Remote Customization of Systems Code for Embedded Devices*

Sapan Bhatia and Charles Consel
LaBRI/INRIA
ENSEIRB, 1 Ave. de Dr. Albert Schweitzer
33402 Talence
France
{bhatia,consel}@labri.fr

Calton Pu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
USA
calton@cc.gatech.edu

ABSTRACT

Dedicated operating systems for embedded systems are fast being phased out due to their use of manual optimization, which provides high performance and small footprint, but also requires high maintenance and portability costs every time hardware evolves.

In this paper, we describe an approach based on customization of generic operating system modules. Our approach uses a remote customization server to automatically generate highly optimized code that is then loaded and executed in the kernel of the embedded device. This process combines the advantages of generic systems software code (leveraging portability and evolution costs) with the advantages of customization (small footprint and low overhead).

We have validated our customization infrastructure with a case study: the TCP/IP stack of the Linux kernel. We analyzed the performance and size of the customized code generated on three platforms: a Pentium III (600MHz), an ARM SA1100 (200Mhz) on a COMPAQ iPAQ, and a 486 (40MHz). The customized 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%.

Categories and Subject Descriptors

D.4 [**Software**]: Operating Systems—*Organization and Design*; C.3 [**Special-purpose and Application-based Systems**]: [Real-time and embedded systems]

General Terms

Performance, Design, Experimentation

Keywords

Remote Customization, Remote Specialization, Specialization Server

1. INTRODUCTION

Traditionally, embedded systems have made use of operating systems (OSes) that are either proprietary to some device manufacturer (*e.g.,* Palm personal digital assistant), or dedicated to embedded devices (*e.g.,* QNX). In both situations, OSes are highly customized for the specific hardware features, the environment characteristics, and the functionalities needed by the 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, replacing expensive indirect function invocations with direct ones *etc.* As a result, the key systems components such as memory management and process scheduling are heavily optimized in time and/or space. For a device manufacturer OS, the customization process can be pushed further because the target is even narrower than for an OS for embedded devices.

Still, this pragmatic approach to developing OSes falls short of keeping pace with the rapid evolution of hardware features, environment characteristics and functionalities of new applications. Indeed, customization is generally performed manually. The process is tedious, error-prone, and causes a proliferation of OS versions. Also, more fundamentally, most OSes for embedded systems are incompatible with mainstream desktop OSes, creating a separation between these two worlds. Because of these drawbacks, proprietary and custom operating systems are slowly being phased out [5].

In recent years, mainstream OSes like Linux and Microsoft Windows have been emerging as the industry standards for embedded systems. A key difference in these OSes is that they are general purpose. As such, their highly generic design results in large code size and many levels of performance overhead. Nevertheless, the use of these OSes is indispensable for devices to be standards-compliant and interoperable at the levels of hardware and software.

In this paper, we propose an approach to reconciling custom OSes and general-purpose OSes. This approach consists of automatically customizing general-purpose systems

*This work has been partially supported by the Conseil Régional d'Aquitaine and by DARPA, DoE, NSF under grants CCR-0121643, CCR-0219902, IIS-0242397, Intel and HP.

EMSOFT'04, September 27–29, 2004, Pisa, Italy.
Copyright 2004 ACM 1-58113-860-1/04/0009 ...$5.00.

modules for a given usage context. Upon request, modules are remotely customized on a server. Once produced, the customized module is loaded into the kernel of the embedded system. Customized code is faster and occupies less space. Because it is performed outside the embedded system, customization does not incur any significant overhead. Because customized modules are produced and loaded on-demand, the embedded system need no longer store large, generic code bases implementing a wide range of features.

To realize our approach, we have developed a customization infrastructure, composed of an architecture, specific mechanisms, and a customization engine. When an application running on a device needs to invoke a *customizable system call*, it issues a customization request as early as the customization values are known. The *customization context* is sent via a dedicated channel to a *remote customization server*. This server invokes a customization engine with both the corresponding systems module and the customization context. Customization automatically produces the highly optimized module that is then sent to the embedded device to be loaded and used.

We have validated our customization infrastructure with a case study: the TCP/IP stack of the Linux kernel. This case study has demonstrated the benefits of our approach in terms of code size and performance on three different hardware platforms.

Specifically, our experiments show that customized code is notably improved in terms of both execution time and size. For the UDP subsystem, the code size produced is less than 5% of the size of the generic 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 running at 600MHz and the local throughput for 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 presents the key components of our customization infrastructure. In Section 3, our infrastructure is validated with a case study, namely, the customization of the TCP/IP subsystem of the Linux kernel. Section 4 details the experiments we conducted to assess the performance benefits of our approach and the results of the performance analysis. Section 6 discusses related work. Section 7 concludes and explores future work.

2. CUSTOMIZATION INFRASTRUCTURE

In this section, we first give an overview of our customization infrastructure (see Figure 1) by examining the various steps involved in requesting and utilizing a customized module. This overview is completed by a detailed presentation of three key parts of our infrastructure, namely, the context manager, the code manager, and the customization component. Finally, we discuss the issues related to potential latency of system calls caused by on-demand customization.

2.1 A Complete Scenario

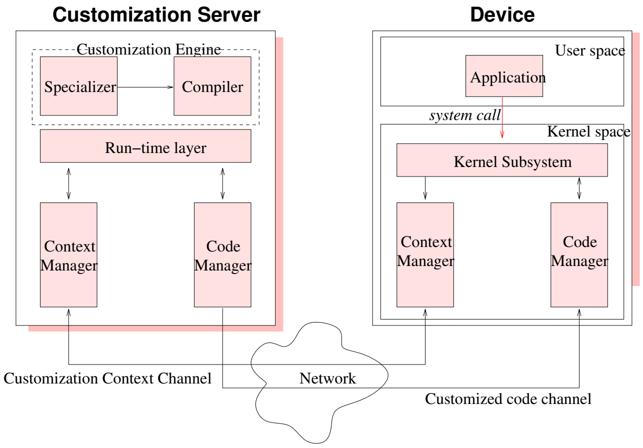


Figure 1: Remote customization infrastructure

The customization infrastructure is defined with respect to a set of system calls, said to be *customizable*. Non-customizable system calls are handled in a standard way through their generic implementations. A global customization interface includes an entry for each customizable system call aimed to launch its customization for a given context. For example, assuming the socket `send` system call is customizable, an application can request a customized version of it by invoking `do_customize_send` with the file descriptor, the destination address of the packets, the protocol to use and the associated flags, and socket options. This request is issued as soon as these customization values become known. This invocation returns a *token* that is then used by the application to refer to the version of the system call, customized for the specific context. Invoking the customized version of the `send` system call is done via `customized_send` that takes three arguments less than the former: the customization parameters. However, it takes one additional argument, namely the token. Let us now examine how each component of our customization infrastructure, depicted in Figure 1, contributes to request, produce, install and utilize customized systems code.

An entry of a customization interface corresponds to a macro (*e.g.*, `do_customize_send`) that expands into a unique system call (`sys_specialize`) that is passed the system call number and the customization arguments. The customization kernel subsystem passes the customization values to the *context manager*. The customization values pertinent for the system call are extracted and used to issue a lookup request to the *code manager*. In the send example, the values such as the destination address and the protocol number are collected by the context manager. Upon request, the code manager looks up the cache of customized code. A cache hit produces the address of the customized version of a system call for a given context. This address is used to create a new entry in a *local system call table* associated with the corresponding process. The entry number is returned as the customization token to the context manager, which returns it to the application through the system call. It serves to later invoke the customized system call (*e.g.*, via `customized_send`).

If the cache does not contain the requested customized system call, then the context manager issues a customization request to the server-side context manager, via the context channel, as shown in Figure 1. The server-side context manager forwards the request to the customization subsystem. This subsystem consists of a run-time layer and a customization engine. The run-time layer is responsible for "virtualizing" the device-side kernel memory with the received customization values so that the system call code can be customized remotely. This operation, further detailed in Section 2.4.2, enables the customization engine to then run as if it resided on the device. Once the customized code is automatically produced, it is passed to the server-side code manager, which transmits it to the device-side code manager, via the customized code channel. A new entry is created in the cache of customized code. The process then proceeds as in the case of a cache hit, updating the process system call table and returning a token to the application.

When a customized system call is invoked by the application, because the customization has been issued earlier, it is hopefully already loaded in the cache. If not, the application waits until customization is completed and the cache is loaded – the potential system call latency is later discussed in Section 2.5. At this point, the invocation of the corresponding customization interface entry (*e.g.*, `customized_send`) branches to the code address of the system call table of the application process, indexed by the customization token.

Lastly, when the customized system call is no longer needed, the application can release the token by passing it to the function `cancel_customize`. This operation can free the corresponding memory space, if no other process points to this token and/or the device runs low in memory.

Let us give further details and practical insights on the key parts of the customization infrastructure.

2.2 Context Manager

On the client side. The context manager consists of extracting the customization values from the arguments of the application customization request. These values have been identified by the systems programmer who made the system call customizable (discussed in Section 2.4.1), and compiled into the context manager for the pertinent system call. This process is a trade-off in that too fine-grained customization values may cause too many customization versions; but, too coarse-grained customization values may lead to under-optimized code [12]. In the send example, we decided to only include in the customization context the values that are either guaranteed, or do not tend to change within a connection, such as the socket descriptor, the destination address *etc.* This is discussed in more detail in Section 3.

The context manager can also be configured to keep the number of specialized versions of a given module bounded. It may do so using one of several policies, such as gradually ignoring customization parameters to reduce the specificity of implementations.

Finally, notice that the context manager could also receive an empty customization context. Such context would lead the customization server to supply the generic version of the functionality to the device. It is worth mentioning that although this version will yield the same level of performance, it will be smaller in size, as even a null specialization context implicitly implies the exclusion of code features that are not used by the requested functionality. Thus, the resulting code can be seen as the result of a simple dependency analysis where only code that the customized functionality strictly depends upon is returned.

On the server side. The context manager processes the customization values in preparation for the customization phase. This task consists of storing these values in a customization table. The index in this table is a hash number corresponding to the original kernel memory address of the customization value. The customization table is used by the customization run time as explained later.

2.3 Code Manager

On the client side. The code manager maintains a cache of customized code indexed by the system call number and the customization context. Notice that this cache is shared across the application processes of the device. The code manager runs in kernel mode and thus directly loads the customized code into the kernel, without using intermediate storage or buffering.

Similar to the policies in the context manager, the code manager can be configured with a cache-replacement policy. An LRU policy would cause the Least Recently Used entry in the cache to be reused if the number of customized versions for a particular system call exceeded a threshold.

Regardless of the code manager policy, it should be noted that the customized code produced is several times smaller than the generic code, as shown by our experimental results (Section 4). As a result, the system can tolerate up to 20 customizations before it occupies as much space as the original code would have on the device.

On the server side. The code manager simply transmits the customized code to the device via the customized code channel.

2.4 Customization Component

The customization component consists of a customization engine and a run-time layer.

2.4.1 Customization Engine

Program specialization [10] is used to perform the customization of systems code. Specialization tools have been successfully used in a number of systems projects to optimize critical systems code [14, 13]. 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.

Specialization consists of two phases: for a given program and a description of its customization context, a binding-time analysis determines all of the computations that can be performed at customization time. These computations solely depend on values known at customization time. Then, for given customization values, the program is specialized by executing the customization-time operations and producing code corresponding to computations that depend on non-customization values.

In this project, we have used the Tempo C specializer [4] to perform specialization. Unlike other specializers, in Tempo, the customization context is specified by the programmer using a high-level declarative language [12]. A declaration is introduced for each program entity to be considered for customization.

2.4.2 *Run-Time Layer*

The run-time layer plays a key role in enabling both the reuse of an existing program specializer and the specialization of kernel code remotely. To do so, the run-time layer handles (1) the access to kernel memory and (2) the execution of *customization-time functions.*

Access to kernel memory. It relies on the customization table filled by the server-side context manager, introduced earlier. Then, in the course of specializing some systems code, a machine instruction may require dereferencing a kernel memory address. This dereferencing raises an exception handled by the run-time layer. Its task is to interpret the machine instruction that caused the exception. This interpretation consists of emulating the effects of the machine instruction, replacing the kernel memory address by the corresponding address in the customization table. For example, consider the instruction `MOV al,c021cca0`. The run-time layer computes the corresponding address of `c021cca0` in the customization table and then loads it in register `al`. Once this interpretation is completed, the execution of specialization is made to resume at the following instruction. A future optimization for this process will be to create a direct mapping between the target memory addresses and the specialization context table in the specializing process, so as to remove this layer of emulation. In this way, the above translations would be handled by the memory management unit.

Execution of customization-time functions. It occurs whenever a function can be completely executed during customization. That is, it solely depends on values known at customization time. For example, in the send case, function `sockfd_lookup` only depends on the available value of the socket descriptor. Such functions are detected by the analysis phase of specialization. A call to these functions, in the code to be specialized, is replaced by a remote-procedure call to the device-side kernel.

2.5 Customization Latency

Performing customization on-demand may increase the latency of system calls, as there is some processing that must be performed before normal operation can resume. We have quantified this potential latency in Section 4 for our current setup. Evidently, it directly depends on the time required to customize code on the server.

Experiments show that such a latency may be eliminated in practice because the customization context often becomes available early enough, so that the customized code is produced and loaded by the time the customized system call is invoked. In the following, we propose strategies to amortize or eliminate this latency.

Customizing in advance. Customization may be performed in advance, anticipating that a system call be used in a particular context. For example, when an FTP session is opened, under normal circumstances, customization would be invoked when the *connect* system call is called to open a TCP connection. To reduce latency, customization can be performed between the time that the destination IP address and port are specified, and when the user has entered the login and password.

Using conservative customization contexts. When making system calls customizable, the systems programmer may define a conservative customization context to enable customization to be triggered earlier. As a byproduct, this strategy should increase sharing of customized versions across the device applications.

Exploiting the intrinsic latency of system calls. Intuitively, customization can be triggered when some kind of session is to be opened (*e.g.,* socket `connect`). At this stage, parts of the customization context have become known. We can exploit the time taken by the system call to actually create the session to perform the customization. As an example of this latency, the time taken for a TCP three-way handshake is twice the round-trip time (RTT) between two hosts. In wide area networks, this can range from a few milliseconds to the order of one or two seconds over certain radio links such as GPRS.

3. CASE STUDY: CUSTOMIZATION OF LINUX TCP/IP

To validate our approach and to assess its benefits, we have applied our customization infrastructure to the TCP/IP stack of the Linux kernel (version 2.4.20). This subsystem consists of nearly 70,000 lines of code, and can take up between 500 kilobytes and 1 megabyte of space in the kernel binary.

In this section, we discuss the customization opportunities in this subsystem and present an example of customization using a code fragment from the socket interface to the TCP/IP stack.

3.1 Customization Opportunities

Identifying specialization opportunities 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.

We identified several customization opportunities in the TCP/IP subsystem that form part of the *customization context.* Constancy in the relationship between certain entities, such as file descriptors and low level socket structures can be used to customize the code, avoiding lookups of low-level socket structures. Socket options that are usually interpreted can be assumed to be constant over the lifetime of a data transfer session, and inlined in the code[1]. Given these opportunities, the customization engine automatically performs the following optimizations:

- Protocol layers are flattened through function inlining. This means that data is processed by a linear stream of code instead of being transferred from one module to the other.

- Routing decisions are hoisted and factorized, as in many cases, routes can be determined at customization time from known connection properties.

- Buffer allocations are optimized with the knowledge of certain sizes of low-level socket buffers that depend on the type of I/O (blocking or non-blocking) and application data unit (ADU).

- Loops are both unrolled and simplified, using constants such as the ADU transferred and the maximum segment size associated with a TCP connection, which becomes known during the TCP handshake.

[1] A detailed presentation of TCP/IP customization is presented in a technical report [2].

```
asmlinkage long sys_sendto(int fd, void * buff, size_t len,
        unsigned flags, struct sockaddr *addr, int addr_len)
{

    struct socket *sock;
    char address[MAX_SOCK_ADDR];
    int err;
    struct msghdr msg;
    struct iovec iov;

    sock = sockfd_lookup(fd, &err);                  /* 1 */
    if (!sock)                                        /* 2 */
        goto out;
    iov.iov_base=buff;
    iov.iov_len=len;
    msg.msg_name=NULL;                                /* 3 */
    msg.msg_iov=&iov;
    msg.msg_iovlen=1;                                 /* 4 */
    msg.msg_control=NULL;                             /* 5 */
    msg.msg_controllen=0;                             /* 6 */
    msg.msg_namelen=0;                                /* 7 */
    if(addr)                                          /* 8 */
    {
        err = move_addr_to_kernel(addr, addr_len, address);
        if (err < 0)
            goto out_put;
        msg.msg_name=address;
        msg.msg_namelen=addr_len;
    }
    if (sock->file->f_flags & O_NONBLOCK)            /* 9 */
        flags |= MSG_DONTWAIT;
    msg.msg_flags = flags;
    err = sock_sendmsg(sock, &msg, len);              /* 10 */
    ...


}
```

Figure 2: The sendto system call

- Conditionals are eliminated, with the knowledge of certain socket options and flags, such as asynchronous or synchronous transfer, the keep-alive option, *etc.*

- *etc.*

3.2 An Example of Customization

We illustrate how customization opportunities are exploited in practice, using a code fragment from the implementation of the socket interface to the TCP/IP stack, displayed in Figure 2. This code fragment shows variables known at customization time, as well as a function call that solely depend on customization values (*i.e.*, a customization-time function). These customization elements are invariant through the scope of the customization (*i.e.* while the corresponding token is valid), and are underlined.

As illustrated in Figure 2, the customization context includes the file descriptor passed to the system call, the flags, the destination address, and the length of the destination address.

Let us now examine how the customization engine (*i.e.*, the specializer) uses this context to optimize the code shown in Figure 2. Only the program lines impacted by specialization have been numbered.

- Line 1. The call to the `sockfd_lookup` function is factorized out and reduced at customization time. The resulting value is inlined into the code. Looking at the implementation of this function, we find that it has its roots in the filesystem subsystem, and can be expected to hinder the instruction and data caches. Recall that such a function call translates into a remote-procedure call from the customization server to the device.

```
asmlinkage long sys_sendto(void * buff, size_t len)
{
    struct iovec iov;
    iov.iov_base=buff;
    iov.iov_len=len;
    msg.msg_iov=&iov;

    {
        /* sock = 0x3f04bb00 flags=56 ... */
        { ... /* sock_sendmsg inlined */ }
        { ... /* inet_sendmsg inlined */ }
        { ... /* tcp_sendmsg inlined */  }
    }
    ...
}
```

Figure 3: The customized sendto system call

- Line 2. The conditional is eliminated, since it depends on the socket structure `sock`, which is known at customization time.

- Lines 3-7. Some assignments are factorized out and memorized at specialization time, as they once again depend on customization values. It is worth mentioning that such optimizations also reduce the amount of data to be passed through from one layer to the other.

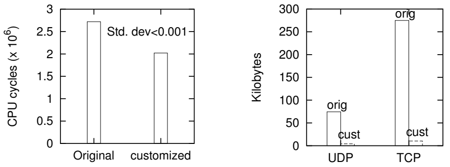
- Lines 8-9. The conditionals are also eliminated as they depend on customization values.

Finally, control is transferred to the `sock_sendmsg` function which, in the original code, goes on to call an indirect reference to a function. This indirection depends on the protocol (UDP or TCP) being used. In the customized code generated, the function is directly inlined, depending on the protocol specified in the customization context. A sugared version of the generated code is shown in Figure 3.

It is worth mentioning that, should at any time, an assumption used to generate the specialized code ceased to be valid, this code would be rendered invalid as well. Although the code has been specialized in such a way that 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. This is done using code guards [14]. The dynamics of establishing guards and the process of replugging has been described in considerable detail elsewhere [14].

4. PERFORMANCE ANALYSIS

In this section, we present the results of a series of experiments conducted on our customization infrastructure as part of the case study described in Section 3. Our setup consisted of three target devices: a Pentium III (PIII, 600MHz, 128MB RAM), a 486 (40MHz, 32MB RAM) and an iPAQ with an ARM SA1100 (200MHz, 32MB RAM). We used version 2.4.20 of the Linux kernel for our implementation and all our experiments. We first describe the experiments conducted and then go on to present the results.



(a) Perf.

(b) Code size

Figure 4: Original *Vs* Customized code: Performance and Size

4.1 Experiments

The experiments conducted compare the performance of the original TCP/IP stack to that of the customized code produced for performing basic data transfer. The measurements were carried out in 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 non-customized and customized versions. These measurements were performed in-kernel.

Measuring throughput improvement. The *Netperf benchmark suite* [3] was used to find the impact of customization 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 non-customized stack, to a modified version using the customized code produced by the customization engine. The latter was modified to use the customization interface.

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

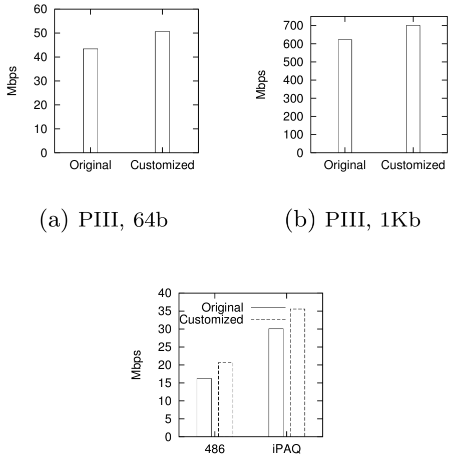
4.2 Size and Performance of Customized Code

Figure 4(a) compares the number of CPU cycles consumed by the Socket, UDP and IP layers before and after customization. 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 customized 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 customized code is a pruned and optimized version of the original code for a given customization context. As can be noticed, the customized 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 customization, 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%.

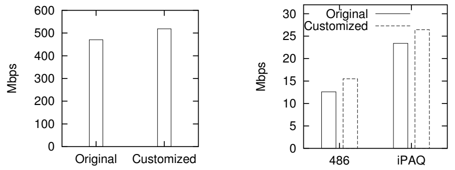


(a) PIII, 64b

(b) PIII, 1Kb

(c) 486/iPAQ, 1Kb

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



(a) PIII

(b) 486, iPAQ

Figure 6: Maximum Throughput of TCP on the PIII, 486 and iPAQ ±2.5% @ 99% confidence

Figures 6(a) and 6(b) show a comparison between the throughput of the Socket, TCP and IP layers before and after customization, 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.

We observe a general trend that the throughput improvement decreases as the processor gets faster. To understand this, let us examine the kind of operations involved in both the customizable code and the uncustomizable code. The latter code is mainly dominated by operations that are insensitive to clock rate increase, like memory transfers and device overhead. Consequently, the ratio of the customizable code and the uncustomizable code tends to diminish as the clock rate increases.

Finally, Figure 7 shows the overhead of performing customization with the current version of our customization engine over an 802.11 (10Mbps) wireless LAN. It should be noted that the current version of our customization engine is assembled from components that are implemented as sep-

	Cold cache[1]	Warm cache
UDP	685ms	285ms
TCP	791ms	363ms

Figure 7: Total customization overhead including network transfer time.

arate programs, running as independent processes. Also, they are reloaded into memory every time customization is performed. We are working on merging these components, in particular the specializer and compiler and making the customization engine a constantly running process. We expect these changes and other optimizations, such as using precompiled headers, to improve the performance of the customization engine dramatically. Indeed, the overhead is presently dominated by these factors.

For the sake of completeness, if we assume that the above mentioned inefficiencies are eliminated and that the customization server is sufficiently powerful, removing the bottleneck to the data transfer over the network, the overhead is given by the following expression:

$$RTT * (\chi + \lceil codesize/TCPwindowsize \rceil)$$

where χ corresponds to the total number of invocations of customization-time functions (in the network stack case, only one). Our experiments have shown that the code produced is small enough to fit in one TCP window (the maximum size of code produced in the TCP/IP case being about 10 kilobytes). Thus, in an amortized sense, the overhead can be said to be given by the following simplified expression: $RTT * (\chi + 1)$.

5. DISCUSSION

In this section, we discuss two key issues required to scale up the application of our approach to an operating system.

5.1 Applicability of our approach

The application of program specialization to OS subsystems is not new. Pu *et. al* have demonstrated the successful application of specialization to two important OS subsystems, namely, filesystems and signals [14, 13]. In principle, specialization can be applied to any subsystem which can be invoked as a *closure* of a set of configuration values. In filesystems, this closure is created when a file is opened, and associated with a file descriptor. When an operation is invoked on this file descriptor, it is carried out in the context of the values saved in the associated closure. In the network subsystem, the socket descriptor usually plays the role of identifying a closure. One of the problems encountered in these previous works [14, 13] was the lack of such a relation in the signals subsystem. The problem was solved by adding an identifier, `last_sig_to`, in the task structure of every process to represent the context which was used for specialization.

We generalize this solution by associating specialization with a specialization token. The context referred to by this token, which is specified at the time the system is engineered for specialization, is collected by the context manager at specialization time.

Note that although we provide the provisions necessary to make any given subsystem amenable to specialization, the actual activity of specialization is not trivial, and needs to be considered separately for every subsystem in question.

5.2 Module dependencies

Modules in OS components often have dependencies that mandate that they have certain features in order to interoperate. The question that arises in the context of our work, then, is whether specialization may break interoperability.

One aspect of the question is addressed by program guards, discussed earlier. If a given module were to access a feature of another module that has been specialized out, then it would hit upon the corresponding program guard installed for the invariant that was used to specialize the feature out. This situation, in effect, makes it critical that a thorough analysis of the module to be specialized be done, and that all possible violations of the invariant be protected by guards.

The other aspect of this question, which involves the composition of specialized components in operating system has not been addressed in the literature. Dominic Duggan provides some insights into this issue in his work on the *Type-based hot swapping of running modules* [6]. This issue, however, is well beyond the scope of this paper, and we leave it open for future work.

6. RELATED WORK

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. In a survey on such customizability, Denys *et. al* [9] classify such customizability on two bases: (1) The initiator of adaptation (human, application or OS) and (2) The time of adaptation (at compile time or run time). Our customization infrastructure performs application-driven customization at run time. In this section, we discuss works in these categories and then go on to discuss other approaches.

Application-driven and run-time adaptation. Many approaches are aimed to provide a fixed set of behaviors that can be selected at run time by the applications. These behaviors are developed by the systems programmer, and are supported by various interfaces and mechanisms. Let us briefly present three projects along this line.

Exokernel, introduced by Engler *et. al* [7] tries to eliminate all kernel abstractions and almost lower the kernel interface to the bare hardware. Each customized behavior corresponds to a systems program; it is introduced as a special *Library Operating Systems.* User programs can then choose the OS libraries to use at run time. The Kea project [16] introduces customized behavior through a *portal.* Depending on the portal an application uses, the kernel decides which implementation of the requested service to use. Like the Exokernel, Kea requires the customized behaviors attached to portal to be developed by a systems programmer. In SPIN [1], the application developer may program the customized behaviors of the OS to match the application requirements.

Other approaches. The VINO project [15] explores the general purpose automatic approach for customization. VINO automatically adapts to newly arising situations based on the periodic retrieval of statistics maintained by each sub-

[1] **Note:** The cache here has no relation to the customized code cache in the client-side code manager.

system and through traces of requests and results. This project did not lead to an implementation. One could imagine that such information could be used similarly to drive our customization infrastructure. OSKit is used to produce customized OSes [8]. It consists of a framework and a module library with interfaces that are used to implement a specific OS.

Another possible criterion in taxonomy, valuable in our context is *what* the customization operates on. Most efforts to customize OSes operate on *functional elements*, defining policies for scheduling [11], efficient implementations of subsystems [7] *etc.* Our customization infrastructure, on the other hand, operates directly on code. In this way, customization can cross-cut functional elements. This is particularly useful when aiming to reduce the size of the system footprint, since narrowing the customization context reduces the code size in proportion. Many OSes use configuration systems that produce customized binaries with the help of context-sensitive macros, which 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.

In most adaptive systems, adaptation of mechanisms and policies are carried out on the same system they reside on. With our remote customization infrastructure, we separate out this adaptive step to be executed on a powerful server. This separation is indispensable in carrying out any non-trivial run-time customization for a device with limited resources. Indeed, on the device, the customization process can incur significant overhead both in space and time. As the gap between the capabilities of mobile systems and mainstream servers increases, this separation becomes increasingly crucial.

7. CONCLUSION AND FUTURE WORK

Embedded systems use manually customized systems code due to the performance requirements of small footprint and low overhead. One of the growing concerns in embedded systems is the trade-offs between these performance requirements and frequent porting and maintenance requirements, due to the increasing speed of hardware evolution. At the same time, using generic systems code is an alternative that suffers from an increasing footprint and decreasing performance, due to added functionality that is unnecessary for embedded systems. In this paper, we describe an approach based on customization of generic systems code that combines the leverage of generic code with the performance and footprint advantages of customized code.

Our approach uses *program specialization*, in particular, tool-based specialization. We have created a customization infrastructure, consisting of a customization server, context managers, and code managers. The customization server uses specific kernel context to create specialized code from generic code. We applied our approach to Linux TCP/IP stack, reducing the code size by a factor of 20, improving the execution speed by up to 25%, and increasing the throughput by up to 21%. The portability and maintenance advantages of our approach are demonstrated by our experiments on three architectures: Pentium III, Intel 486, and ARM.

Among our future projects, we intend to apply our customization infrastructure to other subsystems, like memory management. Also, we plan to explore other general-purpose operating systems like Windows.

8. ACKNOWLEDGMENTS

We are indebted to Anne-Françoise LeMeur, now at the University of Lille, for her help throughout the project and to Abhishek Kumar at Georgia Tech for helping us through some obstacles in the TCP/IP case study. We are also greatful to Julia Lawall at DIKU and Claus Brabrand at BRICS, who significantly improved the writing and presentation of this paper.

9. REFERENCES

[1] 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.

[2] S. Bhatia, C. Consel, A.-F. Le Meur, and C. Pu. Tool-based specialization of protocol stacks in OS kernels. Research report, LaBRI, February 2004.

[3] Hewlett-Packard company Information Networks Divis ion. *Netperf: A network performance benchmark*, February 1996.

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

[5] Evans Data Corporation. Embedded systems development survey. http://www.evansdata.com, 2003.

[6] D. Duggan. Type-based hot swapping of running modules. In *International Conference on Functional Programming*, pages 62–73, 2001.

[7] 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.

[8] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St-Malo, France, October 1997.

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

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

[11] J. L. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *Proceedings of the ACM SIGOPS European Workshop 2002 (EW2002)*, pages 54–62, Saint-Emilion, France, September 2002.

[12] A.-F. Le Meur, J.L. Lawall, and C. Consel. Specialization scenarios: A pragmatic approach to

declaring program specialization. *Higher-Order and Symbolic Computation*, 17(1):47–92, 2004.

[13] 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*, 19(2):217–251, May 2001.

[14] 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.

[15] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996.

[16] A. C. Veitch and N. C. Hutchinson. Kea - a dynamically extensible and configurable operating system kernel. In *Proceedings of the 1996 Third International Conference on Configurable Distributed Systems (ICCDS)*, 1996.