INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fast, Optimized Sun RPC Using Automatic Program Specialization

Gilles Muller , Renaud Marlet , Eugen-Nicolae Volanschi , Charles Consel , Calton Pu , Ashvin Goel

N° 3220

juillet 1997

THÈME 2

Rapport de recherche

ISSN 0249-6399

Fast, Optimized Sun RPC Using Automatic Program Specialization

Gilles Muller , Renaud Marlet , Eugen-Nicolae Volanschi , Charles Consel , Calton Pu , Ashvin Goel

Thème 2 — Génie logiciel
et calcul symbolique

Projet LANDE / avant-projet COMPOSE

Rapport de recherche n ° 3220 — juillet 1997 — 19 pages

Abstract: Fast remote procedure call (RPC) is a major concern for distributed systems. Many studies aimed at efficient RPC consist in either new implementations of the RPC paradigm or manual optimization of critical sections of the code. This paper presents an experiment that achieves automatic optimization of an existing, commercial RPC implementation, namely the Sun RPC. The optimized Sun RPC is obtained by using an automatic program specializer. It runs up to 1.5 times faster than the original Sun RPC. Close examination of the specialized code does not reveal further optimizations opportunities which would lead to significant improvements without major manual restructuring.

The contributions of this work are: (1) the optimized code is safely produced by an automatic tool and thus does not entail any additional maintenance; (2) to the best of our knowledge this is the first successful specialization of mature, commercial, representative system code; and (3) the optimized Sun RPC runs significantly faster than the original code.

Key-words: partial evaluation, RPC protocol, Sun RPC, distributed systems, generic systems, automatic optimisatization.

(Résumé : tsvp)

This report was actually written in March 1997, and later published as an INRIA research report in July 1997. A revised version with further results is currently (i.e., July 1997) at work.

The Sun RPC experiment files, including the specialized implementation, are publicly available upon request to the authors at email address `muller@irisa.fr`.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 02 99 84 71 00 – Télécopie : (33) 02 99 84 71 71

Optimisation du RPC Sun à l'aide de la spécialisation automatique de programmes

Résumé : La rapidité des appels de procédure à distance (RPC) est un souci majeur dans les systèmes distribués. La plupart des études qui visent à produire des RPC efficaces consistent soit en de nouvelles implémentations du paradigme RPC, soit en des optimisations manuelles des sections critiques du code. Ce rapport présente une expérience d'optimisation automatique d'une implémentation RPC déjà existante et commerciale, le RPC de Sun. La version optimisée du RPC de Sun est produite par un spécialiseur automatique de programmes. Son exécution est jusqu'à 1,5 fois fois plus rapide que le RPC de Sun d'origine. En examinant le code spécialisé, on ne trouve pas d'opportunités de spécialisation additionnelle qui apporterait une amélioration significative sans restructuration majeure du code.

Les contributions contenues dans ces travaux sont : (1) le code optimisé est produit de manière sûre par un outil automatique, ce qui n'entraîne aucun coût de maintenance additionnel ; (2) à notre connaissance, il s'agit là de la première expérience réussie de spécialisation d'un code système commercial, mûr et représentatif ; (3) le RPC de Sun optimisé est significativement plus rapide que le code d'origine.

Mots-clé : évaluation partielle, protocole RPC, RPC de Sun, systèmes distribués, systèmes génériques, optimisation automatique.

1 Introduction

Specialization is a well known technique for improving the performance of operating systems [3, 11, 23, 27]. However, only recently have programming tools begun to be used to help system programmers perform specialization. To the best of our knowledge, this paper reports the first successful specialization of a significant OS component (the Sun RPC) using a partial evaluator. This work is significant for a combination of three main reasons: (1) automatic optimization of existing system code using a partial evaluator preserves the original source code (2) specialization applies to mature, commercial, representative system code, and (3) the specialized Sun RPC exhibits significant performance gains. We elaborate each reason in turn.

First, partial-evaluation based specialization is qualitatively different from manual specialization done in the past [27, 23, 3, 11]. Manual specialization requires the system programmer to identify every occurrence of the invariants to be exploited and to write the specialized code exploiting these invariants. Although this approach may lead to significant performance gains, the manual specialization process is error-prone and results in code that is expensive to maintain. In contrast, a partial evaluator preserves the source code, and generates automatically the specialized code guided by the declarations of invariants specified by system programmers. Since we are specializing mature commercial code, the preservation of original code and semantics also preserves safety and maintainability. In our view, tools such as partial evaluators may help the industry to address the operating system code complexity concerns pointed out by an industry panel at OSDI'96.

Second, we specialize mature, commercial code (Sun RPC) that we believe to be representative of production quality code. Sun RPC is one layer in the communication stack, and RPC itself is divided into micro-layers, each concerned with a reasonably small task, e.g., managing the underlying transport protocol such as TCP or UDP. The RPC code has been ported to a variety of software and hardware foundations, while preserving its layered structure.

Third, we obtain significant performance gains using partial-evaluation based specialization. In our experiment, the optimized Sun RPC runs up to 1.5 times faster than the original Sun RPC. In addition, the specialized marshaling process runs up to 3.75 times faster than the original one. Close examination of the specialized code does not reveal further optimizations opportunities which would lead to significant improvements without major manual restructuring.

Our partial-evaluation based specialization experiment shows the promise of direct industrial relevance to commercial systems code.

The rest of the paper is organized as follows. Section 2 presents Sun RPC protocol and the optimization issues. Section 3 examines opportunities for Specialization in the Sun RPC. Section 4 gives an overview of the partial evaluator Tempo and shows its relevance for Sun RPC specialization. Section 5 describes the performance experiments. Section 6 discusses our experience with partial-evaluation based specialization of Sun RPC. Section 7 summarizes related work and Section 8 concludes the paper.

2 The Sun RPC and the Optimization Issues

The Sun RPC (Remote Procedure Call) protocol was introduced in 1984 to support the implementation of distributed services. This protocol has become a *de facto* standard in distributed service design and implementation, e.g., NFS [22] and NIS [29]. Since large networks are often heterogeneous, support for communicating machine independent data involves encoding and decoding. Such environments (e.g., PVM [14] for a message passing model and Stardust [5] for a Distributed Shared Memory model) often rely on Sun XDR. The two main functionalities of the Sun RPC are:

1. A stub generator (`rpcgen`) that produces the client and server stub functions. The stub functions translate procedure call parameters into a machine independent message format called XDR, and XDR messages back into procedure parameters. The translation of parameters into messages in known as *marshaling*.
2. The management of message exchange through the network.

Concretely, the Sun RPC code consists of a set of micro-layers, each one devoted to a small task. For example, there are micro-layers to write data during marshaling, to read data during unmarshaling and to manage specific transport protocols such as TCP or UDP. Each micro-layer has a generic function, but it may have several implementations. As such, the micro-layer organization of RPC code is fairly representative of modular production system software.

A Simple Example

We consider a simple example to illustrates the micro-layer organization of Sun RPC code. The function `rmin` sends two integers to a remote server, which returns their minimum.

The client uses `rpcgen` (the RPC stub compiler) to compile a procedure interface specification for `rmin` into an assortment of source files. These files implement both the call on the client's side and the dispatch of procedures on the server's side. To emphasize the actual code executed, instead of including all the files generated by `rpcgen`, Figure 1 shows an abstract execution trace of a call to `rmin`.[1]

Performance of RPC

Communication using the RPC paradigm is at the root of many distributed systems. As such, the performance of this component is critical. As a result, a lot of research has been carried out on the optimization of this paradigm [32, 6, 17, 36, 15, 26]. Many studies have been carried out, but they often result in using new protocols that are incompatible with an existing standard such as the Sun RPC. The problem in reimplementing a protocol that is specified only by its implementation is that features (and even bugs) may be lost, resulting in incompatible implementation.

Optimizing Existing Code

An alternative to reimplementing a system component for performance reasons is to directly derive an optimized version from the existing code. An advantage of starting with existing code is that the derived version remains compatible with existing standards. Another advantage is that the systematic derivation process can be repeated for different machines and systems.

The question that naturally arises at this point is: are there important opportunities for deriving significantly optimized versions of existing system components?

[1] For clarity, we omit some clutter in code listings: declarations, “uninteresting” arguments and statements, error handling, casts, and a level of function call.

```
arg.int1 = ...                      // Set first argument
arg.int2 = ...                      // Set second argument
rmin(&arg)                          // RPC User interface generated by rpcgen
  clnt_call(argsp)                  // Generic procedure call (macro)
    clntupd_call(argsp)             // UDP generic procedure call
                                    // Write procedure identifier
      XDR_PUTLONG(&proc)            //   Generic marshaling to memory, stream... (macro)
        xdrmem_putlong(lp)          //   Write in output buffer and check overflow
          htonl(*lp)                //   Choice between big and little endian (macro)
      xdr_pair(argsp)               // Stub function generated by rpcgen
                                    //   Write first argument
        xdr_int(&argsp->int1)       //     Machine dependent switch on integer size
          xdr_long(intp)            //     Generic encoding or decoding
            XDR_PUTLONG(lp)         //     Generic marshaling to memory, stream... (macro)
              xdrmem_putlong(lp)    //     Write in output buffer and check overflow
                htonl(*lp)          //     Choice between big and little endian (macro)
                                    //   Write second argument
        xdr_int(&argsp->int2)       //     Machine dependent switch on integer size
          xdr_long(intp)            //     Generic encoding or decoding
            XDR_PUTLONG(lp)         //     Generic marshaling to memory, stream... (macro)
              xdrmem_putlong(lp)    //     Write in output buffer and check overflow
                htonl(*lp)          //     Choice between big and little endian (macro)
```

Figure 1: Abstract trace of the encoding part of a remote call to `rmin`

In fact, existing system components are known to be generic and structured in layers and modules. This results in various forms of interpretation which are important sources of overhead as shown, for example, in the HP-UX file systems [27]. In the Sun RPC, this genericity takes the form of several layers of functions which interpret descriptors (i.e., data structures) to determine the parameters of the communication process: choice of protocol (TCP or UDP), whether to encode or decode, buffer management, . . .

Importantly, most of these parameters are known for any given remote procedure call. This information can be exploited to generate specialized code where these interpretations are eliminated. The resulting code is tailored for specific situations.

Let us now examine forms of these interpretations in the Sun RPC code and how they can be optimized via specialization.

3 Opportunities for Specialization in the Sun RPC

The Sun RPC relies on various data structures such as `CLIENT` or `XDR`. Some fields of those data structures have values that can be available before execution actually takes place; they do not depend on the run-time arguments of the RPC. The values of those fields are either repeatedly interpreted or propagated throughout the layers of the encoding/decoding process. Because these values can be available before execution, they may be the source of optimizations: the computations depending only on known (or *static*) values can be performed during a specialization phase. The specialized program only consists of the computations depending on the unknown (or *dynamic*) values.

We now describe typical specialization opportunities in the Sun RPC. We illustrate these opportunities with actual code excerpts annotated to show static and dynamic computations. In the following figures, dynamic computations correspond to code fragments printed in bold face; static computations are printed in Roman.

3.1 Eliminating Encoding/Decoding Dispatch

We first examine an opportunity for specialization that illustrates a form of interpretation. The function exhibiting that is `xdr_long`, given in Figure 2. This function is capable of both encoding and decoding long integers. It selects the appropriate operation to perform based on the field `x_op` of its argument `xdrs`. This form of interpretation is used in similar functions for other data types.

In fact, the field `x_op` is known from the execution context (i.e., encoding or decoding process). This information, contained in the `xdrs` structure, can be propagated interprocedurally down to the function `xdr_long`. As a result, the dispatch on `xdrs->x_op` is totally eliminated; the specialized version of this function is reduced to only one of the return constructs. In this case, the specialized `xdr_long()`, being small enough, disappears after inlining.

```
bool_t xdr_long(xdrs,lp)                 // Encode or decode a long integer
  XDR *xdrs;                             //     XDR operation handle
  long *lp;                              //     pointer to data to be read or written
{
  if( xdrs->x_op == XDR_ENCODE )         // If in encoding mode
        return XDR_PUTLONG(xdrs,lp);     //     Write a long int into buffer
  if( xdrs->x_op == XDR_DECODE )         // If in decoding mode
        return XDR_GETLONG(xdrs,lp);     //     Read a long int from buffer
  if( xdrs->x_op == XDR_FREE )           // If in "free memory" mode
        return TRUE;                     //     Nothing to be done for long int
  return FALSE;                          // Return failure if nothing matched
}
```

Figure 2: Reading or writing of a long integer: `xdr_long()`

3.2 Eliminating Buffer Overflow Checking

Another form of interpretation appears when buffers are checked for overflow. This situation applies to function `xdrmem_putlong` displayed in Figure 3. More specifically, as parameter marshaling proceeds, the remaining space in the buffer is maintained in the field `x_handy`. Similar to the first example, `xdrs->x_handy` is first initialized (i.e., given a static value), and then decremented by static values and tested several times (for each call to `xdrmem_putlong` and related functions). Since the entire process involves static values, the whole buffer overflow checking can be performed during a specialization phase, before actually running the program. Only the buffer copy remains in the specialized version (unless a buffer overflow is discovered).

This second example is important not only because of the immediate performance gain, but also because in contrast with a manual, unwarranted deletion of the buffer overflow checking, the elimination described here is strictly and systematically derived from the original program.

```
bool_t xdrmem_putlong(xdrs,lp)               // Copy long int into output buffer
  XDR *xdrs;                                 //     XDR operation handle
  long *lp;                                  //     pointer to data to be written
{
  if((xdrs->x_handy -= sizeof(long)) < 0)    // Decrement space left in buffer
    return FALSE;                            //     Return failure on overflow
  *(xdrs->x_private) = htonl(*lp);           // Copy to buffer
  xdrs->x_private += sizeof(long);           // Point to next copy location in buffer
  return TRUE;                               // Return success
}
```

Figure 3: Writing a long integer: `xdrmem_putlong()`

3.3 Propagating Exit Status

The third example uses the results from the previous examples.

The return value of the procedure `xdr_pair` (shown in Figure 4) depends on the return value of `xdr_int`, which in turn depends on the return value of `xdr_putlong`. We have seen that `xdr_int` and `xdr_putlong` have a static return value. Thus the return value of `xdr_pair` is static as well. If we specialize the caller of `xdr_pair` (i.e., `clntudp_call`) as well to this return value, `xdr_pair` needs no longer return a value: the type of the function can be turned in `void`. The specialized procedure, with the specialized calls to `xdr_int` and `xdr_putlong` inlined, is shown in Figure 5. The actual result value, which is always `TRUE` independently of dynamic `objp` argument (writing the two integers never overflows the buffer), is used to reduce an extra test in `clntudp_call` (not shown).

```
bool_t xdr_pair(xdrs, objp)              // Encode arguments of rmin
{
  if (!xdr_int(xdrs, &objp->int1)) {     // Encode first argument
    return (FALSE);                      //     Possibly propagate failure
  }
  if (!xdr_int(xdrs, &objp->int2)) {     // Encode second argument
    return (FALSE);                      //     Possibly propagate failure
  }
  return (TRUE);                         // Return success status
}
```

Figure 4: Encoding routine `xdr_pair()` used in `rmin()`

```
void xdr_pair(xdrs,objp)              // Encode arguments of rmin
{
                                      // Overflow checking eliminated
  *(xdrs->x_private) = objp->int1;    // Inlined specialized call
  xdrs->x_private += 4u;              //   for writing the first argument
  *(xdrs->x_private) = objp->int2;    // Inlined specialized call
  xdrs->x_private += 4u;              //   for writing the second argument
                                      // Return code eliminated
}
```

Figure 5: Specialized encoding routine `xdr_pair()`

3.4 Assessment

The purpose of the encoding is to copy the arguments into the output buffer. To perform this task, the minimal code that we can expect (using the approach of a separate output buffer) is basically what is shown in Figure 5. The same situation applies for decoding, except that additional dynamic tests must be performed to ensure the soundness and authenticity of the server reply.

We have seen that a systematic approach to specializing system code can achieve significant code simplifications. We now discuss how this process is automated using a partial evaluator for C programs, named Tempo.

4 Automatic Specialization Using the Tempo Partial Evaluator

It is not feasible to manually specialize RPC for each remote function, as the process is long, tedious, and error-prone. However, since the process is systematic, it can be automated. Ultimately, it should be as automatic as `rpcgen`.

Tempo is a program transformation system based on *partial evaluation* [7, 18]. Tempo takes a source program $P_{generic}$ written in C together with a known subset of its inputs, and produces a specialized C source program $P_{special}$, simplified with respect to the known inputs. Although Tempo supports both compile-time and run-time program specialization, in the RPC experiments reported in this paper we only use compile-time specialization.

Tempo uses a description of the inputs (which inputs are static and which inputs are dynamic) to analyze $P_{generic}$, dividing it into *static* and *dynamic* parts. Then the static part of $P_{generic}$ is evaluated using concrete values for each known input, while the dynamic part is *residualized* (copied) into the output specialized program. The result $P_{special}$ is typically simpler than $P_{generic}$ since the static part has been pre-computed and only the dynamic part will be executed at runtime.

The binary division of program components (a.k.a. *binding time*) into static and dynamic parts turned out to be insufficient for C programs in operating systems. The main refinements introduced in Tempo to transform system programming code include:

- *partially-static structures*: Figure 3 shows that some fields of the `xdrs` structure are static while others are dynamic. Effective specialization requires that we be able to access the static fields at specialization time. Without such a functionality the whole structure must conservatively be considered dynamic and the repeated buffer overflow checking cannot be eliminated.

- *flow sensitivity:* Possible errors occurring in the decoding of input buffer introduces dynamic conditions after which static information is lost; however each branch of corresponding conditionals still can exploit static information. For that, binding time of variables (i.e., static or dynamic) must not be a global property; it must depend on the program point considered.

- *context sensitivity:* The integer encoding function is usually called with dynamic data, representing the RPC arguments. However, there is one encoding of a static integer in each remote procedure call: the marshaling of the procedure identifier. It is useful to differentiate between the two call contexts, in order not to lose this opportunity for specialization. Effective specialization requires that calls to the same functions with different binding time context refer to different binding-time instances of this function definition.

- *static returns:* As seen with example in section 3.3, the computation at specialization time of exit status tests relies on the ability to statically know the return value of a function call even though its arguments and its actions on input/output buffers are dynamic. More generally, the return value of a function may be static even though its arguments and side-effects are dynamic. Thus we can use the return value of a function call even when the call must be residualized.

Tempo also relies on several other programming language analyses, such as pointer alias and dependency analysis. It goes beyond conventional constant propagation and folding, in the sense that it is not limited to exploiting scalar values *intra-procedurally*. Tempo not only deals with aliases and partially-static data structures, it also propagate them *inter-procedurally*. These features are critical when tackling system code. In fact, Tempo has been targeted towards system software; experiments such as the Sun RPC specialization have driven its design and implementation. Concretely, Tempo is able to achieve the specialization described in Section 3.

5 Performance Experiments

Having explained the forms of specialization that Tempo performs on the RPC code, we now turn to the assessment of the resulting optimized RPC.

The test program. We have specialized both the client and the server code of the 1984 copyrighted version of Sun RPC. The unspecialized RPC code is about 1500 lines long (without comments) on the client side and 1700 on the server side. The test program, which utilizes remote procedure calls, emulates the behavior of parallel programs that exchange large chunks of structured data. This is a benchmark representative of applications that use a network of workstations as large scale multiprocessors.

Platforms for measurements. Measurements have been done on two kinds of platforms:

- Two Sun IPX 4/50 workstations running SunOS 4.1.4 with 32 MB of memory connected with a 100 Mbits/s ATM link. ATM cards are model ESA-200 from Fore Systems. This platform is 3 years old and quite inefficient compared to up to date products, both in term of CPU, network latency and bandwidth (i.e., 155 Mbits / 622 Mbits).
- Two recent 166 MHz Pentium PC machines running Linux with 96 MB of memory and a 100 Mbits/s Fast-Ethernet network connection. There were no other machines on this network during experiments.

Our specialization is tested on different environments in order to check that the results we obtain are not specific to a particular platform. All programs have been compiled using `gcc` version 2.7.2, with the option `-O2`.

Benchmarks. To evaluate the efficiency of specialization, we have made two kinds of measurements: (i) a micro-benchmark of the client marshaling process, and (ii) an application level benchmark which measures the elapsed total time of a complete RPC call (round-trip). The client test program loops on a simple RPC which sends and receives an array of integers. The intent of this second experiment is to take into account architectural features such as cache, memory and network bandwidth that affect global performance significantly. Performance comparisons for the two platforms and the two experiments are shown Figure 6. The marshaling and round-trip benchmark numbers result from the mean of 10000 iterations.

Not surprisingly, the PC/Linux platform is always faster than the IPX/SunOs's one. This is partly due to a faster CPU, but also to the fact that the Fast-Ethernet cards have a higher bandwidth and a smaller latency than our ATM cards. A consequence of instruction elimination by the specialization process is that the gap between platforms is lowered on the specialized code (see marshaling comparisons in Figure 6-1 and 6-2).

Marshaling. Detailed micro-benchmark results are shown in Table 1. The specialized client stub code runs up to 3.7 faster than the non-specialized one on the IPX/SunOS, and 3.3 on the PC/Linux. Intuitively, one would expect the speedup to increase with array size, since more instructions are being executed. However, on the Sun IPX the speedup decreases with the size of the array of integers (see Figure 6-5). The explanation is that program execution time is dominated by memory accesses. When the array size grows, most of the marshaling time is spent in copying the integer array argument into the output buffer. Even though specialization decreases the number of instructions used to encode an integer, the number of memory moves remains constant between the specialized and non-specialized code. Therefore, the instruction savings becomes comparatively smaller as the array size grows. For the PC, this behavior does not appear; the speedup curve only bends.

Array Size	IPX/SunOs			PC/Linux		
	Original	Specialized	Speedup	Original	Specialized	Speedup
20	0.047	0.017	**2.75**	0.071	0.063	**1.20**
100	0.20	0.057	**3.50**	0.11	0.069	**1.60**
250	0.49	0.13	**3.75**	0.17	0.08	**2.10**
500	0.99	0.30	**3.30**	0.29	0.11	**2.60**
1000	1.96	0.62	**3.15**	0.51	0.17	**3.00**
2000	3.93	1.38	**2.85**	0.97	0.29	**3.35**

Table 1: Client marshaling performance in ms

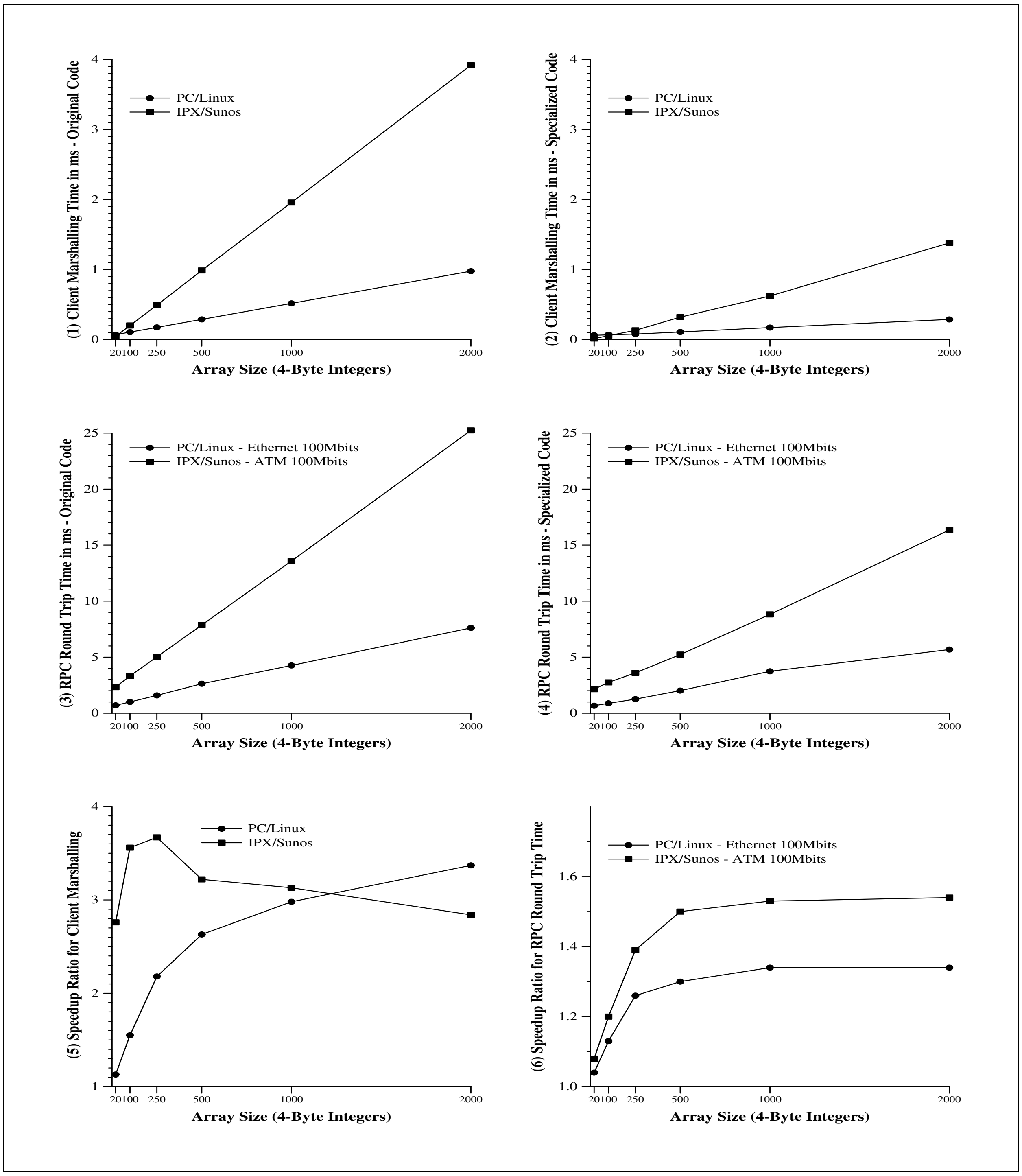

Figure 6: Performance Comparison between IPX/SunOS and PC/Linux

Array Size	IPX/SunOs			PC/Linux		
	Original	Specialized	Speedup	Original	Specialized	Speedup
20	2.32	2.13	**1.10**	0.69	0.66	**1.05**
100	3.32	2.74	**1.20**	0.99	0.87	**1.15**
250	5.02	3.60	**1.40**	1.58	1.25	**1.25**
500	7.86	5.23	**1.50**	2.62	2.01	**1.30**
1000	13.58	8.82	**1.55**	4.26	3.17	**1.35**
2000	25.24	16.35	**1.55**	7.61	5.68	**1.35**

Table 2: Round trip performance in ms

Module	Generic	Specialized (array size)					
		20	100	250	500	1000	2000
client code	20004	-					
specialized client code		24340	27540	33540	43540	63540	111348

Table 3: Size of the SunOS binaries (in bytes)

Round-trip RPC. The application level benchmark results are detailed in Table 2. The specialized code runs up to 1.55 faster than the non-specialized one on the IPX/SunOS, and 1.35 on the PC/Linux. Similar to the micro-benchmark, the speedup decreases with the size of the data due to memory accesses. In addition to these memory accesses, the RPC includes a call to `bzero` to initialize the input buffer on both the client and server sides (hence it does not appear in the marshaling micro-benchmark.) These initializations further increase memory access overhead as the data size grows.

Code size. As shown in Table 3, the specialized code is always larger than the original one. The reason is that the default specialized code unrolls the array encoding/decoding loops completely. It should noticed that the specialized code is also larger for small array size. This due to the fact that the specialized code also contains some unspecialized generic functions because of error handling.

While loop unrolling increases code sizes, it also affects cache locality. An additional experiment was conducted on the PC to measure this effect. Since completely unrolling large loops may exceed the instruction cache capacity, we only partially unrolled the loop to adjust its body to the cache size. This transformation was done manually. As shown in Table 4, the resulting code exhibit a lower deterioration of performance as the number of elements grows.

In the future, such strategy to control loop unrolling is planned to be introduced in Tempo.

6 Discussion

In this section we discuss our experience in using Tempo for specialization (Section 6.1), the lessons learned from working with existing commercial code (Section 6.2), and the relevance of this kind of specialization for general system code (Section 6.3.)

Array Size	PC/Linux				
	Original	Specialized	Speedup	250-unrolled	Speedup
500	0.29	0.11	**2.65**	0.108	**2.70**
1000	0.51	0.17	**3.00**	0.15	**3.40**
2000	0.97	0.29	**3.35**	0.25	**3.90**

Table 4: Specialization with loops of 250-unrolled integers (times in ms)

6.1 Experience with Tempo

Tempo is a state-of-art program specializer under active development. Most of its features described in Section 4 were motivated by system code experiments, including the RPC reported in this paper.

Although priority so far has been given to the realization of program specialization technology (i.e., the partial evaluation "engine"), a suitable user interface is needed for the handling of real world code. Tempo allows the user to visualize the results of the analysis before specialization. Different colors are used to display the static and dynamic parts of a program, thus helping the user to follow the propagation of the inputs declared as known and assess the degree of specialization to be obtained. After specialization, the user can compare the original program with the specialized program, and decide whether appropriate reduction and residualization have been carried out.

As for any other optimizer, the programmer needs to have some knowledge about the optimization process itself. In the case of partial evaluation, one needs to know about fundamental concepts such as binding times.

6.2 Working with Existing Code

An important lesson learned in this experiment is that existing code is a challenge for an optimization technique such as partial evaluation. Indeed, like any other optimization technique, partial evaluation is sensitive to various program features such as program structure and data organization. As a result, specializing an existing program requires an intimate knowledge of its structure and algorithms. It also requires the programmer to estimate what parts of the program should be evaluated away. This is in contrast with a situation where the same programmer both writes and specializes some code: he can structure it with specialization in mind.

Careful inspection of the resulting specialized code shows few opportunities for further optimization without major restructuring the RPC code. However, Tempo is not the panacea and we occasionally had to slightly modify the original source code in order to obtain suitable specializations. Most modifications were to make the some value available for specialization.

More specifically, on the client side, there exists a variable named `inlen` that stores the length of already decoded data. When decoding the input buffer, the variable `inlen` is dynamic because the remote procedure call may fail; ill-formed received data must also be guarded against. However, if no error occurs, `inlen` contains the size of the expected result data, which is usually fixed unless the data structures have variable length. In this case, we know the expected length of the input message, noted `expected_inlen`. In the unmarshaling part, the following code skeleton

```
inlen = <dyn>;
<statements>;
```

is manually rewritten as

```
inlen = <dyn>;
if (inlen == expected_inlen) {
      inlen = expected_inlen;
      <statements>;
} else
      <statements>;
```

As a result, in the "then" branch, the known value of the variable `expected_inlen` is assigned to `inlen`; the following statements may now exploit this additional value. Yet, the "else" branch preserves the semantics: it handles the general case.

The actual value for `expected_inlen` may be computed at specialization time with a dummy encoding-call to the generic encoding/decoding function. We are thus able to specialize the client decoding. This situation actually occurred twice and thus does not contradict our claim of automatically treating existing system code. Overall, the current version of Tempo is very successful in the specialization of system code such as Sun RPC.

6.3 General Applicability

We consider the Sun RPC to be representative of existing system code, not only because it is mature, commercial, and standard code, but also because its structure reflects production quality concerns as well as unrestrained use of the C programming language.

The examples that we highlighted in Section 3 (i.e., dispatching, buffer overflow checking, handling of exit status) are typical instances of general constructions found in system code. The fact that Tempo is able to automatically specialize them reinforces our conviction that automatic optimization tools like partial evaluators are relevant for system code production.

7 Related Work

The specialization techniques presented in this paper relate to many studies in various research domains such as specific RPC optimizations, kernel level optimizations, operating system structuring, and automatic program transformation. Let us outline the salient aspects of these research directions.

General RPC optimizations. A considerable amount of work has been dedicated to optimize RPC (see [32, 17, 36, 25, 24]). In most of these studies, a fast path in the RPC is identified, corresponding to a performance-critical, frequently used case. The fast path is then optimized using a wide range of techniques. The optimizations address different layers of the protocol stack, and are performed either manually (by rewriting a layer), or by a domain-specific optimizer.

Marshaling layer optimizations. Clark and Tennenhouse [6] were the first to identify the presentation layer as an important bottleneck in protocol software. They attribute it to up to 97% of the total protocol stack overhead, in some practical applications. Rather than optimizing an existing implementation, they propose some design principles to build new efficient implementations.

Hoschka and Huitema [15] convert marshaling code from a table-driven implementation to a procedure-driven implementation. In the table-driven implementation, a generic interpreter selects among several elementary decoding procedures, organized as a function table. The procedure-driven implementation is a straight sequence of code specialized for a given compound type. Their transformation does not include complex optimizations. Rather, they are interested in the time *vs.* space tradeoff between the two implementations.

O'Malley *et al.* [26] present a universal stub compiler, called USC. As opposed to XDR, which converts between a fixed host format and another fixed external representation, USC converts data between two user-specified formats. USC integrates several domain-specific optimizations, resulting in much faster code than the one produced by XDR. However, in order to perform these aggressive optimizations, USC imposes some restrictions over the marshaled data types: types such as floating point numbers or pointers are not allowed. In fact, USC is not designed for general argument marshaling, but rather for header conversions and interfacing to memory-mapped devices.

Blackwell [4] manages external data formats which allow variable encoding, such as Q.93B [13] or ASN.1 [16]. In these representations, each data field is tagged to indicate its actual format, chosen between several possible ones. Blackwell builds a special-purpose on-line compiler, which generates specialized marshaling code for the formats that are encountered frequently at run time. The optimizations integrated in this compiler aggressively exploit domain-specific information, such as the absence of aliases, the ability to reorder copy operations of distinct fields, or the alignment properties which make it possible to collapse several adjacent fields into a single word.

All these studies require one to build a special-purpose code generator, with a complexity ranging from an ad-hoc template assembler to a full, domain-specific, optimizing compiler. In contrast, we take the stubs generated by an existing stub compiler, and derive the specialized stubs with Tempo, a general program specialization tool.

Kernel-level optimizations. It is well recognized that physical memory copy is an important cause of overhead in protocol implementation. Finding solutions to avoid or optimize copies is a constant concern of operating system designers. For instance, copy-on-write [12] was the technique which made message passing efficient enough to allow operating systems to be designed based on a micro-kernel architecture [30, 31]. Buffers are needed when different modules or layers written independently for modularity reasons have to cooperate together at run time. This cause of overhead has been clearly demonstrated by Thekkath and Levy in their performance analysis of RPC implementations [36]. Recent proposals in the networking area explore solutions to improve network throughput and to reduce latency. Madea and Bershad propose to restructure network layers and to move some functions into user space [20]. Mosberger *et al.* describe techniques for improving protocols by reducing the number of cycles stalled to wait for memory access completion [24].

Manual specialization. In a first step, operating systems specialization has been performed manually in experiments such as Synthesis [28, 21], and Synthetix [27]. Manual specialization, however, tends to compromise other system properties such as maintainability and portability. Furthermore, manual specialization is typically uniquely tailored to each situation and therefore requires a high degree of programmer skill and system knowledge. While tool-based specialization may not fit the traditional kernel development process, we see it as a natural next step for operating system development the same way compilers became useful programming technology decades ago.

Recently, a semi-automatic approach to program transformation has been developed; it extends the C language to include syntactic constructs aimed at building code at run time[9]. It has been used for a realistic system application, namely the packet filter [10]. This work demonstrates that exploiting invariants can produce significant speedups for demultiplexing messages. This result has been obtained at the cost of manually rewriting a new algorithm that adapts to the specific usage context.

Automatic program transformation. Program transformation has been used successfully for specializing programs in domains such as computer graphics [19]. The key point of program transformation is that it preserves the semantics of the program. Therefore, if the transformation process can be automated, the final code has the same level of safety than the initial program. Tempo relies on partial evaluation [7, 18], a form of program transformation which is now reaching a level of maturity that makes it possible to develop specializers for real-sized languages like C [8, 1] and apply these specializers to real-sized problems.

C-Mix is the only other partial evaluator for C reported in the literature. Unfortunately, the accuracy of its analyses does not allow it to deal with partially-static structures and pointers to these objects interprocedurally [2].

Extensible operating systems. Safety is a well-known problem encountered in extensible operating system when an extension code has to be down-loaded directly into the kernel. Recent extensible operating systems such as SPIN and Aegis have been designed with safety as a goal. In SPIN [3], extensions are written in a strongly-typed language (MODULA-3), which prevents possible invalid memory references. Aegis [11] relies mostly on system libraries executed at application level. Still, it uses Application-specific Safe Handlers (ASH) executed at kernel level in specific cases. The safety of ASHs is ensured by the use of the *software fault isolation* technique [37], which rewrites the binary code to insert software-based memory protection instructions.

The extensible nature of these operating systems suggests that a partial evaluator could be a useful tool to optimize the generic components to be down-loaded into the kernel.

8 Conclusion

We have described the specialization of Sun RPC using the Tempo partial evaluator. This is the first successful partial-evaluation based specialization of a significant OS component. The experiment consists of declaring the known inputs of the Sun RPC code and allowing Tempo to automatically evaluate the static parts of code at specialization time. Examples of known information include the number and type of RPC parameters.

There are three reasons why partial-evaluation based specialization is a significant innovation, in comparison to manual specialization [27, 23, 11]. First, Tempo preserves the source code at the programming level, thus preserving the safety and maintainability of a mature commercial code. Second, Tempo achieves speedup up to 3.75 in micro-benchmarks, and 1.5 in test programs. Close inspection of the specialized Sun RPC did not reveal obvious opportunities for further significant improvements without major code restructuring. Third, the successful specialization of commercial code is automatic and shows the promise of industrial application of Tempo.

We carried out experiments on two very different platforms, namely Sun 4/50 workstations running SunOS connected with a 100 Mbits/s ATM link, and 166 MHz Pentium machines running Linux, connected to a 100 Mbits/s Ethernet network. The differences of these platforms and the consistency of performance gains show a robust applicability of Tempo as a tool for partial-evaluation based specialization of layered operating systems code such as Sun RPC.

References

[1] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, San Francisco, CA, USA, June 1992. Yale University, New Haven, CT, USA. Technical Report YALEU/DCS/RR-909.

[2] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[3] B.N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP95 [35], pages 267–283.

[4] T. Blackwell. Fast decoding of tagged message formats. In *Fifteenth Annual Joint Conference of the IEEE Computer and Communication Societies*, San Francisco, CA, March 1996.

[5] G. Cabillic and I. Puaut. Stardust: an environment for parallel programming on networks of heterogeneous workstations. *Journal of Parallel and Distributed Computing*, February 1997.

[6] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, September 1990. ACM Press.

[7] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

[8] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

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

[10] D.R. Engler and M.F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In SIGCOMM96 [33].

[11] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [35], pages 251–266.

[12] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[13] ATM Forum. ATM user-network interface specification version 3.0, 1993.

[14] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunde. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[15] P. Hoschka and C. Huitema. Control flow graph analysis for automatic fast path implementation. In *Second IEEE workshop on the architecture and Implementation of high performance communication subsystems*, Williamsburg, VA, September 1993.

[16] ISO. Specification of abstract syntax notation one (ASN.1). ISO standard 8824, 1988.

[17] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software - Practice And Experience*, 23(2):201–221, February 1993.

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

[19] B.N. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, September 1987. EUUG.

[20] C. Maeda and B.N. Bershad. Protocol service decomposition for high-performance networking. In SOSP'93 [34], pages 244–255.

[21] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.

[22] Sun Microsystem. NFS: Network file system protocol specification. RFC 1094, Sun Microsystem, March 1989.

[23] 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 94–20, Department of Computer Science, The University of Arizona, 1994.

[24] D. Mosberger, L.L. Peterson, P.G. Bridges, and S.W. O'Malley. Analysis of techniques to improve protocol processing latency. In SIGCOMM96 [33].

[25] D. Mosberger, L.L. Peterson, and S.W. O'Malley. Protocol latency: MIPS and reality. Technical Report 95-02, Department of Computer Science, The University of Arizona, 1995.

[26] S. O'Malley, T. Proebsting, and A.B. Montz. USC: A universal stub compiler. Technical Report TR94-10, University of Arizona, Department of Computer Science, 1994. Also in Proc. Conf. on Communications Archi. Protocols and Applications.

[27] 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 SOSP95 [35], pages 314–324.

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

[29] R. Ramsey. *All about administering NIS+*. SunSoft, 1993.

[30] R.F. Rashid, A. Tevanian Jr., M.W. Young, D.B. Golub, R.V. Baron, D. Black, Bolosky W.J., and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[31] V. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX - Workshop Proceedings - Micro-kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, USA, April 1992.

[32] M.D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[33] *SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford University, CA, August 1996. ACM Press.

[34] *Proceedings of the 1993 ACM Symposium on Operating Systems Principles*, Asheville, NC, USA, December 1993. ACM Operating Systems Reviews, 27(5),ACM Press.

[35] *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5),ACM Press.

[36] C.A. Thekkath and H.M. Levy. Low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[37] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. In SOSP'93 [34], pages 203–216.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399