder why the bipartite layout consistently outperformed the micro-positioning approach. It is difficult 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 (i.e., where it incurs the minimum number of replacement misses). 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. 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 the 36 replacement misses were cheaper than the four or five cold misses introduced by the micro-positioning approach.

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.

2.4 Last Call Optimization

Operating systems in general, and networking code in particular, often suffer from deeply nested calling sequences. Frequently, a function's last action is to call another function. If this is the case, and if none of the local state is accessible to the called function, it is possible to deallocate the caller's stack frame *before* performing the last call. Ideally, in a calling sequence that is nested N deep, the amount of stack space used would be just the *maximum* frame size among the active functions rather than the *sum* of the frame sizes. This last call optimization is an old technique (e.g., [Er83]) and appears to be popular in compilers for functional languages, but not for imperative languages such as C. The main problem with imperative languages is that it is difficult to decide whether the callee has access to the caller's state or not. One simple solution is to apply the last call optimization only if none of the local (automatic) variables have had their addresses taken. This is overly conservative, but is easy to compute and still able to detect many last call opportunities.

A trial-implementation of the last call optimization in GCC showed improvements of up to 30%. We experimented with it in the TCP/IP stack but found no measurable difference. The reason is that many last call opportunities are destroyed by the *x*-kernel's reference counting scheme. We are considering fixing this problem and expect that this would roughly double the number of times the optimization can be applied. More importantly, however, our test programs did not stress stack usage. As we will explain below, the tests always use the same area of memory for execution stacks. As a result, very few stack accesses miss in the data-cache (d-cache). We believe that under a different scenario—for example, with many stacks concurrently in use and with stacks in virtual memory so that an access can result in a TLB miss and/or a page fault—the last call optimization could make a positive difference.

2.5 Other Optimizations

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 released to the pool of pre-allocated buffers. Traditionally, a message buffer was refreshed by destroying it first and then followed by allocating the required memory. Destroying a message buffer may or may not result in memory being freed, depending on whether there are other references to the same message. However, in most cases, incoming messages are consumed immediately. That is, by the time protocol processing has completed, there are no other reference to the message anymore and destroying the message buffer will result in freeing exactly as many bytes as are needed to replenish the buffer again. Thus, in the vast majority of cases, it is possible to avoid a call to *free()* and *malloc()* simply by detecting this case and taking appropriate actions. As will be shown in Section 3.4, this simple change alone reduced protocol processing overheads by almost 5% (measured in the number of instructions executed during processing).

TCP timer management requires efficient support for visiting all open connections. In the *x*-kernel, this can be done easily by traversing the hash-table that is used for demultiplexing IP datagrams to TCP sessions. 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) will be empty. Not only is this slow, but it also results in a needlessly large (d-cache) footprint. To alleviate this situation, we modified the hash-table to contain a list of non-empty hash-table buckets. Unfortunately, managing this list imposed unacceptably high overheads because removing an element from the list would either require a list-traversal or a doubly-linked list. This was solved by evaluating list-removals lazily: when a hash-bucket becomes non-empty, it is added to the linked list, but when it becomes empty again, it is just left on the list. The linked list is cleaned up only when it is traversed because at that point removing empty buckets is trivial.

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. It should be noted that TCP timers run at a frequency low enough that will barely affect TCP/IP roundtrip time. However, keeping timer management from interfering with the d-cache and reducing CPU utilization are nevertheless desirable goals.

Our experiments clearly showed the importance of an end-to-end approach to latency improvements. For example, thread and memory management are not usually considered to be part of the *x*-kernel proper. However, rather significant latency improvements were achieved by optimizing these parts. In the case of the thread manager, we made extensive use of continuations [DBRD91]. We also converted stacks to first-class objects, rather than being statically attached to the thread context. This, together with a LIFO management of stacks has the effect that latency sensitive path invocations will always use the same stack memory area, thereby increasing the chance that the memory will be cached already. Continuations also reduce the amount of state that needs to be saved or loaded in a context switch. Similarly, we found the memory manager's *free()* function to have potential for latency reduction. Adding sentinels to the free-memory list greatly simplified the core loop in *free()* and resulted in a good end-to-end latency improvement. During early testing, we found the changes to thread and memory management improved TCP/IP roundtrip latency by about 10 μs.

2.6 Protocol-Specific Optimizations

Compared to protocol infrastructure, relatively little time was spent optimizing actual protocol code. One reason for this is that protocol-specific optimizations will generally pay off for that particular protocol only. We also found the *x*-kernel protocol code to be of relative high-quality already, so there are not that many optimization opportunities anyway. Except for a trivial change in IP, involving the inlining of a function, only TCP and LANCE were modified.

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 adding a simple if-statement, it is possible to detect this case and 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 checking for 33 percent instead, it is possible to replace the integer multiplication and division by a simple shift and add. This change should not affect the operation of TCP noticeably[1]. These two changes had a relatively small effect on end-to-end latency. Roundtrip time decreased by about 3.8 μs. More importantly, removing the large division routine from the critical path reduces the cache-footprint and increases the effectiveness of cloning.

As described in Section 2.3, the best performance was achieved with a bipartite layout. Usually, it is straightforward to classify a function either as a path or a library function. However, traces indicated that TCP's send function (called *tcp_output()* in BSD-derived implementations) really consists of two parts: the first half checks whether it is necessary to send a packet and the second half performs the actual sending. During protocol processing, it is frequently necessary to check whether a packet should be sent, but there is only one instance where a packet actually is sent. Thus, we split *tcp_output()* into two parts: *tcp_output()* and *tcp_send()*. The former has the same semantics as the original function, but when it is necessary to send a packet, it calls *tcp_send()* instead of directly doing so. This allows classifying *tcp_output()* as a library function and *tcp_send()* as a path function.

[1] Some versions of TCP, for example the one that is shipped with NetBSD, check for 50 percent instead.

A final modification to TCP was to change the type of some of TCP's session state variables. The Alpha architecture has no hardware support for byte and short-word (16-bit) memory-accesses. Consequently, changing `char` and `short` declarations to `int` declarations typically results in a reduction of code-size. Normally, this reduction is not very significant, but for TCP, changing about a dozen declarations in a single C structure resulted in a surprising code-size reduction. In fact, this simple change turned out to be the one with the largest savings in instruction count, as will be shown in Section 3.4. Of course, changing variable types has a potential for affecting correctness of a program. For TCP, this was not a problem, though.

The LANCE driver communicates with the LANCE chip on the network adapter via a shared region in the main memory. This shared region holds receive and transmit frame buffers as well as descriptors used to manage these frame buffers. The LANCE chip has a 16-bit bus interface, while the TURBOchannel to which it is connected is 32 bits wide. The effect of this is that the shared memory is used sparsely—for descriptors, every 16 bits of shared memory are followed by a 16-bit gap. For simplicity, most LANCE drivers for DEC machines therefore update descriptors in the shared memory by copying them first into dense memory, applying the necessary modifications, and the writing the entire descriptor back 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. This reduced latency roughly by 8 μs. As a side-effect, the resulting code is also much easier to understand and maintain.

3 Evaluation

This section evaluates the techniques discussed in the previous section. The experimental environment and the measurement methodology are first described, and then we present and discuss the test cases and the results obtained.

3.1 Experimental Setup

The hardware consists of two DEC 3000/600 workstations connected over a private 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 32B large cache-blocks. For the i-cache, this implies that a cache block holds 8 instructions each. 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. 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-bit wide.

To achieve maximum control over the experiments, the software was implemented in a minimal stand-alone version of the *x*-kernel. 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 (subject to the condition that there are no b-cache conflicts). The protocol stack uses the protocols distributed with *x*-kernel version 3.2. Only TCP was modified to fit better into the *x*-kernel paradigm, but these changes affected mostly connection establishment and teardown and were not performance oriented.

The metrics that are ultimately of most interest are, of course, end-to-end latency and throughput. Both of these were measured with a timer running at 1024Hz, yielding roughly a 1ms resolution. To gain a better understanding of system behavior, we also employed a performance monitor built into the 21064 CPU and collected execution traces. The trace collection facility is simplistic and amounts to little more than automatic single-stepping through the program. After each step, the current program counter value, the instruction at that address, and the values of the registers that may be used by that instruction are recorded. This information is usually packed into four 64-bit words, but requires a fifth word when a long jump is encountered. This method is memory intensive and slow, as it involves a context-switch per instruction executed. However, the traces we are interested in are all less than 10,000 instructions long and slowdown is not a problem in practice.

Event Source	Description
Total cycles:	Counts CPU clock ticks (i.e., clocks at 175MHz).
Total issues:	Counts number of instruction that issued for execution.
Total non-issues:	Counts number of times an instruction cannot be issued.
Pipeline dry:	Counts cycles during which nothing issued due to an i-cache fill, branch misprediction, branch delay-slot or pipeline drain for exception.
Pipeline frozen:	Counts cycles during which nothing issued because of a resource conflict.
Dual issue:	Counts cycles that can issue two instructions at once (dual issue).
Integer instruction:	Counts number of integer operate instructions executed.
Load instruction:	Counts number of load instruction executed.
Store instruction:	Counts number of store instruction executed.
Floating-point instruction:	Counts number of floating-point instruction executed.
Branch instruction:	Counts number of branch instruction executed.
Branch misprediction:	Counts mispredicted branches.
D-cache miss:	Counts (primary) d-cache misses.
I-cache miss:	Counts (primary) i-cache misses.
B-cache miss:	Counts (secondary) b-cache misses.
B-cache victim:	Counts victim writes. A victim write occurs when a dirty block is evicted from the b-cache.
PALmode:	Counts cycle spent in PALmode, which is a special, privileged mode that gives access to resources not normally visible in kernel-mode.

Table 1: 21064 Performance Monitor Event Sources

The 21064 performance monitor essentially consists of a counter that, every 4096 ticks, resets itself and generates an interrupt. The counter can be multiplexed among 17 different event sources [Dig92]. The event sources are described in Table 1. Figure 3 schematically shows the relationship between event sources relating to CPU instruction issue performance. Suppose we observe the CPU for 1000 cycles. During some of these cycles the CPU is able to issue two instructions simultaneously, during others only one can issue, and in the worst case, nothing at all can be issued. For simplicity, the diagram shows these three cases as contiguous groups—in reality, single, dual, and no-issue cycles are freely interspersed. As shown in the figure, total issue rate and total non-issue rate are related to other event sources as follows:

$$\begin{aligned}\text{total issue rate} &= \text{single issue rate} + 2 \cdot \text{dual issue rate} \\ \text{total non-issue rate} &= 2 \cdot (\text{pipeline dry rate} + \text{pipeline frozen rate}) + \text{single issue rate}\end{aligned}$$

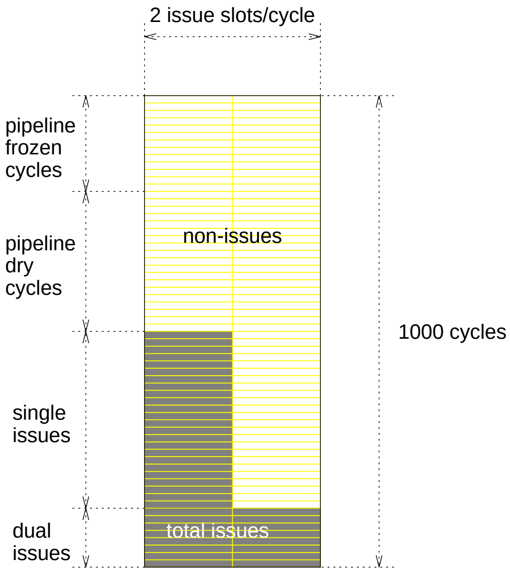

Figure 3: Relationship among instruction-issue related event-sources.

For instance, a program that continuously dual-issues instructions would achieve a total issue rate of 2000 instructions per 1000 cycles, resulting in a cycles-per-instruction (CPI) value of 0.5. Total issue rate is also related to the issue rates for the individual instruction classes:

$$\text{total issue rate} \quad = \quad \text{integer op rate} + \text{load rate} + \text{store rate} + \text{floating-point rate} + \text{branch rate}$$

None of the test-programs uses floating-point operations and, for the remainder of this paper, floating-point instruction rate should implicitly be understood to be zero.

The CPU can enable and disable the performance counter, but unfortunately, it cannot *read* the current counter value. Thus, rather than accurate counts, one is limited to measure *average event rates*. For example, to measure i-cache miss rate, one can set the multiplexor to select the "i-cache miss" event source and measure the time between two successive performance monitor interrupts. The rate is then given by 4096 misses/elapsed time. Fortunately, the CPU provides an independent cycle-counter register so that at least the elapsed time can be measured at the resolution of individual CPU cycles. Even so, care has to be taken to measure meaningful rates—in the worst case, there is an interrupt every 2,048 cycles. Without proper precaution, interrupt handling alone would account for the overwhelming portion of this time and the measured rate would reflect more the behavior of the interrupt handling itself, rather than the program under study. With careful coding of the interrupt handler, we were able to achieve interrupt overheads of less than 15% even in a worst-case scenario. Measured rates therefore should be accurate to within that percentage. Fortunately, typical accuracy is more in the 2–3 percent range.

3.2 Test Cases

We measured three versions of a TCP/IP stack. The first one, labeled Std, is a standard *x*-kernel implementation of the stack. Except for continuations, this version does not employ any latency improvement technique. In contrast, version Opt employs all the techniques discussed in Section 2, except for the last call optimization. Bad uses code identical to Opt, but the cloning facility was used to force 118 replacement misses in the b-cache. This allows us to quantify the sensitivity to cache behavior. Latency was measured by ping-ponging 1B TCP/IP messages between a client and a server machine.[2] For each version, the test was run ten times and involved exchanging 200,000 messages each. The latency number reported is the *processing* time required to complete a full roundtrip. In particular, the time on the wire has been factored out.[3] In practice, controller latency is also significant. In [TL93], it was reported to be 51 μs for an earlier generation workstation that uses the same LANCE controller. We believe controller latency is about the same in our system but because were unable to measure it directly, we do not subtract it. This has the effect that latency improvements will look somewhat worse than they really are. Throughput was measured by ping-ponging 20,000B TCP/IP messages between the client and the server. Because Ethernet's MTU is much smaller, computation can be overlapped with communication and throughput should be close to the theoretical maximum of 10Mbps. The maximum window size ("socket-buffer size") was set to 65120B. For each version, the test was run ten times and involved exchanging 2,000 messages each.

3.3 End-to-End Results

Table 2 shows the results we obtained. Entries of the form $\mu \pm \sigma$ indicate that the mean of the ten samples was μ and the sample standard-deviation was σ. The columns labeled Δ give the performance relative to column Opt. First, it shows that the latency of Std is 25% worse than it is for the optimized version. The throughput row shows that latency reduction was not achieved at the cost of reduced throughput. As a matter of fact, Opt has a slightly higher throughput than either of the other versions. While 25% is certainly respectable, it is less than we originally hoped for. However, in the light of the results for Bad, even this achievement appears to be of questionable value—just on the order of one-hundred b-cache misses are sufficient to increase latency to a value much worse than that for Std. In fact, we found that only a few dozen b-cache misses are needed to drop performance of Opt back to the level of Std. The conclusions we derive from these observations are discussed in Section 4.

	Opt	Std	Δ	Bad	Δ
Latency [μs]	211 ± 0.164	264 ± 0.121	+25%	394 ± 0.106	+87%
Throughput [Mbps]	$9.37 \pm 129 \cdot 10^{-6}$	$9.36 \pm 138 \cdot 10^{-6}$	-0.1%	9.34 ± 129^{-6}	-0.3%

Table 2: End-to-end Latency and Throughput

3.4 Detailed Analysis

To gain an understanding of the significance of each technique, we ran the same tests with the performance monitor enabled. With large messages, the results for the three versions were identical, so we report them under the single label Large. To achieve reasonable accuracy for the event rates, the number of exchanged messages was increased by a factor of ten. In return, the number of samples acquired was reduced from ten to five. Of the seventeen event sources,

[2] The PUSHALWAYS flag in the TCP session was turned on to ensure each 1B message really is sent immediately, rather than batched together to a single, MTU sized packet. Also, the isolated Ethernet showed no losses for all practical purposes.

[3] The time on the wire is $2 \cdot 72 \cdot 8 \cdot 100\,\text{ns} = 115.2\,\mu\text{s}$ because a roundtrip involves exchanging two minimum-sized frames of 72B each (64B plus 8B preamble) and at 10Mbps, transmitting a bit takes 100ns.

we did not measure total cycle rate, floating-point instruction rate, b-cache rates, and PALmode cycle rate. The first can be measured more accurately with the cycle-counter register and the latter have such small rates that they would require excessively long run times for the test program.

We found that, across the five samples, normalized standard deviation is 2.5%, which is much smaller than the potential systematic error due to interrupt overheads. It is therefore sufficient to simply report average rates. Because fractional events are not intuitive, we scaled event rates by a factor of 1000. For example, we report the average number of i-cache misses as 22 in a thousand cycles rather than as 0.022 per cycle. Table 3 shows the measured rates. Clearly, the numbers do not always add up perfectly. For example, in column Large, total issue rate was measured as 322 but the sum of the individual instruction issue rates is 371 (209 + 42 + 78 + 42), accounting for an error of 15%. Fortunately, in all other cases the error is no more than 3%. It is not immediately clear why accuracy in the Large case was so much worse. Figures 4 and 5 illustrate some of the same data graphically.

	Large	Opt	Std	Bad
total issue rate	**322**	**311**	**288**	**166**
integer op rate	209	147	147	78
branch rate	42	47	42	24
load rate	78	73	61	38
store rate	42	42	36	21
total non-issue rate	**1601**	**1660**	**1693**	**1811**
dry cycle rate	370	495	540	631
frozen cycle rate	304	217	205	209
dual-issue rate	**70**	**44**	**41**	**24**
i-cache miss rate	**22**	**30**	**33**	**23**
d-cache miss rate	**29**	**18**	**15**	**9**
branch mispred. rate	**11**	**16**	**12**	**9**

Table 3: Measured Event Rates

There are several interesting observations. First, the total issue rate given as the sum of instruction-class issue rates (rows 2–5 in Table 3) shows that, on average, the CPU is never busy for more than 37% of the time. It is unrealistic to expect the theoretically possible CPI of 0.5 for this CPU and code. Even with a perfect memory system, dual-issue opportunities are rare (the functional units in the CPU are independent, but not replicated). However, the measured CPI of more than 3 is clearly higher than anticipated. Apparently, memory-bandwidth is a serious bottleneck, and this despite the fact that the entire kernel fits into the b-cache!

Second, note that the Large case is memory-bandwidth limited as it involves copying and checksumming 80,000B per roundtrip. If we compare the load and store rates of Large and Opt in Figure 4, we find that they are almost identical. That is, even when processing small messages, the path is memory bandwidth limited. Figure 5 shows why—the additional bandwidth available to Opt due to smaller messages (relative to Large) is completely absorbed by a corresponding increase in i-cache miss rate. In essence, these measurements indicate that during protocol processing, a path is either instruction-bandwidth or data-bandwidth limited, but never CPU limited. That is, without dramatic code improvements, performance is bound to scale with memory bandwidth, rather than CPU performance. As a side note, Figure 5 also shows that the maximum b-cache miss-rate is about 50 in one thousand cycles. This is not

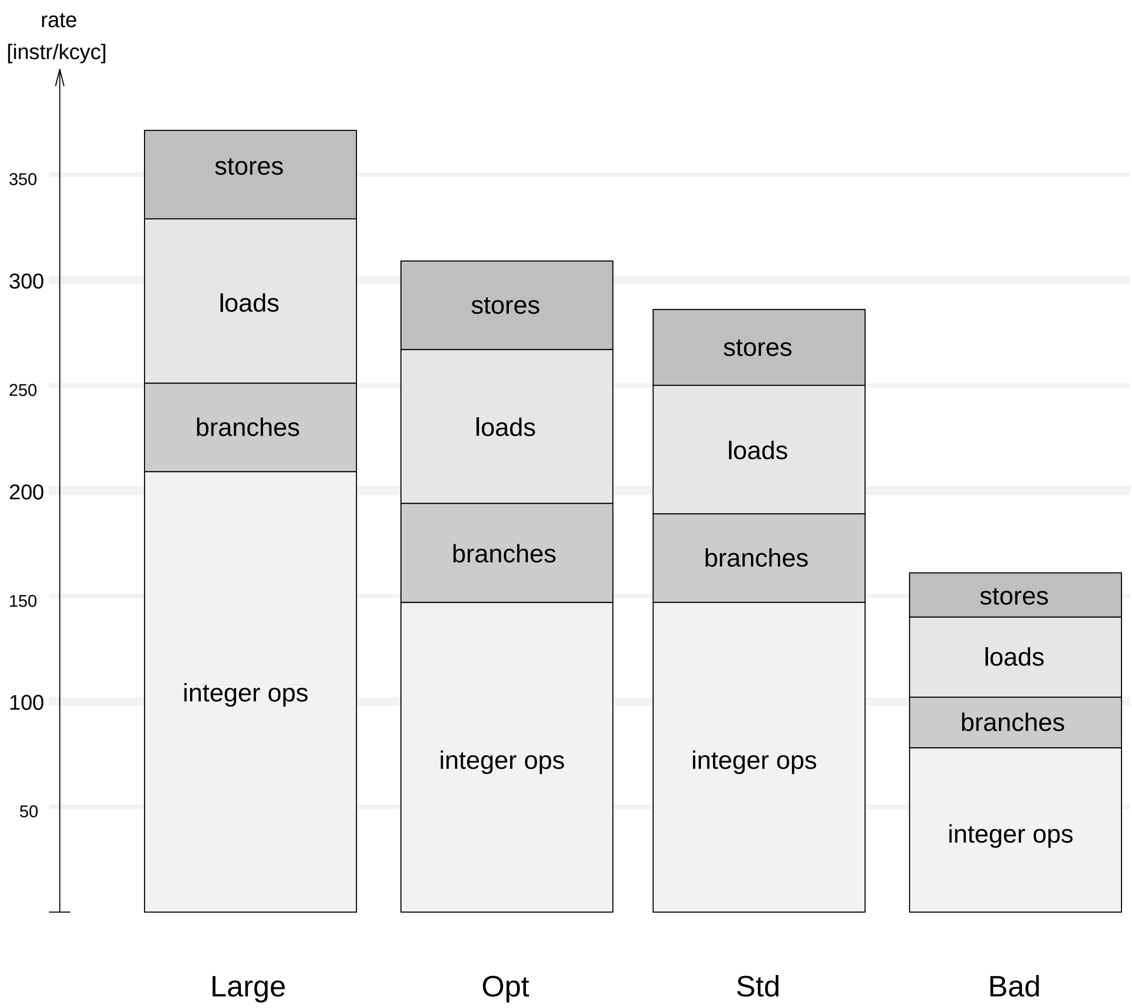

Figure 4: Measured Instruction Issue Rates

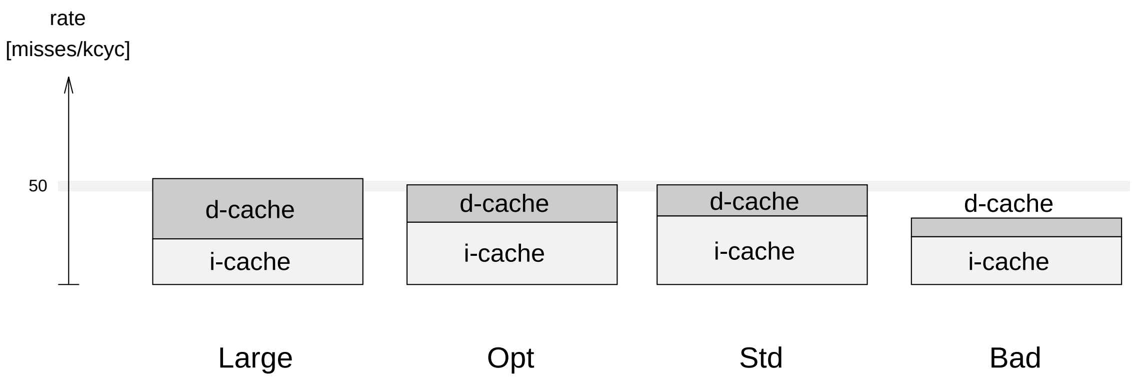

Figure 5: Measured Cache Miss Rates

unreasonable. Loading a b-cache block takes 10 cycles, which would correspond to a rate of 100 per thousand cycles. However, transferring the b-cache block into the d- or i-cache takes additional time. For example, an i-cache miss that can be satisfied from the prefetch buffer in the 21064 CPU still takes 9 cycles to complete [Eus94]. Also, each i-cache miss results in an additional i-cache prefetch request that may or may not be useful. It is not unreasonable that these additional overheads account for another 10 cycles on average. An average latency of 20 cycles per miss corresponds to the measured rate of 50 per thousand cycles.

Third, comparing version Opt and Std in Figure 4 we see that integer operation issue rates are identical, while Opt has slightly higher rates for loads, stores, and branches. This implies that Opt has fewer integer operate instructions available, relative to Std. Most likely, this is a result of inlining. As Figure 5 shows, Opt has a slightly lower i-cache miss rate. It appears that due to outlining and cloning, the i-cache is used more effectively. The resulting reduction in memory-bandwidth load can in return be absorbed for d-cache misses, thus resulting in higher load and store rates. Similarly, if more branches hit the i-cache, they can be executed at a higher rate, as is the case for Opt.

Finally, Figure 4 makes it apparent that in version Bad, 118 b-cache replacement misses are sufficient to increase the CPI from roughly 3 to almost 7.

Table 4 presents additional numbers that can be derived, either from the rates measured with the performance monitor or from the collected traces. The tracing facility cannot accommodate large messages and the corresponding entries in column Large are marked not available (n/a). The first row shows that outlining reduced the number of branches by 89: from 582 (Std) down to 493 (Opt), or by about 15%. The number of if-statements that are annotated (for the benefit of outlining) is 79. This close correspondence indicates that, for instructions, there is very little temporal locality of reference.

	Large	Opt	Std	Bad
branch instr. count	**n/a**	**493**	**582**	**493**
conditional	n/a	457	537	457
mispredicted	n/a	155	156	174
i-cache miss ratio				
measured	6%	10%	12%	14%
simulated	n/a	10%	12%	16%
d-cache miss ratio				
measured	24%	16%	15%	15%
simulated	n/a	18%	21%	19%
b-cache repl. misses	**n/a**	**0**	**0**	**118**

Table 4: Derived Quantities

The second row in Table 4 gives the number of conditional branches in the traces. To compute the number of mispredicted branches, we can first derive the branch misprediction *ratio* by dividing the branch misprediction *rate* by the branch instruction rate. If this ratio is multiplied by the number of conditional branches along the path, we arrive at the number of mispredicted branches. Curiously, as the third row in Table 4 shows, Opt has almost exactly the same number of mispredicted branches as Std even though the former has 80 fewer conditional branches. It appears that outlining got rid of all the branches that were easy to predict in the first place. This is not surprising because the 21064 uses a branch-history table that can achieve almost perfect accuracy for branches around infrequently executed basic blocks. The issue is mostly moot, however, as we also experimented with turning off the branch-history hardware and found no measurable difference. Most likely, branch mispredictions can be overlapped with memory system accesses and the difference in the number of CPU stalls just doesn't affect end-to-end latency.

We also used the traces to simulate the memory hierarchy. The traces cover only actual protocol processing. In particular, they exclude interrupt processing. Nevertheless, they provide an opportunity to compare the miss ratios derived from performance monitor measurements with simulation results based on the traces. As shown in rows four and five of Table 4, the i-cache miss ratios agree very well. For d-cache miss ratios (rows 6 and 7) the simulation results are more pessimistic than the measured ratios. This difference is most likely due to the fact that the simulator assumed cold caches at the beginning of the simulation. This is realistic for the i-cache, but not for the d-cache. With small messages, the d-cache is large enough to keep data cached across multiple path activations (e.g., the stack memory is likely to be in the cache already). We expect d-cache simulation accuracy could be improved by collecting traces spanning multiple path activations, or possibly simply by simulating the same trace repeatedly until a steady initial state is achieved.

If we go back and compare Tables 2 and 3, we find an apparent discrepancy: end-to-end latency for Std is 25% worse than for Opt, but total instruction issue rate for Opt is only 8% better! Where did the remaining 17% come from? The answer lies in different (dynamic) path lengths. Again excluding interrupt processing, the Opt trace is 4692 instructions long, while Std is 5825 instructions long. Thus, Opt requires about 20% fewer instructions to complete protocol processing. Clearly, most of the end-to-end latency improvement must be attributed to the shorter code path, not to improved processing rates that result from outlining, cloning, and better scheduling. We would like to emphasize that this latency improvement is *not* a result of instruction execution time reduction—the CPU could have easily processed instructions at twice the observed rate. That is, instructions are expensive not because of the time it takes to executed them in the CPU but because of the time it takes to load them into the i-cache.

Given how important size reduction is to latency improvement, it is interesting to see which technique has the biggest impact. Table 5 gives a break down of the (dynamic) instruction count savings by technique. As the first row shows, the winner-by-surprise is the simple TCP change that replaced some byte and short variables in the session state by words. Again, this is due to the absence of byte and short memory-accesses in the Alpha architecture. Other architectures do not have this problem. 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 by 208 instructions. Interestingly, this improvement does *not* affect end-to-end latency. 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. Separate measurements that are not listed here for brevity give evidence: pure protocol processing time (excluding interrupt handling) for Std is over 33% slower than for Opt, while end-to-end latency is only 25% slower. Clearly, the change does reduce CPU utilization, which is also desirable. At the lowest rank, we find the optimization built into cloning that skips part of the function prologue where possible. It is certainly not a very important optimization, but given its small cost, there is no reason not to do it. There are a number of ways to evaluate outlining effectiveness. We showed already that outlining reduced the number of branches by about 15%. Another measure is the percentage of instructions that remain unused in each cache block. Analysis of the traces shows that, on average, version Std leaves 21% of the instructions in a block unused (corresponding to 1.68 instructions out of 8). Thanks to outlining, average unused i-cache space in Opt is only 12% (or 0.96 instructions out of 8). An improvement by almost a factor of two is respectable, especially when considering that outlining was applied conservatively. Outlining is even more dramatic when measuring the amount of code that was outlined—of 5750 instructions 2037, or 35%, could be outlined! Thus, we consider outlining a useful technique not so much because of its direct benefits, but rather as a means to greatly

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
Use sentinels in *free()*:	60
Skip part of function prologue:	55

Table 5: Dynamic Instruction Count Reduction By Technique

improve cloning effectiveness. 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.

4 Conclusions and Future Work

Our experiments suggest two main conclusions. First, none of the techniques clearly dominates the others in terms of effectiveness. By itself, each technique is rather insignificant, but when applied simultaneously, they do add up to a respectable latency decrease. While respectable, the decrease is still fractional. Clearly, improving latency is hard. Fortunately, most of the techniques apply to the infrastructure, and are not protocol specific. This allows us to amortize the investment over many different protocol stacks. The few protocol-specific improvements that we applied are either straightforward (e.g., replacing shorts by integers), or employ a tool that make them readily applicable to other protocols (e.g., USC for the LANCE driver).

Second, we selected the techniques based on their potential to improve cache behavior during protocol processing. However, as our measurements show, most of the achieved latency decrease has to be attributed to the shorter path length, not to reduced CPU stall rates. This should not come as a complete surprise: the test system is so small that there are no pathological cache effects in version Std, even though it scatters protocol processing functions more or less "randomly" in the instruction segment. In essence, average behavior of direct mapped caches is much closer to the optimum than to the worst-case. But as illustrated by version Bad, relatively few unfortunate replacement misses in the b-cache can deteriorate end-to-end latency profoundly. In fact, we found that as few as one or two dozen b-cache misses are sufficient to drop performance of Opt back to the level of Std. As systems tend to grow, so does the likelihood that a "random" placement of functions will result in pathological cache behavior. Consequently, large systems often exhibit unpredictable performance. Worse still, performance of one subsystem (e.g., networking) may be affected by trivial changes to another subsystem (e.g., filesystem). We believe that the techniques presented in this paper are fundamental to solving this problem. But they are clearly just a first step. The more global question of how subsystems that are individually well-behaved can be molded into a single unit without introducing adverse cache effects remains an open question and is currently a topic of active research.

In addition to this question of global cache management, there is one latency-improving technique that we have not yet had time to explore. It is what we refer to as *super-inlining*. The idea is to compile the critical path of important protocol-stacks as one big function. The hope is that benefits similar to a non-layered (flat) implementation could be achieved without giving up on the modular structure of the source code. We envision super-inlining as a preprocessing step: given the sequence of functions constituting an important critical path, the preprocessor would gather the necessary C source code from the modular protocol implementations and transform it into a single, well-formed C function. The transformation involves inline-expanding code for direct and indirect function calls and applying appropriate renaming to avoid name-clashes (e.g., for type names). Indirect function calls and references to state that needs to be shared with the modular code provide a special challenge. But we believe these can be solved with a few simple stylistic conventions. The super-function generated by the preprocessor can then translated by a normal C compiler.

Super-inlining obviously would avoid most of the function-call overheads. More importantly, we believe that the increased context visible to the compiler will allow it to generate code that is of much superior quality than what is arrived at through separate compilation. This is particularly the case because C compilers make very conservative assumptions on the effects of function calls. Initial experiments with manual super-inlining are indeed very encouraging. They also made it apparent that super-inlining is too error-prone to be applied manually to any reasonably complex protocol stack. A concern with super-inlining is that compilers may break down when dealing with such large functions. For example, it is not clear how well current global register allocators would perform on such large and complex functions. Also, it is very likely that a ground-up flat implementation of a protocol stack will always perform better than any automatic technique based on modular source. Nevertheless, correctness and maintenance concerns make super-inlining very attractive.

Super-inlining can be viewed as an early form of cloning. Instead of at runtime, it is performed at compile time. Compile-time information allows much more aggressive optimization than is possible at runtime. Dynamic information, such as the sequence of functions constituting a path, has to be supplanted by static, system-builder provided data. That is, runtime cloning provides flexibility and adaptivity while super-inlining provides ultimate performance. Runtime cloning and super-inlining are not necessarily mutually exclusive, however. For example, any Internet protocol communicating through a specific network adaptor will share the code path between the driver and the IP protocol. Similarly, all ftp sessions share the code path between the FTP and the IP protocol. Thus, it may be reasonable to use super-inlining on these subpaths and employ runtime cloning to glue partial paths into specific end-to-end paths.

Acknowledgments

We would like to thank Allen Brady Montz for providing us with an implementation of the last-call optimization for the GNU C Compiler. Several people at DEC provided valuable insight into the Alpha architecture. In particular, we are grateful for Jeffrey Mogul's relentless help.

References

[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.

[Dig92] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor Hardware Reference Manual*. Digital Press, Maynard, Massachusetts, first edition, October 1992. Order number EC-N0079-72.

[Er83] M. C. Er. Optimizing procedure calls and returns. *Software Practice and Experience*, 13:921–939, 1983.

[Eus94] Alan Eustace. Personal communication, October 1994.

[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.

[HMPT89] Norman C. Hutchinson, Shivakant Mishra, Larry L. Peterson, and Vicraj T. Thomas. Tools for implementing network protocols. *Software—Practice and Experience*, 19(9):895–916, September 1989.

[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.

[MO93] Edwin F. Menze III and Hilarie K. Orman. *x-Kernel Programmer's Manual*. Tucson, AZ 85721, 1993. `http://www.cs.arizona.edu/xkernel/www/manual/manual.html`.

[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.

[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.