Tool-Based Specialization of Protocol Stacks in Operating System Kernels

Sapan Bhatia* Charles Consel* Anne-Françoise Le Meur* Calton Pu†

*LaBRI/INRIA
Domaine Universitaire
33400 Talence, FRANCE
{bhatia,consel,lemeur}@labri.fr

†Georgia Institute of Technology
Atlanta, Georgia 30332
calton@cc.gatech.edu

Abstract

In this paper, we use *program specialization* to optimize protocol stacks in operating systems kernels. We argue that due to their generic design, protocol stack implementations are inherently inefficient, and can be systematically optimized to achieve significant gains in terms of performance as well as code footprint. We have successfully specialized the TCP/IP implementation in the Linux kernel and used the optimized protocol stack in existing applications. These applications were minimally modified to request the specialization of code based on a particular usage context, and to use the specialized code generated instead of its generic version.

In particular, we conducted a series of experiments to evaluate the performance and size benefits of our approach. These experiments were conducted on three platforms: a Pentium III (700MHz), a 486 (40MHz) and an ARM SA1100 (200MHz) on a Compaq iPAQ. The specialized code runs about 25% faster and its size reduces by up to a factor of 20. The throughput of the protocol stack improves by up to 21%.

1 Introduction

Program specialization [12] has been acknowledged to be a powerful technique for optimizing systems code [22]. Conceptually, a specializer takes a generic program as input along with a *specialization context* consisting of values of known data items. It then evaluates the parts of the program that depend only on these known values, and produces a simplified program, which is thus *specialized* for the supplied specialization context. For example, a specialization context in the process of sending TCP packets can define the Maximum Segment Size (MSS) associated with the connection as known and invariant. In this context, the TCP `send` routine and dependent functions can be specialized, such that conditional tests and expressions depending on MSS can be fully or partially evaluated. Thus, the process of specialization exploits specialization opportunities that arise from program invariants. The knowledge of invariants enables specializers to optimize programs in many ways. Invariants can be inlined as *instruction immediates* instead of being referred to, mathematical expressions can be reduced in complexity, and conditional tests that depend on known variables alone can be evaluated early, eliminating entire branches of code.

In this project, we have used program specialization to optimize the TCP/IP implementation of the Linux kernel. We call our use of specialization *tool-based specialization* as it relies on well-known tools in their readily available form. This is crucial in making our approach portable to a variety of Operating Systems (OSes) and OS subsystems. These tools include: the Tempo C specializer to perform specialization [5], the GNU C Compiler to compile the specialized code, and GNU modutils to load and manage the specialized code in the kernel.

In our approach, applications that wish to use a specialized version of the protocol stack issue a *specialization request* through a documented system call, as soon as the specialization context becomes available. This causes a run-time system to collect the specialization context, as specified by the application and contained by the system environment. This context is then made available to Tempo, which uses it to generate specialized code. The specialized code is then compiled using GCC, loaded into the kernel, and hooked into the calling process to be used by the application. Typically, the specialization context remains valid through the duration of the session.

Experiments conducted using this setup have shown that there is a notable code speedup, as well as a reduction in code size. In the case of the UDP protocol, the size of the specialized code once compiled is only about 5% of the generic compiled code. For TCP, this ratio is less than 3%. The execution time of the code in the case of UDP decreases by about 26% on a Pentium III (700MHz) and the local throughput of 1Kb packets increases by about 13%. For a favorable packet size of 64b, this improvement is about 16%. On a 486, the increase in throughput for 1Kb packets is about 27%. For TCP, the throughput increases by about 10% on the Pentium III and about 23% on the 486. On an iPAQ running an SA1100 processor at 200MHz, we observe an improvement of about 18% in the throughput of 1Kb packets for UDP.

The rest of the paper is organized as follows. Section 2 first identifies the different specialization opportunities that exist in the implementation of a typical UNIX protocol stack and then describes under which assumptions these optimizations can be safely performed. Section 3 explains how the specialization process is concretely integrated and automatically enabled. Section 4 describes and analyzes the performance measurements obtained while performing various experiments. Section 5 presents some related work, and finally Section 6 concludes.

2 Specialization of protocol stacks: concept and mechanism

Due to its generic design, a typical UNIX implementation of a protocol stack – including the socket layer, the transport layer (UDP or TCP) and the IP layer – exhibits many specialization opportunities. Identifying a specialization opportunity amounts to finding fragments of code that, in a given execution context, depend on invariants and thus can be evaluated once and for all at specialization time. In this section, we first describe and illustrate with some code excerpts[1] the specialization opportunities and invariants that we have identified. Then we explain how, using code guards, we can ensure that the system remains stable even when an invariant ceases to be valid.

2.1 Specialization opportunities

We begin by presenting the main specialization opportunities that we have identified and exploited. To ensure that none of these opportunities are based on specific aspects of the Linux implementation, and can be thus generalized, we have also cross-compared the code we have analyzed in Linux to that in FreeBSD, and considered only opportunities that exist in both code bases.

2.1.1 Sockets, UDP and IP layers

Following is a list of the specialization opportunities with the invariants they depend on:

Eliminating lookups. Every time a send call is issued, the system call handler retrieves the socket structure, based on the file descriptor `fd`, as shown in the following code extract:

```
asmlinkage long sys_sendto(int fd, void * buff, size_t len, unsigned flags,
                           struct sockaddr *addr, int addr_len) {
    ...
    sock = sockfd_lookup(fd, &err);
}
```

[1] All the excerpts of code, including the comments, are taken directly from the source code of Linux and BSD, with only minor formatting changes.

The routine that performs this operation lies in the file system subsystem, and thus is expected to hinder the instruction cache. This feature allows processes to use normal file I/O mechanisms with socket descriptors. Since the binding between a socket descriptor and the socket structure does not change once the socket has been created, the code can be specialized so that the socket structure and its fields are inlined. Besides, since these field values are now explicit, further optimizations can be performed. Thus, the reference to the target socket structure is defined as static (*i.e.,* invariant), and is retrieved once and for all at specialization time.

Eliminating interpretation of options. Throughout the network code in the kernel, we observed that socket options and attributes are interpreted. Some such attributes, as illustrated by the following excerpt, inform whether the session is blocking (`O_NONBLOCK`), whether the message is being sent to probe the MTU (`msg_controllen`), whether the address is unicast or multicast (`MULTICAST`), *etc.*

```
if (MULTICAST(daddr))
...
if (sock->file->f_flags & O_NONBLOCK)
...
if (msg->msg_controllen)
...
```

When socket attributes are invariant, the computations that depend on them can be performed once and for all at specialization time, instead of being repeatedly performed at run time.

Eliminating routing decisions. The route associated with the destination address of a packet is validated when the packet headers are constructed. This is to cope with situations in which the route changes during a transmission session. The occurrence of such an event is, however, extremely unlikely and can be neglected for most applications. These occurences are guarded against, as discussed in Section 2.2. Thus, we can freeze the route, or more precisely, use the destination cache, without validating it for every packet. The destination cache is used to speed up the dispatch of fully formed IP datagrams. This circumvents the code that checks whether the route is obsolete, and allows us to inline in the code attributes associated with the destination, such as the destination IP address, the output interface, the destination port, *etc.*

Optimizing buffer allocation. We found that the routines used to allocate memory for the send buffer are also amenable to specialization. As can be seen in the code below, attributes of the allocation procedures, such as their behavior towards blocking the process, the zone in which the memory is allocated, *etc.* are interpreted and can be specialized.

```
if (0 && (gfp_mask & GFP_WAIT)) {                    /* Check if function can block */
    static int count = 0;
    if (++count < 5) { ... }                         /* Report a kernel error */
    gfp_mask &= ~GFP_WAIT;
    ...

    npages = (dlen + (PAGE_SIZE- 1)) >> PAGE_SHIFT;
    skb->truesize += dlen;
    ((struct skb_sharedinfo *) skb->end)->nr_frags = npages;
    for (i = 0; i < npages; i++) { ... }
```

Furthermore, if an application can be made to commit that it will be sending data in fixed Application Data Units (ADUs), several conditions and predicates based on the size of the allocated buffer can be reduced. For example, in the second half of the above excerpt, the variable `dlen`, which is a sum of the buffer size and some constant header sizes, becomes invariant. We can calculate `npages` and subsequently cause the following for-loop to unroll. The specialization of buffer allocation routines itself results in about 5% of speedup in the UDP send path, and a corresponding 3% of improvement in throughput on the Pentium.

2.1.2 TCP

As one can imagine, considering the richness of the TCP protocol, there can be several situations in which its complexity can be reduced. It often happens that the characteristics of the data transfer process are predictable, and can be exploited to specialize the TCP layer. We have listed opportunities for such specialization below. All the opportunities previously discussed in the context of sockets, UDP and IP apply to TCP as well.

The `tcp_send` routine, which is the entry point into the TCP layer for sending a packet, begins by determining whether the buffer being transmitted can be accommodated into the last unsent TCP segment. This process, called TCP coalescing reduces the number of small packets transmitted, and thus the header overhead as well.

```
if (tp->send_head == NULL  || (copy = MSS_now - last_skb_len) <= 0) {
    if (!tcp_memory_free(sk))                    /* Allocate new segment. If the interface is SG, */
        goto wait_for_sndbuf;                    /* allocate skb fitting to single page. */

    skb = tcp_alloc_pskb(sk, select_size(sk, tp), 0, sk->allocation);
```

We specialize this code by assuming that the Maximum Segment Size (MSS) associated with the connection and the size of the ADU are invariant (recall that the latter is also used in UDP). If the size of the ADU is a multiple of the MSS, TCP coalescing is ruled out, as every segment sent out is MSS-sized; in the code above, `MSS_NOW - last_skb_len` becomes zero. This information is also used to unroll the main loop (illustrated by the sugared block of code below) which fragments the ADU into multiple segments, if it is larger than the MSS, after filling the previous buffer's allocated space completely. The MSS for a connection is determined when a connection is established, and does not change unless the Path MTU (PMTU) for the current route changes. This situation is the condition used to *guard* the invariant, as discussed in detail in Section 3. Assuming a constant MSS also enables us to specialize out Nagle's algorithm [20], since there are never small packets in flight. Finally, having an ADU size smaller than the MSS is beneficial to the receiver, since it saves it from having to gather ADUs fragmented into multiple TCP segments.

```
while (seglen > 0)                               /* While some data remains to be sent */
{
    copy = MSS_now - last_skb_len;               /* Calculate bytes to push into previous skb */
    if (copy > 0) {                              /* Is there enough space in the previous skb? */
        if (copy < seglen)
            copy = seglen;
        push_into_previous(copy);                /* Push copy bytes of the buffer into the previous segment */
    }
    else {
        copy = min(seglen, MSS_now);             /* Take smaller of remaining data size and MSS */
        push_into_current(copy);                 /* Push copy bytes of the buffer into the current segment */
    }
    seglen -= copy;
}
```

In the function that actually puts a queued segment on the network, once again, flags and attributes are interpreted. This is illustrated below:

```
if (tcb->syn_flagged) {
    tcp_header_size = sizeof(struct tcphdr) + TCPOLEN_MSS;
    ...
}
```

There are also several variables in the congestion control algorithms that can be used for specialization. For example, the Selective Acknowledgments (SACK) option [16] is useful only in situations where multiple segments are lost in one window. Since a slow or congested network defeats the primary purpose of specialization anyhow, this option, and other similar ones can be removed from the fast-path.

Most congestion control features that are not mandatory correspond to system-wide variables (`sys_ctls`) that can be used to disable these features for the entire system. With specialization, we make these variables a part of the specialization context and set them on a per-process basis. Furthermore, since these are known at specialization time, we can use their values to specialize code that depends on them. In our experiments, we

have not used specialization to disable congestion control altogether. We specialize out only those congestion control mechanisms that become unnecessary as a result of assumed invariants.

Although the specialization opportunities in TCP outnumber those in the rest of the network stack code, there are many features that are seemingly unspecializable, and have been left out. Some of these opportunities are unexploited because the associated invariants are too complex to be handled by specialization. These invariants include algorithms that are invoked on comparing complex variables such as the congestion window, the number of unacknowledged segments, *etc.*

2.1.3 Cross-comparing with FreeBSD

As mentioned earlier, the specialization opportunities exploited in this project occur across UNIX systems. This is confirmed by an analysis of the FreeBSD-5.1 sources. We find that the opportunities listed in the Linux Sockets/UDP/IP layers exist in FreeBSD as well. Every time the `send` routine is invoked, a lookup is done to retrieve the socket structure:

```
mtx_lock(&Giant);
if ((error = fgetsock(td, s, &so, NULL)) != 0)
    goto bad2;
```

Furthermore, options are interpreted:

```
dontroute =   (flags & MSG_DONTROUTE)
           && (so->so_options & SO_DONTROUTE) == 0
           && (so->so_proto->pr_flags & PR_ATOMIC);
if (control)
    clen = control->m_len;
```

Unlike Linux, which uses linear socket buffers, BSD uses chains of small fixed-size `mbuf` structures for its network buffers. Apart from the small fixed-sized region (typically 112 bytes) available in the `mbuf`, data can be stored in a separate memory area, managed using a private page map and maintained by the `mbuf` utilities. We found that due to its complexity, there are far more opportunities for specialization in the allocation system used by BSD than there is in the linear `sk_buffs` in Linux. Supposedly, a key reason to use `mbuf` structures in BSD is the fact that memory was far more expensive at the time it was designed. BSD copes with this design by using clusters to get as close to linear-buffer behavior as possible. This behavior is invariant at run time, and thus can be specialized.

Figure 1 contains a fragment of the fast-path of the UDP send operation. All the conditionals that depend on invariants are printed in boldface. As can be observed, this code will be drastically pruned by specialization.

The routing decisions in the IP layer (the `ip_output` function) closely resemble the ones in Linux and offer the same specialization opportunities. The assumptions made in TCP are all protocol centric and we do not depend on any Linux specificity, such as its formulation of segments in flight or RTO calculation algorithm. Specifying the ADU size explicitly, freezing the MSS (it is calculated based on the PMTU, like in Linux), avoiding the SWS algorithm and the explicit specialization-time removal of optional features such as ECN and SACK are available in FreeBSD as well.

2.2 Code guards

Should at any time, an invariant used for specialization cease to be valid, the corresponding optimized code would be invalid as well. Although most events that cause this to happen are highly improbable, they are nevertheless possible, and one needs to ensure that on their occurrence, the system is returned to a consistent state. To do so we use *code guards*, as introduced by Pu *et al.* [22]. They describe the dynamics of establishing guards and the process of replugging. Although we have not used a tool that lists out all the possible sites where invariant variables can be modified in a fool-proof way, as they did, instead we have used the LXR source cross-referencing system. Linux arbitrates access to shared variables in a systematic way, using accessor functions, macros, *etc.*, uniformly throughout the kernel. Although this strategy simplifies the process of isolating modification sites, the existence of assembly code and functions directly manipulating memory makes it very difficult to guarantee completeness.

```
do {
   if (uio == NULL) {                                        /* Data is prepackaged in "top" */
      resid = 0;
      if (flags & MSG_ EOR)
         top->m_flags |= M_EOR;
   } else do {
         if (top == 0) {
              MGETHDR(m, M_WAIT, MT_DATA);
              if (m == NULL) {
                error = ENOBUFS;
                goto release;
              }
              mlen = MHLEN;
              m->m_pkthdr.len = 0;
              m->m_pkthdr.rcvif = (struct ifnet *)0;
         } else {
              MGET(m, M_ WAIT, MT_ DATA);
              if (m == NULL) {
                 error = ENOBUFS;
                 goto release;
              }
              mlen = MLEN;
         }
         if (resid >=  MINCLSIZE) {
              MCLGET(m, M_ WAIT);
              if ((m-> m_ flags & M_ EXT) == 0)
                 goto nopages;
              mlen = MCLBYTES;
              len = min(min(mlen, resid), space);
         } else {
              len = min(min(mlen, resid), space);
              if (atomic && top == 0 && len <  mlen)          /* For datagram protocols, leave room */
                 MH_ALIGN(m, len);                            /* for protocol headers in first mbuf */
         }
         ...
}
while (!buffer_sent);
```

Figure 1: Fast-path of the UDP send operation in BSD

Events that can violate invariants can be classified into two categories: *application-triggered events* and *environment-triggered events.* An application triggered event is caused when an application invokes a routine that explicitly violates an invariant. It is relatively easy to guard against such events, since the guards can be established at the source, *i.e.,* at the entry points of such routines. Environment-triggered events on the other hand, are caused by side-effects on the state of the system. These events are more difficult to guard against. We have used the LXR tool to enumerate such events. We handle them as follows.

Application-triggered violations

- When an attempt is made to modify certain socket options during a session, a guard is made to reject the operation. This is achieved by routing the `setsockopt` system call to an entry in the local system call table of the process (see Section 3.2.2), which identically fails.
- Recall that the size of the buffer used in the `send` system call can be asserted invariant. This assumption is guarded similarly in the local process' version of the `send` system call, where it is checked against the expected length of the buffer. This expected length is specified by the application as a new socket option `(SO_ADU)`.

Environment-triggered violations

- When a socket is closed during the transmission of a packet, the specialized code is immediately invalidated, as it is based on a frozen socket structure which has ceased to exist. Closing a socket results in the closing of the associated file descriptor in the `filp_close` routine. The strategy used to recover from such a situation is as follows. The `filp_close` routine is made to acquire a semaphore associated with the instance of the specialized code that is being affected. When the specialized code is deployed at the time of specialization, it is protected by this semaphore at the boundaries. This strategy may admittedly increase the latency of the `close` operation in case it is invoked during the execution of the specialized code, but since the increase is limited by the time taken for one execution

```
C code:
struct sk_buff *sock_alloc_send_pskb( struct sock *sk,
                                      unsigned long header_len,
                                      unsigned long data_len,
                                      int noblock,
                                      int *errcode) {

   ...
}

Tempo specialization declarations:
 Sock_alloc_send_pskb :: intern sock_alloc_send_pskb( Spec_sock( struct sock)  S(*) sk,
                                                      S( unsigned long) header_len,
                                                      S( unsigned long) data_len,
                                                      S( int) noblock,
                                                      D( int *) errcode) {
   ...
 };
```

Figure 2: Specialization declarations

of the code path, and the situation is highly improbable, it is neglected. The use of synchronization primitives in code guards was first proposed by Pu *et al.* [22].

- When the route associated with a destination address changes during a session, we once again prevent the execution of the code using the route by acquiring a semaphore. In such a situation, there can be two possible courses of action. The first consists in stopping the execution of the old code, re-specializing the code according to the new route and then resuming the execution. This approach, however, is infeasible because it would stall the operation in progress for an extended length of time. We instead propose to reinforce the assumption by offsetting the behavior of the code. That is, instead of changing the code to make it correct, we offset the system to achieve the same result. Concretely, a Network Address Translation rule is installed as a reinforcement to ensure delivery to the correct physical destination. Using one or the other option could be a policy left to the system administrator.

- Every time an acknowledgment is received by the TCP sender, the MSS is recalculated. If the result differs from the value we have assumed, we propose to use a reinforcement based on a packet mangling rule. The Linux iptables package has a packet mangling target that allows one to modify the MSS of packets.

3 Enabling the specialization

Once the specialization opportunities have been identified, they must be described to the specializer so that only these opportunities are taken into consideration during specialization. Furthermore, specialization can either be launched at compile time or run time, depending on the stage at which specialization values become available. In the case of the protocol stack, specialization must be performed at run time because the specialization values are not known until the application executes. This situation raises a number of issues which are addressed in this section.

3.1 Describing the specialization opportunities to the specializer

The specializer Tempo provides a declaration language that allows one to describe the desired specialization by specifying both the code fragments to specialize and the invariants to consider [13, 14]. Concretely, this amounts to copying the C declarations in a separate file and decorating the types of each parameter with `S` if the parameter is an invariant and `D` otherwise. An example of the declarations we have written for the Linux protocol stack is shown in Figure 2. These declarations specify that the function `sock_alloc_send_pskb` has to be specialized for a context where the parameters `header_len`, `data_len`, `noblock` are invariant. Furthermore, the pointer `sk` is also an invariant and points to a socket data structure that exhibits invariant fields, as specified by `Spec_sock` which is not shown. These declarations enable Tempo to appropriately analyze the code. Once the analyses are done, the specialization may be performed as soon as the specialization context (*i.e.*, the values of the invariants) is made available.

3.2 Enabling specialization in the kernel

Most notably, we need to determine how the specializer is supplied with the kernel specialization values, and how it interacts with the kernel to execute static kernel fragments. These are key issues because, a priori, a tool like Tempo runs in user space and is linked with user-space libraries.

One strategy to overcome this problem would be to use a standalone run-time specializer, as proposed by Noel *et al.* [21] and implemented in Tempo. In this approach, the specializer is a simple program that manipulates binary templates and fills them with run-time values. Although the run-time specializer of Tempo has been successfully used for a number of applications [21], it has been found to be less efficient than its compile-time counterpart [6] because the template-based approach obstructs some optimizations.

Because our goal is to improve the performance of systems code that has already been manually optimized, not only should we use all key specialization opportunities in the protocol stack implementation, but we should also fully exploit the optimization opportunities exposed in the specialized code. Achieving the latter goal is done by using an aggressively optimizing compiler (not a template-based specializer).

To use the full potential of a compiler without perturbing the system that requires specialized code, we need to defer the specialization process to a remote server. This strategy is useful in such context as small devices but it entails separating the activities that constitute specialization of kernel code. That is, the running of the specializer, invoking the specializer with the specialization context, and loading the specialized code in the kernel.

Furthermore, specialization is most beneficial to long-running processes with stable invariants. This situation enables the specialization cost to be amortized over a long period of time, allowing extensive optimizations to be performed on the specialized code.

3.2.1 A scenario

In principle, our specialization architecture allows specialized versions to be requested for a fixed set of system calls that is defined at the time the OS is compiled. In the context of this paper, we focus on the `send` system call. The resulting specialized code is a streamlined and optimized version of all the functionalities required to perform the `send` operation.

Specialization of the `send` system call is requested through the corresponding entry in the global specialization interface. This entry, `do_customize_send`, corresponds to a macro function that expands into a unique system call, common to the entire interface. This is invoked as early as the specialization context becomes known, with the values forming the specialization context, such as the file descriptor, the destination address, the protocol to use, *etc.* This invocation returns a token, which is used by the application to refer to the version of the system call, specialized for the specific context. Defining a new token to multiplex operation instead of the file descriptor allows for multiple versions of the `send` system call to be used with the same file descriptor. This can be useful particularly in the use of TCP, where different instances of `send` can be requested for various anticipated traffic profiles.

Invoking the specialized version of the system call is done via `customized_send`, which takes three arguments less than the former, as they have been inlined into the specialized code. However, it takes one additional argument, namely the token.

3.2.2 Overview of our specialization strategy

Our strategy consists of four major phases: a contextualization phase collects the specialization values to be used by the specialization phase. This phase produces a specialized code which is then processed by an optimizing compiler. Finally, the compiled code is loaded into the kernel. Let us describe each phase in details.

Contextualization. Contextualization involves amassing the values of known parameters, *i.e.*, the specialization context, and making them available to the specializer. In this paper, we limit ourselves to the case in which contextualization, the specializer and the specialized code run on the same system. The infrastructure for situations in which contextualization happens on a small device and the specializer runs on a remote specialization server is described in [3].

The process of making the context available to the specializer is a crucial step. Indeed, the specializer actually executes parts of the specialized code that depend solely on invariants, and the specialization context must be exposed to this code in the form that it expects it. Specifically, values must be loaded into the right memory addresses. In the case of remote specialization [3], we use an emulator to furnish the values of the specialization context at the expected locations using CPU exceptions. We note that, in contrast with the case where the context needs to be transferred from one physical host to the other, in the local case, the context is available on the same machine. Thus, instead of extracting the specialization context and using an emulator to render it correctly for the specializer, we give the specializer direct access to kernel memory. The technique used to accomplish this is described in detail by Toshiuki in his masters thesis [15]. As a result, the specialization engine, although loaded in user-space, runs in supervisor mode and can access kernel memory as part of its own address space.

Specialization. The Tempo specializer is executed in kernel-mode with direct access to the specialization context in the form that it is available in kernel memory. At first glance, it appears that this would raise several issues related to concurrency. However, this is not the case because the data that the specializer operates upon, when it performs specialization-time execution, is identically invariant; it is what the process of specialization is based on. Some of these invariants are protected by guards, which themselves use synchronization primitives. Thus, this code is automatically protected against external influences.

There is only one possibility in which a fully static function being executed at specialization time could write to the external environment: modifying a potentially contentious variable. Tempo allows for values of static variables to be initialized in code, instead of being fed in at specialization time. Thus, a fully static function could initialize a static variable, assigning it its invariant value. This operation is also safe, as long as the original code (the kernel code) is safe and is written properly to use locks when accessing contentious variables. This follows from the fact that the process of specialization executes fully static functions as they are found in the original code. Since specialization preserves the identity of behavior of the code being specialized, this property of safety is preserved in the specialized code as well.

Compilation. Once the code has been specialized, it must be compiled to be loaded into the kernel. Since we use the Linux kernel-module interface to install code, it needs minor modification for compatibility reasons. Optimizing compilation is performed using GCC.

Loading. The compiled code is loaded into the kernel space, and its address is used to create an entry in a local system-call table. Notice that we introduce one such table per process. The offset of this entry in this table is returned to the application as the specialization token. Later, when an application invokes `customized_send`, it is directly routed into the specialized code via this entry.

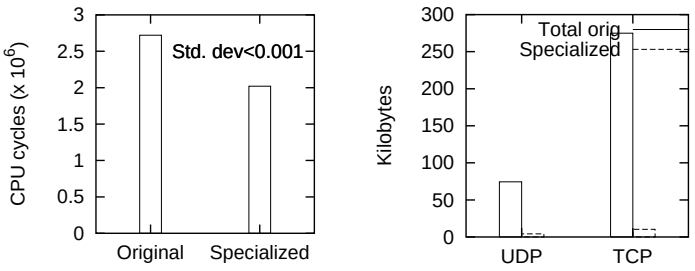
4 Experimental performance evaluation and analysis

In this section, we present the results of a series of experiments conducted to evaluate the impact of specialization on protocol stacks in OS kernels. Our setup consisted of three target devices: a Pentium III (PIII, 700MHz, 128MB RAM), a 486 (40MHz, 32MB RAM) and an iPAQ with an ARM SA1100 (200MHz, 32MB RAM). We evaluated the performance of specialized code produced using our architecture for each of these individually. It is worth mentioning that in all three cases, the specialized code was generated on the PIII, running under normal working conditions of processor and disk activity and utilization. On the PIII itself, this process follows the approach described in Section 3 and on the 486 and iPAQ, the approach described in [3]. We used version 2.4.20 of the Linux kernel for our implementation and all our experiments.

We first describe the experiments conducted, then present the results and finally characterize them and conclude.

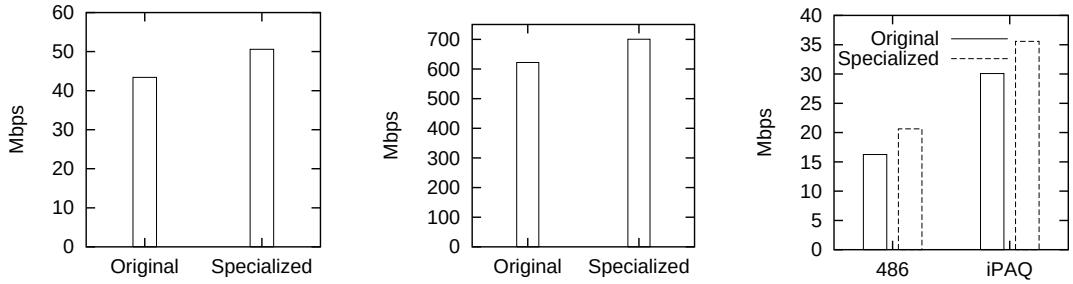
4.1 Experiments

The experiments conducted compare the performance of the original TCP/IP stack to that of the specialized code produced for performing basic data transfer over the network. The measurements were carried out in



(a) Performance (b) Code size

Figure 3: Original vs. specialized code: performance and size



(a) PIII, 64b (b) PIII, 1Kb (c) 486/iPAQ, 1Kb

Figure 4: Maximum Throughput for 64b and 1Kb UDP packets on the PIII, 486 and iPAQ ±5% at 99% confidence

two stages:

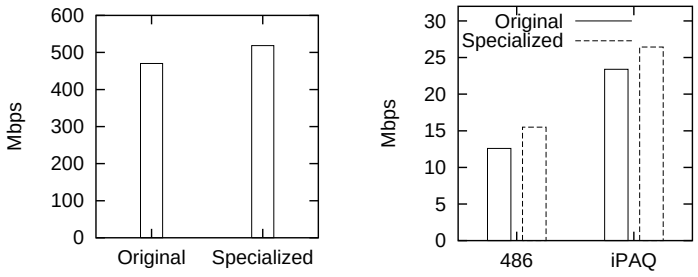
- Measuring code speedup. The following experiment was repeated several times: send a burst of UDP packets over the local loop-back interface, and record the number of CPU cycles taken by the pertinent code (*i.e.,* the socket, UDP and IP layers) in the un-specialized and specialized versions. These measurements were performed in-kernel.
- Measuring throughput improvement. The *Netperf benchmark suite* [4] was used to find the impact of specialization on the actual data throughput, measured over the local loop-back interface. The results shown compare the throughput measured by the original implementation of Netperf using the un-specialized stack, to a modified version using the specialized code produced by the specialization engine. The latter was modified to use the specialization interface.

Along with the results of these experiments, we also present the associated overheads in performing specialization.

4.2 Size and performance of specialized code

Figure 3(a) compares the number of CPU cycles consumed by the Socket, UDP and IP layers before and after specialization. We find that there is an improvement of about 25% in the speed of the code. It should be noted that this value is not affected by other kernel threads running on the system, as the kernel we have used is non-preemptable.

Figure 3(b) compares the size of the specialized code produced, to the size of the original code. The original code corresponds to both the main and auxiliary functionalities required to implement the protocol stack. The specialized code is a pruned and optimized version of the original code for a given specialization context. As can be noticed, the specialized code can be up to 20 times smaller than the original code.



(a) PIII (b) 486, iPAQ

Figure 5: Maximum throughput of TCP on the PIII, 486 and iPAQ ±2.5% at 99% confidence

	Cold cache	Warm cache
UDP	8.84sec	3.82sec
TCP	7.77sec	2.87sec

Figure 6: Specialization overhead

Figures 4(a) and 4(b) show a comparison between the throughput of the Socket, UDP and IP layers before and after specialization, measured by the UDP stream test of Netperf on the PIII. Figure 4(c) shows the same comparison for the 486 and the iPAQ respectively.

On the PIII, for a favorable packet size of 64b, the improvement in throughput is found to be about 16%, and for a more realistic size of 1Kb, it is about 13%. On the 486, the improvement for 1Kb packets is about 27%. For the iPAQ, again with 1Kb packets, the improvement is about 18%.

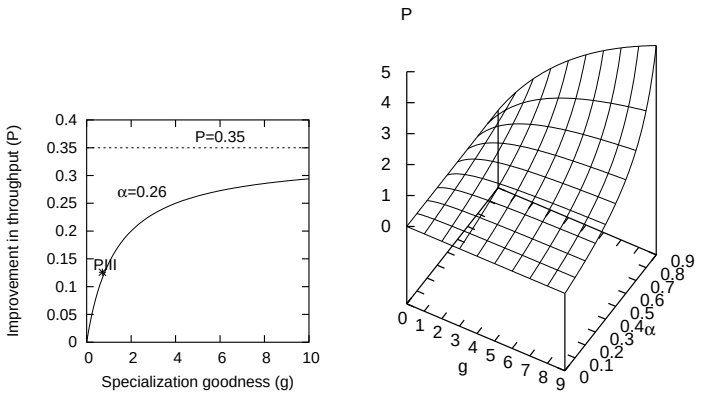
Figures 5(a) and 5(b) show a comparison between the throughput of the Socket, TCP and IP layers before and after specialization, measured by the TCP stream test of Netperf on the PIII, 486 and iPAQ. Corresponding to a TCP Maximum Segment Size of 1448 bytes, there is an improvement of about 10% on the PIII, 23% on the 486 and 13% on the iPAQ.

Finally, Figure 6 shows the overhead of performing specialization with the current version of our specialization engine, running on the PIII. It should be noted that the current version of our specialization engine is assembled from components that are implemented as separate programs, running as independent processes. Also, they are reloaded into memory every time specialization is performed. We are working on merging these components, in particular the specializer and compiler and on making the specialization engine a constantly running process. We expect these changes and other optimizations, such as using precompiled headers, to improve the performance of the specialization engine dramatically. Indeed, the overhead is presently dominated by these factors.

4.3 Characterizing the performance gains

In this subsection, we attempt to characterize the performance gains we have obtained in this project. This analysis is applicable to any I/O subsystem in which transfer rates are limited by the CPU. If u is the *unspecializable effort*, (*e.g.* code that cannot be specialized, context switch overhead *etc.*), v is the *specializable* effort, α is the code speedup achieved by specialization and g is the *specialization goodness*, which is defined as the ratio of the *specializable effort* and the unspecializable effort, the increase, P, in the transfer rate of such a subsystem can be calculated as follows:

$$\begin{aligned} P &= \frac{1/(u+v-v\alpha) - 1/(u+v)}{1/(u+v)} \\ &= \frac{u+v}{u+v-v\alpha} - 1 \end{aligned}$$



(a) α constant (b) α variable

Figure 7: Improvement vs. specialization goodness

$$\begin{aligned} & = \frac{1 + v/u}{1 + v/u - v\alpha/u} - 1 \\ \textit{or} & \\ P & = \frac{1 + g}{1 + g(1 - \alpha)} - 1 \end{aligned}$$

Figure 7(a) shows the improvement in throughput as a function of the specialization goodness. α here is taken to be 0.26, corresponding to the improvement in the UDP experiment on the PIII. As can be trivially deduced from the previous equation, the improvement is asymptotically bounded by the value $P_{max} = \frac{\alpha}{1-\alpha}$ which in this case, is about 0.35.

Figure 7(b) shows the improvement in throughput as a function of the specialization goodness and α. It is observed that g decreases as the CPU of the machine gets faster and the transmission process becomes more efficient (*e.g.* as the packet size increases). The improvement in throughput is therefore more than that on the PIII. Using this graph, we can predict the levels of improvement we are likely to get on various systems.

5 Related Work

Customizable OSes. The main purpose of customizability in OS research is to provide flexible mechanisms and policies, so that functionalities can suit the needs of applications and users. Denys *et al.* have performed a survey of customizable OSes and efforts to customize OSes [10]. Let us briefly present two key projects along this line. Exokernel, introduced by Engler *et al.* [8], tries to eliminate all kernel abstractions and lower the kernel interface almost to the bare hardware. Each customized behavior corresponds to a systems program; it is introduced as a special *Library Operating System*. User programs can then choose the OS libraries to use at run time. In SPIN [2], the application developer may program the customized behaviors of the OS to match the application requirements. The customizability of these OSes at run time is limited to the foreseen and available behaviors. Customization behaviors are defined at the granularity of a policy.

In contrast, our approach operates directly on code, cross-cutting functional elements. Customizing systems code with respect to a usage context is realized automatically by a specializer. This tool-based optimization reduces code size and improves execution time.

Many OSes use configuration systems that produce customized binaries with the help of context-sensitive macros, that expand into context-specific code. The configuration system of the Linux kernel is one such example. These systems, however, are highly coarse-grained and inflexible. Loading code with macros and preprocessor directives (like `#ifdefs`) adversely affects its readability. Furthermore, it is virtually impossible to express customization behaviors with rich customization contexts.

Optimization of systems code using code generation. There have been attempts that use dynamic code generation to optimize functionalities. 'C, by Engler *et al.* [7] is one example of these. Auslander *et al.* [1] have presented an approach for the dynamic compilation of C code. Our approach, unlike these, uses existing, mature code bases and optimizes them at run time using specialization declarations. Furthermore, tool-based specialization uses a clearly defined architecture that we have described in this paper for local systems and described elsewhere [3] for a distributed environment, such as mobile networks.

Some recent specialization efforts have used Tempo, through its low-level interface to specialize systems code [17, 18, 19, 22] and without declaration support. However, these works cannot be used dynamically in an automated fashion, as is made possible by our tool-based approach.

Protocol stack optimization. Some earlier works have also examined techniques to optimize protocol stacks. X-kernel [11] is an object-based framework for implementing network protocols. With the help of well-documented interfaces, it enables developers to implement protocols and create processing chains rapidly. In the Synthesis kernel [23], protocol stacks are flattened and optimized using aggressive function inlining. Plexus [9] allows the creation of application specific protocols in a typesafe language. These can be inserted in the kernel dynamically. Some works, such as the one by Van Renesse *et al.* [24], use optimizations specific to high speed networks to attain certain thresholds of performance, such as lowering packet latency below a particular value. Our approach, as contrary to these, applies to already existing mature implementations of protocol stacks. It uses the leverage of evolved OS code and optimizes it in a way that entails negligible modifications in itself and minimal modifications in applications that utilize it.

6 Conclusion and future work

In this paper, we have described an approach to combining the leverage of a generic protocol stack, with the footprint and performance advantages of a customized one. To achieve this combination, we use tool-based program specialization. We have implemented a facility for applications to invoke such specialization and use specialized code with minimal modifications. This implementation is optimized for local specialization, but has been extended to specialization in a distributed environment as well [3].

Specialization of the Linux TCP/IP stack reduced the code size by a factor of 20, improved the execution speed by up to 25%, and improved the throughput by up to 21%. The portability of the approach has been demonstrated by our experiments on three architectures: PIII, Intel 486, and ARM and our perusal of FreeBSD 5.1 to establish a correlation.

Among our future projects, we intend to explore the specialization of protocol stacks in non-UNIX OSes, such as Windows. We also intend to specialize other subsystems such as memory management and file-systems.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, PA, USA, May 1996. ACM SIGPLAN Notices, 31(5).

[2] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Gün Sirer. SPIN – an extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Department of Computer Science and Engineering, University of Washington, February 1994.

[3] S. Bhatia, C. Consel, and C. Pu. Remote customization of systems code for mobile devices. Research report, INRIA, November 2003.

[4] Hewlett-Packard company Information Networks Division. *Netperf: A network performance benchmark*, February 1996.

[5] C. Consel, L. Hornof, J.L. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), September 1998.

[6] C. Consel, J.L. Lawall, and A.-F. Le Meur. A tour of tempo: A program specializer for the C language. Research Report 1299-03, LaBRI, April 2003.

[7] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 131–144, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[8] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[9] Marc E. Fiuczynski and Brian N. Bershad. An extensible protocol architecture for application-specific networking. In *USENIX Annual Technical Conference*, pages 55–64, 1996.

[10] F. Matthijs G. Denys, F. Piessens. Survey of customizability in operating systems research. *ACM Computing Surveys*, 34(4):450–468, December 2002.

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

[12] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[13] A.-F. Le Meur, J.L. Lawall, and C. Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 2003. Revised version of Towards Bridging the Gap Between Programming Languages and Partial Evaluation (PEPM'02), To appear.

[14] Anne-Françoise Le Meur. *Approche Déclarative à la spécialisation de programmes C.* Thèse de doctorat, Université de Rennes 1, France, December 2002.

[15] Toshiyuki Maeda. Safe execution of user programs in kernel mode using typed assembly language. Master's thesis, University of Tokyo, asdfasdf, 2002.

[16] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP selective acknowledgment options, October 1996. Status: PROPOSED STANDARD.

[17] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[18] G. Muller, E.N. Volanschi, and R. Marlet. Automatic optimization of the Sun RPC protocol implementation via partial evaluation. In *European Research Seminar on Advances in Distributed Systems (ERSADS)*, pages 105–110, March 1997.

[19] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–125. ACM Press, June 1997.

[20] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks, January 1984. Status: UNKNOWN.

[21] F. Noël, L. Hornof, C. Consel, and J.L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[22] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995.

[23] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[24] R. Van Renesse. Masking the overhead of protocol layering. In *Proceedings of the 1996 SIGCOMM Conference*. ACM Press, 1996.