Automatic Specialization of Protocol Stacks in Operating System Kernels

Sapan Bhatia* Charles Consel* Anne-Françoise Le Meur*
*LaBRI/INRIA
ENSEiRB
1 Ave du Dr. Albert Schweitzer
33400 Talence
France

Calton Pu†
†College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

***Abstract*— Fast and optimized protocol stacks play a major role in the performance of network services. This role is especially important in embedded class systems, where performance metrics such as data throughput tend to be limited by the CPU. It is common on such systems, to have protocol stacks that are optimized by hand for better performance and smaller code footprint. In this paper, we propose a strategy to automate this process.**

Our approach uses *program specialization*, and enables applications using the network to request specialized code based on the current usage scenario. The specialized code is generated dynamically and loaded in the kernel to be used by the application.

We have successfully applied our approach to 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 the current usage context, and to use the specialized code generated instead of its generic version. Specialization can be performed locally, or deferred to a remote specialization server using a novel mechanism [1]

Experiments conducted on three platforms show that the specialized code runs about 25% faster and its size reduces by up to 20 times. The throughput of the protocol stack improves by up to 21%.

I. Introduction

The goal of efficient data processing in protocol stacks is well-established in the networking community [2], [3], [4], [5], [6]. This is becoming increasingly more important with embedded devices becoming more and more networked, as throughput on such systems is invariably limited by the processing capabilities of the CPU.

Protocol stacks for embedded devices are thus highly customized with a view of their target applications. The customization process aims to eliminate unnecessary functionalities and instantiate the remaining ones with respect to parameters of the device usage context. This process typically consists of propagating configuration values, optimizing away conditionals depending on configuration values, *etc.* These hand-optimized protocol stacks not only save on processing steps, but also have smaller footprints, better suited to embedded systems [7]. Such strategy, however, raises a conflict between thorough specialization, to obtain significant improvements, and conservative specialization, to preserve the usage scope of the system. Indeed, many usage parameters become known at run-time, as the system is in use. As such, this situation raises the need for a tool that can be used to exploit them systematically.

Program specialization [8] has been acknowledged to be a powerful technique for optimizing systems code [9], [10]. 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 for a connection. In this context, the TCP `send` routine and dependent functions can be specialized, such that conditional tests and expressions depending on the 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, arithmetic expressions can be reduced in complexity, and conditional tests that solely depend on known variables can be evaluated early, eliminating entire branches of code. Furthermore, specialized code has better cache properties, and hence lower operational load (in cycles per instruction) because it is smaller and operates on working sets that are inevitably smaller than those of its generic version.

In this paper, we describe an approach to speeding up TCP/IP on CPU limited systems using run-time code generation with program specialization. In our approach, specialization is a continuous process, sensitive to the everchanging needs of applications. The usage contexts and specialization opportunities associated with them are defined in two phases 1) certain aspects of the system are defined as specializable by the OS developer and 2) applications are modified by programmers to specialize their functionalities before using them. The former is done via annotations written in a declarative language, aside from the target program [11], and the latter through documented system calls. Applications can trigger specialization as early as the specialization context becomes known. For instance, a specialization context can consist of the TCP MSS, the destination IP address, and the route associated with the address, all of which become defined upon the execution of a TCP handshake.

Perhaps the most significant barrier to using such an optimization approach is the fact that specialization is a heavyweight process, and can be expected to undermine the gains of using specialized code. Break-even points for the simplest specializations running on an embedded system would require applications to use specialized code for unreasonably long periods of time. We address this problem by allowing the activity of specialization to be deferred to a more powerful specialization server. Concretely, the specialization client conveys the specialization context with some key run-time state to the specialization server, and the specialization server specializes the code and returns it to the client. Our experiments show that issuing a specialization request and uploading the specialized module do not require much bandwidth in practice, in order for specialization to be effective. Furthermore, deferring specialization to a remote server obviates the need to store generic versions of code on the embedded device, as the pertinent specialized versions can be downloaded on demand. Indeed, the most ostensible improvement observed in our experiments are in code size.

The TCP/IP stack we have used in our proof-of-concept implementation and experiments is that of the Linux kernel. We have validated our effort with experiments on three platforms: a Pentium III (600MHz), an ARM SA1100 (200MHz) on a COMPAQ iPAQ, and a 486 (40MHz). Experiments conducted using this setup have shown that there is a notable code speedup, and a drastic 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 II 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 III explains how the specialization process is concretely integrated and automatically enabled. Section IV describes and analyzes the performance measurements obtained while performing various experiments. Section V presents some related work, and finally Section VI concludes.

II. Specialization of protocol stacks: concept and mechanism

The key observation that makes a protocol stack amenable to program specialization is of its mode of usage. An application that needs to exchange data over the network must create a channel (a socket) and configure it at various stages of the communication for certain properties, such as protocol versions to use, the time-to-live of packets, connection timeouts *etc.* Setting such properties causes the socket to be bound with specific functionalities, which are invoked in the process of sending and receiving data. The values of properties are usually application dependent. For example, conventionally, an HTTP server that needs to maximize its throughput is likely to use non-blocking asynchronous communication. In constrast, a server that needs to minimize request latency might use blocking synchronous communication. Similarly, applications that transfer data in bulk would favor the use of large local buffers, while an application transferring data in small transactions would try to minimise connection lifetimes.

Such sensibilities, although well defined in the process of application development, become known to the OS kernel (*i.e.*, the protocol stack) only once the application is deployed.

Using program specialization, we use these properties as *program invariants*, and use the process of configuration of properties to dynamically generate specialized code. The scope of the invariance is determined systematically at the time the OS kernel is built. The kernel is extended with routines to invalidate code or take a different course of action, if the value of a supposed invariant changes.

Central to this process of specialization, are the invariants and the specialization opportunities they entail.

A. Specialization opportunities

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 Linux code to the FreeBSD code, and considered only opportunities that exist in both code bases.

1) Sockets, UDP and IP layers: We now describe the chief categories of specialization opportunities, illustrated with representative examples.

Eliminating lookups. To be abstracted as files, sockets are implemented as entries in a special file system, and accessed through accessor functions that perform lookups and translations from one level to the other. Other than an added layer of indirection, using these cross-module functions also tends to impair caches.

The code fragment below shows the implementation of the `sendto` system call, which begins by using the `sockfd_lookup` function to fetch the pertinent `socket` structure from the *inode* corresponding to the file descriptor.

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

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. Execution paths for sending and receiving packets are highly branched with the interpretation of several levels of options. 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, and straight-lined specialized code be generated.

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 II-B. 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. Depending on the chosen buffer management strategy (linear socket buffers versus small fixed sized buffers, scatter-gather I/O versus versus block copies), memory is allocated at various points during the processing of a packet. Although the allocation and initialization of socket buffers tend to be cached in kernel caches (*e.g.,* the slab cache [12]), large bursts of data, and low memory situations can cause buffer allocation to go through the full length of the Virtual Memory subsystem (in Linux, through the slab allocator, the buddy allocator, the zone allocator and page allocation routines). We profiled the Linux kernel with an in-kernel web server subject to heavy load with 100 concurrent connections at a time, and found that some of the most called and most loaded functions deal with memory management, as shown in figure 1.

Memory management routines are also amenable to specialization. Calls to generic allocation routines (to allocate a socket buffer, for instance) can be specialized to produce routines that simply allocate a physical page and return.

```
61367 total                          0.0219
  630 tcp_v4_rcv                      0.3485
  338 alloc_skb                    1.4083
  268 schedule                        0.2018
  242 ip_rcv                         0.2224
  234 kmem_cache_alloc             2.0893
  226 tcp_rcv_state_process           0.0948
  211 sock_wfree                     2.6375
  201 parse_http_message             0.0257
  181 tcp_send_fin                    0.2514
  156 ip_route_input                  0.4239
  156 __might_sleep                    0.8864
  155 tcp_write_xmit                  0.2018
  150 tcp_parse_options               0.2679
  142 kmem_cache_free              1.7750
  142 kfree                         1.2679
  141 number                          0.1798
  140 skb_release_data             0.5469
  137 __kfree_skb                   0.5352
```

Fig. 1. Profile of an in-kernel web server subject to a continuous load of 100 concurrent requests

Furthermore, we also found it useful to add a socket option for an application to be able to commit that it will be sending data in fixed Application Data Units (ADUs). In this way, for many workloads, several conditions and predicates based on the size of the allocated buffer can be reduced. For example, in the second half of the excerpt below, 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.

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

2) TCP: As one can imagine, considering the richness and variations possible in 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))
        goto wait_for_sndbuf;
```

```
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 is invariant over a TCP connection (as is usually the case), causing all associated conditionals to be elided, and constants inlined.

Interestingly, if an application commits the size of the ADU to be a multiple of the MSS (using a socket option, as mentioned in the UDP case), 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 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 III. Assuming a constant MSS also enables us to specialize out Nagle's algorithm [13], since there are never small packets in flight.

As a side benefit, 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 some data remains to be sent*/
while (seglen > 0)
{
/* Calculate bytes to push into previous skb*/
    copy = MSS_now - last_skb_len;
/* Is there enough space in the previous skb?*/
    if (copy > 0) {
        if (copy < seglen)
            copy = seglen;
        push_into_previous(copy);
    }
    else {
        copy = min(seglen, MSS_now);
        push_into_current(copy);
    }
    seglen -= copy;
}
```

There are also several variables in the congestion control algorithms that can be used for specialization. For example, the Selective Acknowledgments (SACK) option [14] is useful in situations where multiple segments are lost in one window. An application functioning in a high-speed, uncongested local area network may wish to specialize this away.

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.*

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;
```

The code is highly branched with options being 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. 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 2 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 Silly-Window-Sindrome algorithm and the explicit specialization-time removal of optional features such as ECN and SACK are available in FreeBSD as well.

B. 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

```
do {
    if (uio == NULL) {
        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);
        /*For datagrap protocols, leave room*/
            if (atomic && top == 0 && len ¡ mlen)
        /*for protocol headers in first mbuf*/
                MH_ALIGN(m, len);
        }
        ...
}
while (!buffer_sent);
```

Fig. 2. Fast-path of the UDP send operation in BSD

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* [9]. The dynamics of establishing guards and the process of code replugging were first described by Pu *et al.* [9]. 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.

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 III), 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 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.* [9].
- 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 choose the second strategy, and 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 (NAT) rule is installed as a reinforcement to ensure delivery to the correct physical destination. Using one or the other option, and how the rule is put into effect 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 use a reinforcement based on a packet mangling rule.

III. Enabling the specialization

Before discussing the implementation of the specialization infrastructure, *i.e.*, the machinery that actually generates specialized code and loads it in the kernel of the target machine, we will describe a typical scenario to acquaint the reader with how our approach works in practice.

1) A scenario: Our specialization architecture allows specialized versions of code to be requested for a fixed set of system calls, defined at the time the OS is compiled. To simplify discussion, we will focus on the `send` system call, which is the most common function used to submit data to the protocol stack.

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 socket 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 socket descriptor allows for multiple versions of the `send` system call to be used with the same socket descriptor.

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.

A. Describing the specialization opportunities to the specializer

The program specializer we used in this project is the Tempo C Specializer [15]. 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 [11]. 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 3. 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 is done, the specialization may be performed as soon as the specialization context (*i.e.*, the values of the invariants) is made available.

B. Performing specialization: Local or Remote?

The most important issue to address when specializing code for CPU limited systems is where to execute the process of specialization, as it can be expected to consume a lot of resources.

We have implemented two versions of our specialization infrastructure, one of which loads and executes the program specializer, and the compiler to compile the generated code, locally. The other version, described in detail in a technical report [1] requests specialized code to be generated by sending the specialization context used and downloading the specialized code generated. This is the approach of choice for embedded network systems. In the following subsections, we give a short description of both approaches.

Original 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) {
    ...
 };
```

Fig. 3. Specialization declarations

C. Specializing locally

Specializing locally may be desirable in cases when a reasonably powerful server needs to maximize its efficiency transferring over a high speed link, such as 1 or 10 Gigabit.

The most important implementation issue here is making the specialization context, consisting of invariant properties, available to the specializer. This is significant as these values are available in the address space of the kernel, and cannot be accessed by the program specializer, which runs in user address space.

In the local case, we solve this problem by runing the specializer as a privileged process and giving it direct access to kernel memory. The technique used to accomplish this is described in detail by Toshiuki [16].

D. Specializing remotely

Being able to specialize remotely is crucial for low-end systems such as PDAs, as running specialization on them would consume scarce memory and storage resources as well as take a long time to complete. In remote customization, the OS kernel on the target device for which the specialized code is needed packages the specialization context and key run-time information and sends it to a remote specialization server. The context and run-time information are used to emulate the device run-time environment on the server, and the specializer run to use this environment in part to generate the specialized code. The specialized code is finally sent to the device.

IV. 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.

Specialization was performed remotely, as described in our tech report [1] for all three architectures, on a fast server and over a 10Mbps[1] wireless lan. 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.

A. 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 two stages:

- Measuring code speedup. We send a burst of UDP packets and record the number of CPU cycles taken by the pertinent code (*i.e.,* the socket, UDP and IP layers) in the unspecialized and specialized versions. These measurements were performed in-kernel.
- Measuring throughput improvement. The *Netperf benchmark suite* [17] was used to find the impact of specialization on the actual data throughput of the TCP/IP stack. 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. This measure also gives an indication of how much CPU resource is freed up, as the additional CPU cycles now available may be used for activities other than data transmission.

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

B. Size and performance of specialized code

Figure 4(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 4(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.

Figures 5(a) and 5(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 5(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(d) and 5(e) 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, in the setup described earlier. 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.

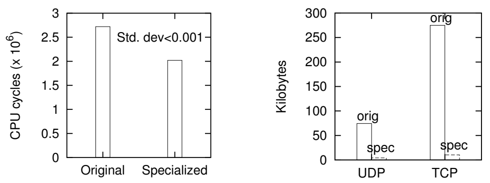
V. Related Work

Optimizing protocol stacks has been a consistent area of research in network systems. Protocol stacks have been optimized using various approaches over the past two decades. And even today, work continues on flexible OS architectures that facilitate fast networking. We see our work as fitting in the broad scope of these efforts, with a specific motivation to automate optimization for embedded network systems.

Mosberger *et. al* [18] list some useful techniques for optimizing protocol stacks. Our approach captures most of the optimizations described in this work. *Path-inlining* comes for free, as the specialization context directly identifies the fastpath associated with operations, bringing all code that goes into it together. Function *outlining* works in the same way, as unneeded functions are specialized away from the code used. Function *cloning* can happen when a function is fully static and determined at specialization time.

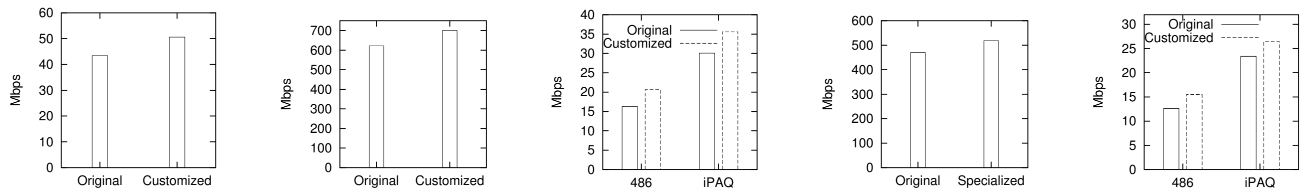
X-kernel [19] is an object-based framework for implementing network protocols. With the help of well-documented interfaces, it enables developers to implement protocols and create packet processing chains rapidly. Run-time code generation has been known to yield impressive performance gains in prior works such as DPF [2] and Synthesis [20]. Synthesis also used aggressive inlining to flatten and optimize protocol stacks. Plexus [21] allows the creation of application specific protocols in a typesafe language, which can be dynamically inserted into kernels. Prolac [22] is a statically-typed, object-oriented language to implement network protocols that deviated from theoretical models for protocol definition and

[1]Note: this was used by the specialization extensions to procure specialized code, and not by the application to transfer data.



(a) Performance (b) Code size

Fig. 4. Original vs. specialized code: performance and size



(a) UDP, PIII, 64b (b) UDP, PIII, 1Kb (c) UDP, 486/iPAQ, 1Kb (d) TCP, PIII (e) TCP, 486/iPAQ

Fig. 5. Original vs. specialized code: maximum throughput on the PIII, 486 and iPAQ for UDP ($\pm5\%$ at 99%) confidence and TCP ($\pm2.5\%$ at 99% confidence)

	Cold cache	Warm cache
UDP	685ms	285ms
TCP	791ms	363ms

Fig. 6. Specialization overhead

focussed on readability and ease of implementation. These efforts, however, are orthogonal to our work as our aim is to reuse existing protocol stack implementations in an efficient way, as opposed to encoding new ones. 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.

VI. 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 automatic 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 [1].

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 are also working on extending the specialization interface to empower applications to express more powerful specialization predicates, to further optimize operation.

References

[1] S. Bhatia, C. Consel, and C. Pu, "Remote customization of systems code for mobile devices," LaBRI," Research Report, February 2004.

[2] D. Engler and M. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *SIGCOMM Symposium on Communications Architectures and Protocols*. Stanford University, CA: ACM Press, Aug. 1996, pp. 26–30.

[3] C. Thekkath and H. Levy, "Limits to low-latency communication on high-speed networks," *ACM Transactions on Computer Systems*, vol. 11, no. 2, pp. 179–203, May 1993.

[4] H.Fujinoki, M.Sanjay, and C.Shah, "Web file transmission by object packaging: Performance comparison with http 1.0 and http 1.1 persistent connection," in *Proceedings of the 28th International Conference on Local Computer Networks*, Sept. 2003.

[5] E. P. Marketos, "Speeding up tcp/ip: Faster processors are not enough," in *Proceedings of the 21st IEEE International Performance, Computing, and Communications Conference*, Apr. 2002.

[6] D. P., "Operating system support for high speed networking," in *Communications of the ACM*, vol. 39, no. 2, Sept. 1996, pp. 41 – 51.

[7] T. Fischer, "Optimizing embedded Linux," May 2002.

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

[9] 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*, Copper Mountain Resort, CO, USA, Dec. 1995, pp. 314–324.

[10] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet, "Specialization tools and techniques for systematic optimization of system software," *ACM Transactions on Computer Systems*, vol. 19, no. 2, pp. 217–251, May 2001.

[11] A.-F. Le Meur, J. Lawall, and C. Consel, "Specialization scenarios: A pragmatic approach to declaring program specialization," *Higher-Order and Symbolic Computation*, 17 (1) (2004).

[12] J. Bonwick, "The slab allocator: An object-caching kernel memory allocator," in *USENIX94*, 1994.

[13] J. Nagle, "RFC 896: Congestion control in IP/TCP internetworks," Jan. 1984, status: UNKNOWN.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "RFC 2018: TCP selective acknowledgment options," Oct. 1996, status: PROPOSED STANDARD. [Online]. Available: ftp://ftp.internic.net/rfc/rfc2018.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc2018.txt

[15] C. Consel, J. Lawall, and A.-F. Le Meur, "A Tour of Tempo: A Program Specializer for the C Language," *Science of Computer Programming*, 2004, to appear.

[16] T. Maeda, "Safe execution of user programs in kernel mode using typed assembly language," Master's thesis, University of Tokyo, 2002.

[17] H.-P. company Information Networks Divis ion, *Netperf: A network performance benchmark*, Feb. 1996.

[18] D. Mosberger, L. Peterson, P. Bridges, and S. O'Malley, "Analysis of techniques to improve protocol processing latency," in *SIGCOMM Symposium on Communications Architectures and Protocols*. Stanford University, CA: ACM Press, Aug. 1996, pp. 26–30.

[19] N. C. Hutchinson and L. L. Peterson, "The *x*-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, Jan. 1991.

[20] C. Pu, H. Massalin, and J. Ioannidis, "The Synthesis kernel," *Computing Systems*, vol. 1, no. 1, pp. 11–32, Winter 1988.

[21] M. E. Fiuczynski and B. N. Bershad, "An extensible protocol architecture for application-specific networking," in *USENIX Annual Technical Conference*, 1996, pp. 55–64. [Online]. Available: citeseer.nj.nec.com/fiuczynski96extensible.html

[22] M. F. K. Eddie Kohler and D. R. Montgomery, "A readable tcp in the prolac protocol language," in *SIGCOMM99*, 1999.

[23] *SIGCOMM Symposium on Communications Architectures and Protocols*. Stanford University, CA: ACM Press, Aug. 1996.