Safe Operating System Specialization: the RPC Case Study*

Eugen-Nicolae Volanschi & Gilles Muller & Charles Consel

IRISA/INRIA,
Campus de Beaulieu,
F-35042 Rennes Cedex, France
E-mail: {volanski,muller,consel}@irisa.fr

Abstract

Adaptive operating systems allow one to optimize system functionalities with respect to common situations. We present an experiment aimed at optimizing the RPC implementation in CHORUS by manual specialization. We show that there exist numerous opportunities for specialization and that they can lead to great improvements. Then, we discuss how this optimization can be reproduced automatically with a specializer for C programs.

1 Introduction

Services offered by operating systems are, by nature, general. As new applications and hardware platforms emerge, this increasing generality penalizes performance. This conflict between generality and performance is at the basis of several research projects which are aimed at designing operating systems which can adapt to usage patterns to treat common cases efficiently [6, 9, 1, 7]. This adaptation is mainly based on customizing/specializing operating system calls with respect to some contextual information.

To achieve this adaptation a few important issues must be addressed: (1) how can code be adapted? (2) how can the adapted code be trusted by the kernel? (3) how should the adapted code be incorporated in the kernel?

This paper explores these issues in the context of Remote Procedure Calls (RPC) in CHORUS/ClassiX. We describe how aspects of protocol layers can be adapted to exploit a usage pattern. This adaption is achieved by applying systematically specialization techniques. Performance measurements of the specialized fragments demonstrate that this approach produces major improvements (on these fragments).

After briefly presenting a specializer for C programs, named Tempo [4], we also discuss how this optimization on RPC could be done automatically using Tempo.

*This research is supported in part by France Telecom/SEPT, ARPA grant N00014-94-1-0845, and NSF grant CCR-92243375.

2 Specializing Operating Systems

Trusting optimized code. An important requirement when importing code in an operating system is to ensure that it can be trusted. Let us examine how this requirement is addressed by the three main approaches to adaptive operating systems. In Synthesis [11], adaptation relies on templates written by the programmer. As a consequence, the specialized code may be unsafe. In the SPIN operating system [1], extensions are written in a strongly-typed language; this makes it possible to ensure some degree of safety. Other safety aspects are handled at run time using predicates on the code. In the context of RPC, this run-time approach is used by Thekkath *et al.* [12] to ensure the safety of the optimized code. This is achieved by a mechanism similar to packet filtering.

In contrast, we propose to optimize system components by specializing the existing code. Like any optimizer, if it is semantic preserving the specializer will produce optimized code that can be trusted (if the non-specialized one is). This approach has already been applied to the HP-UX file system as reported by Pu *et al.* [10]. In this work, the notion of optimistic specialization is introduced. That is, at run time, when invariants become valid, they are used to specialize specific code fragments. If the invariants are invalidated, the specialized fragments are removed and the general versions are reinstalled. In a more advanced strategy, when invalidation occurs, fragments specialized with respect to other invariants should be reinstalled.

Importing optimized code. Specializing parts of an operating system requires the specialized parts to be incorporated (statically or dynamically) into the operating system. Since, the address space of the kernel is limited, it is important to keep the specialized code within the application as a library of optimized routines. This code should run with supervisor privileges to allow it to exploit data from the kernel. For safety reasons, access from the application to this code should be restricted, even if it lies in the same address space. To do so, an approach proposed by Muller and Bryce consists of defining several *domains of protection* (DP) [2] within the application. This insures both protection and scalability. The specialized code is placed in a DP supervisor, and only previously declared entry points allow the application program to exploit this code. In the implementation, a DP call is defined as a system trap which takes an entry point specification within this restricted set. All other function calls in the DP are accessible only within the DP itself.

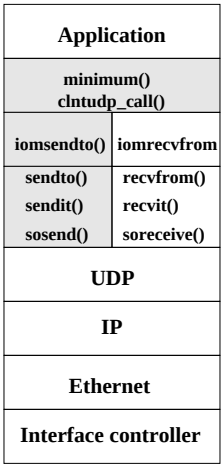

Figure 1: The protocol stack underlying the RPC implementation. The grayed part indicates the code fragment which has been specialized.

3 The RPC Case Study

Our study is concerned with the specialization of part of the RPC in Chorus/ClassiX. The stack of protocols to be optimized in presented in Figure 1. For the interface compiler, `rpcgen` is used — Sun's RPC stub generator. The underlying communication primitives used by `rpcgen` are those of a BSD socket. As an example of an RPC service, we define a procedure which calculates the minimum of two numbers.

3.1 Manual Specialization

In the RPC case, the system routines for message passing are used to implement a particular client/server interface. This situation offers several opportunities for specialization.

Main invariants. First, in the automatically generated stubs, part of the communication configuration is fixed (*e.g.,* whether the protocol is connection oriented). In our study, because the underlying protocol is UDP, a socket is non-blocking and datagram, and the send operation is atomic. Also, the stub contains calls to the send routine (`sendto()`) where some flags are constant.

Second, a given client/server interface may fix the size of messages for a service call. This situation occurs when the service requires a fixed number of arguments of a fixed size. Knowing the size of the messages, their packing routines can be unfolded; also, the size of the kernel buffer can be computed statically and its allocation can be optimized.

Another opportunity corresponds to the location of the server which may be fixed during some period of time. As long as the server does not migrate, its location does not need to be re-computed, nor does it need to be copied in kernel space at every socket operation.

Last, for modularity reasons, each layer of the protocol stack is written independently of the adjacent layers. For a fixed stack of layers, some functions can be unfolded and the copy of some parameters can be avoided.

Exploiting invariants. We have specialized the fragment of the protocol stack between the application down to the socket level. Only the sending side (Fig. 1) has been addressed.

The `sendit()` routine is unfolded in `sendto()` intermediate data structure is eliminated. The kernel buffer for the receiver's address is allocated and initialized when the client is created rather for each send. The invariants mentioned above make it possible to eliminate many tests in the socket layer. The size of the packet being known (48 bytes), the XDR packing routines are rewritten without any function call. The pre-initialization of the XDR header is extended to the RPC header (service number, credentials). In the optimized version only the arguments have to be packed for each call.

	Original	Specialized	Speedup
Socket level (emission only)	43.75	11.90	72.8%
Stub to socket (emission only)	109	74	32.1%

Table 1: Performance comparison. Times are given in μs. Both versions execute the specialized code in a DP supervisor within the application itself.

Performance We measured the speedup of the manually specialized version on two Pentium 90 PC's running CHORUS/ClassiX, connected by a 10M-bit/s Ethernet network. The timings are given in Table 1.

As can be noticed the speedup at the socket level for the send operation is over 70% (the time was measured by replacing the call to inferior layers by an immediate return). The importance of this gain decreases to about 30% when considering all the specialized layers (stub to socket). This is because the time spent in the stub and in the DP-IOM levels has not changed, except for $3\mu s$ which comes from the optimized packing function for the arguments.

4 Automatic Specialization

The speedup obtained by manual specialization clearly demonstrates that this optimization technique is promising.

We are currently developing a specialization system for C programs, named Tempo [4]. Advanced features are being incorporated to process system code. This new system will allow us to reproduce automatically the optimization of the RPC.

4.1 The Tempo Platform

Tempo is a partial evaluator. It takes a source program written in C and parts of its input, and produces a specialized version. Tempo is an off-line partial evaluator [3, 8] in that it processes a program in two steps: during the first phase only a known/unknown division of the input is given. The program is analyzed and a transformation is associated with each program construct. The result of the first phase can either be used for compile-time or run-time specialization. In the former case, a concrete value is given for each known input at compile time, and the program is specialized at compile time as well. In the latter case, the concrete values only become known at run time, and specialization

relies on a strategy based on templates [5]. In both cases, the specialization process is guided by the transformations generated by the first phase.

The input to the first phase is given in a *binding-time context* file. The values for the second phase are given in a *specialization context* file. A third file defines the initial alias context.

The analysis phase handles *partially-static structures* (in which only some but not all the fields are known). It also treats pointers to partially-static data; they are handled in a dual way: they are both dereferenced during specialization and residualized in the specialized program. We will refer to them as *explicated pointers*.

4.2 A Systematic Approach

We are describing here a systematic way to apply our specialization tool to system code. This description focuses on the comparison between what can be done manually and the corresponding automatic transformations. In this context, an important issue is concerned with classifying the invariants. Another issue is to identify additional mechanisms needed when specializing only a part of a large system, both at specialization time and at run time.

Categories of Invariants

In this paper an *invariant* is defined as a variable having a constant value during some period of time.

Invariants are said to be *compile time* if they are known prior to run time and they are valid throughout the execution of the program. Quasi-invariants are compile time if they are known prior to run time but can change during execution.

Invariants are said to be *run time* if they are not known prior to run time and cannot be invalidated. Quasi-invariants are run-time if they are not known until run time but can be invalidated during the execution.

Interestingly, we can sometime use compile-time specialization for runtime invariants, when the set of possible values of a variable is known prior to execution, and this set is small enough (*e.g.*, boolean variables). Then, we can perform a compile-time specialization for each possible value, and rely on a run-time mechanism to select between several versions. The reason why compile-time specialization is used whenever possible is efficiency: no overhead for specializing the code is required during run time. Even though Tempo offers an efficient run-time specializer.

Let us now describe some classes of invariants which frequently appear in operating system code. We start with those which can be used for compile-time specialization:

Compile-Time Specialization

Tight coupling of software components. Modules allow some functionalities of a system to be added or replaced. However, when the coupling of modules is fixed, they can be optimized by specialization. In our RPC study, the IP layer can support both UDP and TCP, but only UDP is used. This situation allows us, for example, to eliminate retransmission buffers.

Options as invariants. In our study, options can be used for compile-time specialization, even if their value is not known before runtime. As described previously, it suffices to select commonly occuring values to perform specialization. This is simple in the case of some options where possible values are well-defined. As an example, in the context of sockets, the flag SO_DONTROUTE varies during a send, but can be treated as a quasi-invariant.

Constants in the text. They can occur in several situations. First, there are elements of a static configuration (*e.g.,* the size of a buffer or of an Ethernet packet). Then, some library functions are written in a generic way but most of the time they are used in a specific way. For instance, the function may take a vector of strings, but is often invoked with a vector of size one. This is the case for `uiomove()` which copies memory, and `sendit()` which can send a whole `iovec`. Third, constants can come from functionalities which are not implemented. For example, structure fields reserved for future use (the access rights of a message, in RPC).

Here we see that there are many opportunities for compile-time specialization, besides the obvious constants introduced by `#define`.

Let us now examine cases of run-time specialization.

Run-Time Specialization

Session-oriented invariants. Invariants are often created when a session is opened (*e.g.,* opening a file or a connection). Then, a system data structure is allocated to record various pieces of information. Parts of the information are repeatedly interpreted and yet do not change for the duration of the session or for a long period. For instance, during a TCP session, the server address cannot change. In the RPC case, the identity of the server may change, although this situation does not occur frequently.

Manual specialization exploits the server address invariant by "lifting" the code which allocates and initializes the address from the sending routine to the client creation routine. This code motion corresponds to run-time specialization in that, during object creation the sending function is specialized with respect to the server's address. This specialization involves executing the "static" parts of send, including the address allocation and initialization. So, code motion is done in a transparent way by the specializer.

Isolating The Code To Be Specialized

Now that the possible invariants have been identified, we examine what pieces of code should be specialized with respect to these invariants.

Tempo allows one to specialize an isolated part of a program. Any collection of functions may be selected, provided the right "context" for specialization is specified. For large programs, this may be non-trivial. In the RPC case, the context of the send routines is constructed by the socket creation routines (`socreate()`, `bind()`) called when the client is created. This context includes a great number of nested structures, which are partially static and contain pointers to each other. Supplying the context involves finding out which fields are static, determining their values, and specifying the initial aliases.

For the specialization values, we are investigating a systematic approach where program parts not involved in the specialization can be executed to construct the specialization context and then collect the static parts of the state. This state-extracting function might be written by the user of Tempo, or (semi-)automatically generated based on some declarations.

Also, a few functions need to be supplied to simulate, at specialization time, the run-time behavior of some external functions which otherwise could not be called statically. An example is `getsock(fd)`, which takes a file number — not known until runtime — and returns a partially-static file descriptor.

Predicting speedup Once we have chosen the invariants and the piece of code to specialize, we might be interested in having an estimate of the speedup, based on the cost of static computations which will be evaluated away. Having a tool for this could avoid generating many different specialized kernels and then running benchmarks on each of them. Even a rough estimate of the speedup would be helpful.

Runtime support We already saw that the code can be specialized at compile-time with respect to several interesting values. But also, it might be specialized with respect to different binding-time contexts. For example, we can have a server interface which offers a service that takes a list of arguments; here, the size of the messages will not be known, but we still want to have an optimized `sendto()` which exploits all the other invariants. Therefore, the correct code to be invoked depends either on values or on binding-times. In our example however, almost all the invariants are true invariants. The main exception is the address of the remote server, which can sometimes change. In this case, we must modify its number in the packet headers. Additionally, if the new server runs on a different architecture, we must modify the packing routines for scalar types, used during marshaling.

For the runtime support, we will use the technique of guards and re-plugging presented by Pu *et al.* [10].

Comparison When comparing manual and automatic specialization, there are a number of differences to consider.

For instance, when `sendit()` passes to `sosend()` the address of `struct uio auio`, manual specialization will pass only the dynamic part of this partially-static structure, namely the content of the message — the embedded `struct iovec`. The Tempo version will still pass `&auio` as an explicated pointer. The same difference occurs when a partially static structure is copied in an unstructured way, with `bcopy()` for example. In the first case, not much is lost (the cost of an indirection), but in the second case Tempo needs to be extended. We are currently investigating some automatic solutions for treating simple situations of unstructured copying, since this occurs frequently in the code.

Other differences come from the fact that Tempo is an off-line partial evaluator, even for compile-time specialization, where all the static *values* are known. For instance, manual specialization will consider the expression `(flags & MSG_DONTROUTE) && (so->so_options & SO_DONTROUTE) == 0` as False if the corresponding flag is known to be 0, but Tempo will consider it dynamic, because of the unknown `so_options`. This loss of accuracy will eventually be circumvented by a CPS analysis of conditionals.

5 Conclusions

This paper explores opportunities to apply specialization techniques to system code. In particular, we study the optimization of protocol layers. We show that great speedup can be obtained using this approach. Then, we discuss how this process can be automated using a specializer for C programs.

Acknowledgments

We want to thank Frédéric Mazé for his important contribution to the manual specialization, and Luke Hornof for his thorough comments on drafts of this paper.

References

[1] B. 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, University of Washington, Seattle, Washington, February 1994.

[2] C. Bryce and G. Muller. Matching micro-kernels to modern applications using fine-grained memory protection. In *Procedings of the seventh IEEE Symposium on Parallel and Distributed Processing*, pages 272–279, San Antonio (Tx), Oct. 1995.

[3] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*, pages 493–501. ACM, ACM, 1993.

[4] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.-N. Volanschi. A uniform approach for compile-time and run-time specialization. Publication interne 979, Irisa, Rennes, France, Dec. 1995. To appear in the Proceedings of the Seminar on Partial Evaluation, Dagstuhl, 1996.

[5] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the 23^{rd} Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, St. Petersburg Beach, Florida, USA, Jan. 1996. ACM Press.

[6] C. Consel, C. Pu, and J. Walpole. Incremental partial evaluation: The key to high performance, modularity, and portability in operating systems. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, Copenhagen, 1993.

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[8] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[9] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report TR-94-20, University of Arizona, Tucson, Arizona, 1994.

[10] 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 15th ACM Symposium on Operating Systems Principles*, 1995.

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

[12] Thekkath, Lazowska, Nguyen, and Moy. Implementing network protocols at user lvel. Technical Report TR 93-03-01, University of Washington, 03 1993.