Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol

Gilles Muller, Eugen-Nicolae Volanschi, Renaud Marlet
IRISA / INRIA
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
{muller,volanski,marlet}@irisa.fr

Abstract

We report here a successful experiment in using partial evaluation on a realistic program, namely the Sun commercial RPC (Remote Procedure Call) protocol. The Sun RPC is implemented in a highly generic way that offers multiple opportunities of specialization.

Our study also shows the incapacity of traditional binding-time analyses to treat real system programs. Our experiment has been made with Tempo, a partial evaluator for C programs targeted towards system software. Tempo's binding-time analysis had to be improved to integrate partially static data structures (interprocedurally), context sensitivity, use sensitivity and return sensitivity.

The Sun RPC experiment files, including the specialized implementation, are publicly available upon request to the authors.

1 Introduction

Remote Procedure Call (RPC) is a protocol that makes a remote procedure look like a local one. A call to this procedure is done transparently on the local machine but the actual computation takes place on a distant machine.

Performance is a key point in RPC. A lot of research has been carried out on the optimization of the layers of the protocol [23, 5, 16, 24, 14, 18]. Many studies have been proposed, but they necessitate the use of new protocols, incompatible with existing standard such as Sun RPC.

The high genericity of the RPC implementation is an invitation to specialization. Our group is currently developing a partial evaluator for C, named Tempo [7]. It is targeted at realistic programs, as opposed to toy examples or especially (re)written programs. It is more specifically designed to treat industrial-strength system code. The RPC experiment described here has been one of the driving test examples of Tempo's recent research, design and implementation.

Our contributions are the following:

- We have automatically optimized the Sun RPC by reusing the existing software layers, and obtained a 1.35 speedup on complete remote procedure calls (including network transport). On the RPC protocol itself (only the client encoding of data before it is sent), the specialized code runs up to 3.75 times faster.
- We have shown that traditional binding time analysis (BTA) is not fine enough to successfully specialize system code such as Sun RPC, and illustrated the need for the improved BTA that has been implemented in Tempo.
- Concerning software engineering, we have illustrated the fact that partial evaluation is a very appropriate tool to suppress fine-grain modularity overhead in generic software.

This work[1] shows that partial evaluation is reaching a level of maturity that makes it suitable to treat realistic programs.

The paper is organized as follows. Section 2 gives an introduction to RPC and describes relevant implementation details on the Sun RPC. Section 3 explores specialization opportunities. Section 4 describes Tempo and the functionalities that were added to it in order to treat the RPC case. Section 5 gives some benchmarks on a real example. Section 6 discusses related work in the field of system software as well as in partial evaluation. Section 7 concludes and lists some future work.

2 The Sun RPC Standard Protocol

The Sun Remote Procedure Call (RPC) protocol was introduced in 1984 as a basis for the implementation of distributed services between heterogeneous machines. This protocol has become a standard in distributed operating systems design and implementation. It is notably used for implementing widespread distributed services such as NFS [17] and NIS [21].

Because large networks are often heterogeneous, distributed environments need to encode data and often rely on Sun XDR protocol (one of the components of Sun RPC). Examples of these environments are PVM [11] for a message passing model and Stardust [4] for a Distributed Shared Memory model.

The RPC implementation used in this paper is the 1984 copyrighted version of the Sun RPC.

2.1 The Layers

The RPC protocol provides one main functionality (see Figure 1): it makes a remote procedure look like a local one. It supplies an interface between a client (on the local machine) and a server (on the remote machine) through *stub* functions. Those functions are automatically generated from the signature of the called procedure.

The RPC relies on two kinds of operations.

(1) It *marshals / unmarshals* (*i.e.*, encodes / decodes) data from a local machine dependent representation to a network inde-

To appear in the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, June 1997, Amsterdam.

[1] This research is supported in part by France Télécom/SEPT grant 951W0009.

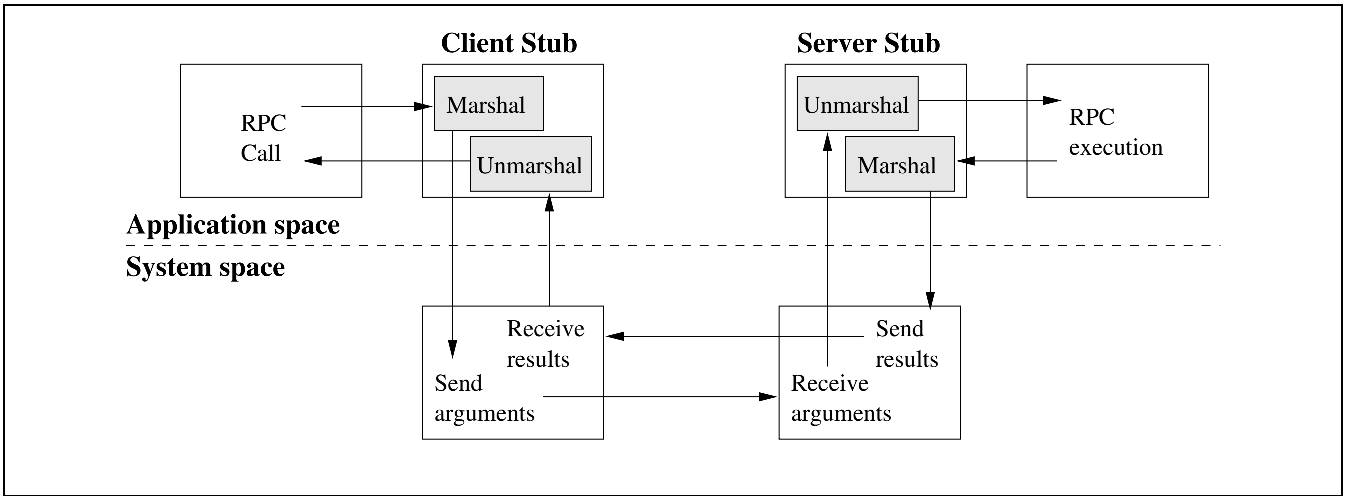

Figure 1: The RPC protocol

pendent one. The network data representation is standardized by the eXternal Data Representation (*i.e.*, XDR) protocol.

(2) It manages the exchange of messages through the network.

The RPC implementation is composed of a set of modular micro-layers, each one devoted to a small task such as managing the transport protocol (*e.g.*, TCP or UDP), or reading / writing data from / into the marshaling buffers. The micro-layers may have several implementations although, most of the time, given an application, the configuration never changes.

Let us consider a very simple example: a function `rmin` which takes two integers and returns their minimum, computed on a remote server. From the procedure interface specification, `rpcgen` (the RPC stub compiler) produces an assortment of source files that implement the `rmin` interface function on the client's side and the dispatch of procedures on the server's side (the same server may manage several procedures).

Figure 2 shows an abstract[2] execution trace of a call to `rmin`. The actual arguments are stored in a structure that is passed as a single argument.

2.2 The Internals

To describe the specialization opportunities we must first examine some RPC internals.

On the client end, a RPC call performs the following operations (see Figure 1): encoding of arguments into some output buffer, emission of the output buffer on the net, reception of the result in some input buffer, decoding of the input buffer and return of decoded result to the caller. The server performs similar actions, but in the opposite order: first decoding of arguments, then coding of the result.

Before doing any remote procedure call, the layers of protocol management must be initialized. In order to do that, the user must explicitly call the function `clnt_create()`, specifying a server name, the remote procedure identifier and version, and the chosen transport protocol. This initializes some state variables and stores some values at the beginning of the output buffer, forming a constant header which is the prefix of all procedure calls. In those variables lie our potential invariants.

[2] In the following code listings, irrelevant items are removed for clarity: some declarations, "uninteresting" arguments and statements, casts. Moreover, some structures may be flattened (only fields are shown).

The important "variables" (actually fields of a structure named `cu_data`) in the encoding and decoding are the following (see Figure 3):

- `cu_inbuf`: input buffer,
- `cu_outbuf`: output buffer,
- `cu_sendsz`: size of input buffer,
- `cu_recvsz`: size of output buffer,
- `cu_xdrpos`: size of output header.

Reading or writing buffers makes use of the following variables (fields of structure named `XDR`):

- `x_op`: flag saying if we are encoding or decoding,
- `x_base`: "base" pointer to start of buffer,
- `x_handy`: remaining space in the buffer,
- `x_private`: "current" pointer to buffer.

Reception uses this additional variable:

- `inlen`: number of received characters.

Function `xdrmem_putlong()` (see Figure 4) shows the use of `x_handy` and `x_private` for writing an long integer into the output buffer. Reading an integer follows a similar pattern.

2.3 Genericity in Sun RPC

The high genericity of the RPC implementation must be noted. It is already apparent in the execution abstract trace of Figure 2.

More specifically, a string argument that is passed to function `clnt_create()` specifies the choice of the transport protocol (*e.g.*, TCP or UDP). In our case, functions `clnt_create()` and `clnt_call()` will eventually call the more specific functions `clntudp_create()` and `clntudp_call()`. Similarly, the XDR encoding / decoding protocol has several implementations. In our case, with the XDR implementation using memory buffers, a generic call like `XDR_PUTLONG()` amounts to calling `xdrmem_putlong()`. All protocol parameterization are implemented with function pointers.

Besides, there is one single function to perform the encoding or the decoding of a given structure type. Only `x_op` ultimately decides if a value should be actually read or written (actual argument is a pointer). Figure 5 illustrates this on function `xdr_long()`.

```
arg.int1 = ...                       // Set first argument
arg.int2 = ...                       // Set second argument
rmin(&arg)                           // RPC User interface generated by rpcgen
  clnt_call(argsp)                   // Generic procedure call (e.g., TCP, UDP...)
    clntudp_call(argsp)              // UDP procedure call
                                     // Write procedure identifier
      XDR_PUTLONG(&proc_id)          //    Generic marshaling (e.g., to memory, stream...)
        xdrmem_putlong(lp)           //    Write long int into output buffer and check overflow
          htonl(*lp)                 //    Possible big/little endian conversion
      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...
              xdrmem_putlong(lp)     //      Write into output buffer and check overflow
                htonl(*lp)           //       Possible big/little endian conversion
                                     //    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...
              xdrmem_putlong(lp)     //       Write into output buffer and check overflow
                htonl(*lp)           //       Possible big/little endian conversion
```

Figure 2: Abstract trace of the encoding part of a remote call to `rmin`

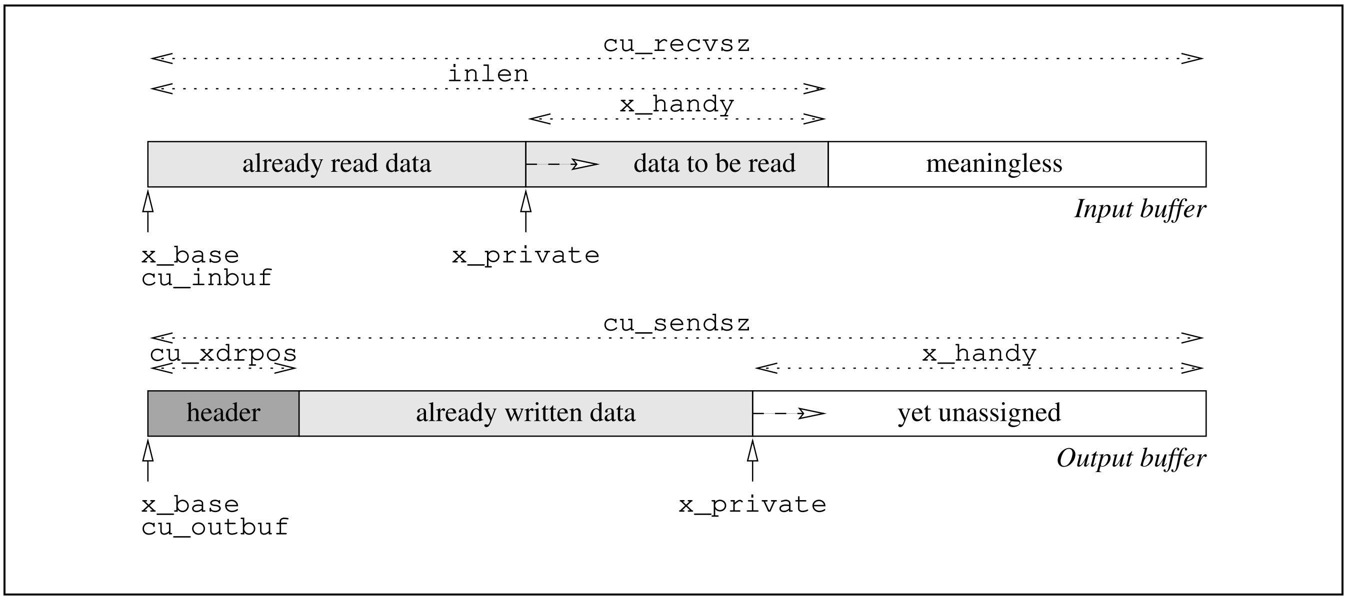

Figure 3: The input and output buffers

```
bool_t xdrmem_putlong(xdrs,lp)                // Copy long integer into output buffer
  XDR  *xdrs;                                 //     XDR handle
  long *lp;                                   //     Pointer to long integer to write
{
  if( (xdrs->x_handy -= sizeof(long)) < 0 )   // Decrement space left in buffer
    return FALSE;                             //     Report 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;                                // Report success
}
```

Figure 4: Writing a long integer into the output buffer: `xdrmem_putlong()`

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

Figure 5: Reading or writing a long integer: `xdr_long()`

For example a higher level functions like `xdr_pair()`, that calls twice `xdr_int()`, may be used to either read or write two integers, depending on `x_op` (see Figure 9).

3 Specialization Opportunities

Specialization is a process that exploits invariants in order to optimize a program. Seeing several generic software layers, a dispatch like in `xdr_long()`, and a buffer overflow check like in `xdrmem_putlong()`, the heart beat of the specialization hunter speeds up.

We only consider here the partial evaluation of the client[3] stub routine (*i.e.*, the actual function that performs the remote procedure call), as opposed to the specialization of a user's code that would make use of stub routines; we want the stub functions to be reusable in many contexts. In that sense, it may be seen as some kind of post-processing optimization to `rpcgen`[4].

3.1 Specializer *vs.* Optimizing Compiler

There is no doubt that a smart compiler can optimize a function like `xdr_int()` (see Figure 6). Using constant folding on the test condition, the body can be reduced to a single call, either to `xdr_long()` or `xdr_short()`, that in addition can be easily inlined.

[3] We are currently working on the specialization of the server stub routine, which is very similar to the client part, as far as the encoding/decoding is concerned.

[4] Note that we consider here only the specialization of the (user) library protocol layer of RPC, as opposed to the system protocol layer (in the kernel).

But there is nothing an optimizing compiler can do in cases such as `xdr_long()` (see Figure 5) where the actual value of `x_op` is computed somewhere above in the call tree. Clearly, we must resort to a partial evaluator.

3.2 The Invariants

Roughly speaking, off-line specialization relies a partition in two sets of all expressions and statements of the program. Those that depend only on known input are called *static* (they can safely be *evaluated* at specialization time); others, that may depend on yet unknown input, are called *dynamic* (they should be *residualized*).

The Sun RPC requires initializations before any remote procedure is called. There lies the known parameters we want to exploit in order to achieve specialization. They are given as arguments to the initialization function `clnt_create()`. After calling this function, which is done only once, the following variables will never vary later on; they are run-time *invariants*.

- `cu_recvsz` and `cu_sendsz` are assigned constant buffer size values.
- `cu_xdrpos` is assigned the length of the output buffer header, after it is initialized.
- `cu_inbuf` and `cu_outbuf` are assigned some dynamic memory space obtained by a call to `malloc()`.

In principle, those variables are not known until run-time (hence they are dynamic) and what we actually need here is a run-time specializer. The partial evaluator that we used (see section §4) does have a run-time specializer, but unfortunately it cannot treat all C constructions yet. Thus, we had to do the specialization in the context of the client initialization in order to be able to consider the

```
bool_t xdr_int(xdrs, ip)                    // Generic encoding or decoding of an integer
  XDR *xdrs;                                //     XDR handle
  int *ip;                                  //     Pointer to integer to read or write
{
  if( sizeof(int) == sizeof(long) )         // According to machine integer size
    return xdr_long(xdrs,(long *)ip);       //     Encode or decode as a long integer
  else                                      // Or
    return xdr_short(xdrs,(short *)ip);     //     Encode or decode as a short integer
}
```

Figure 6: Reading or writing an integer: `xdr_int()`

```
#define XDR_PUTLONG(xdrs,longp) \           // Generic encoding of a long integer
  (*(xdrs)->x_ops->x_putlong)(xdrs,longp)   //     Choice of protocol via function pointers
```

Figure 7: Encoding of a long integer: `XDR_PUTLONG()`

above variables as static.

Calling function `clnt_create()` at specialization time provides actual values for variables `cu_recvsz`, `cu_sendsz` and `cu_xdrpos`. But actual values for variables `cu_inbuf` and `cu_outbuf` may vary from one execution to another, due to the call to `malloc()`. However, those values will never be residualized because there are pointers. What matters to us is the *relative* arithmetic operations involving pointers to buffers array cells. Any initial value will do.

Finally, protocols such as XDR or transport protocol are sets of operations implemented as a structure of function pointers. For example, consider macro `XDR_PUTLONG` in Figure 7. Field `x_ops` of structure `XDR` holds the chosen XDR implementation; field `x_putlong` of structure `x_ops` holds a pointer to the integer encoding function; and calling the operation involves pointers dereferencing. All protocol structures are constant (hence static) after the client initialization is performed.

At first sight, treating the above variables as static might be considered as a trick. But is actually not uncommon in operating systems [20]. Using such properties of the code is the only way one can handle run-time invariants without having to do run-time specialization.

3.3 Binding Times of State Variables

The remaining variables reflect some state of the buffer encoding and decoding processes.

- `x_op` is always assigned a known value (whether to encode or to decode) before being used; it is static.
- `x_base` is assigned `cu_inbuf` or `cu_outbuf`, which may be considered static (as seen above).

Whereas a local `XDR` structure is defined for decoding, and discarded when it is finished, the same structure is reused for all encodings. Here is how this structure is assigned during the `clnt_create()` initialization (see Figure 3):

```
x_base = cu_outbuf;
x_handy = cu_xdrpos;
x_private = x_base;
```

Variables are thus assigned known, static values. Then, each time that a new encoding is performed, `x_handy` and `x_private` are reinitialized in the following manner:

```
start = x_base + cu_xdrpos;
last  = x_private + x_handy;
x_private = start;
x_handy   = last - start;
```

Whereas `x_private` is straightforwardly set to a pointer to the first cell after the buffer output header, the previous values of `x_handy` and `x_private` are used to compute the last position after the output buffer, from which the actual buffer size (*i.e.*, the new `x_handy` value) is recomputed. From the algorithm point of view, there exists a simpler way to compute the buffer size, but it is not the point here because what we want is to treat automatically existing code. What is important is that `x_private` and especially `x_handy` stay static throughout the code.

All those variables being static, we might expect to eliminate the dispatch on variable `x_op` and, when writing for output, buffer overflow check controlled by `x_handy`.

3.4 Specializing Common Cases

Since `inlen` is the size of the received data, it is intrinsically dynamic. The reason why `inlen` is unknown in general is that the remote procedure call may fail. Ill formed received data must also be guarded against. However, most of the time, `inlen` is the size of the expected result data, which is usually constant, apart from the case of variable length data structures.

There is a standard trick to deal with that, involving only minor rewriting. Suppose we know `expected_inlen`, the expected value for the input message length. Then a piece code that looks like

```
inlen = some dynamic value;
statements using inlen
```

may be manually rewritten as

```
inlen = some dynamic value;
if( inlen == expected_inlen) {
    inlen = expected_inlen;
    statements using (static) inlen
} else
    statements using (dynamic) inlen
```

```
bool_t xdrmem_putlong(xdrs,lp)                  // Static return value
  XDR  *xdrs;                                   //   Static & dynamic pointer to partially static structure
  long *lp;                                     //   Static pointer to a dynamic value
{
  if((xdrs->x_handy -= sizeof(long)) < 0)       // Use static facet of xdrs pointer
    return FALSE;                               //   Return static failure
  *(xdrs->x_private) = *lp;                     // Use dynamic facet of xdrs pointer
  xdrs->x_private += sizeof(long);              // Static & dynamic increment
  return TRUE;                                  // Return static success
}
```

Figure 8: Binding-time analysis of `xdrmem_putlong()`: *Static*, **Dynamic**, and ***Static & Dynamic***

Now, in the "then" branch of the condition, the static facet of the variable `expected_inlen` is transmitted to `inlen`, that the following statements may exploit. Yet, the "else" branch preserves the semantics; it handles the general case.

The actual value for `expected_inlen` may be computed at specialization time thanks to a dummy encoding call to the generic encoding / decoding function. We are thus able to specialize the client decoding of result.

4 Scaling up Tempo for the RPC

Since partial evaluation rose to a recognized field of computer science, starting ten years ago, the spectrum of program transformations has not changed much. However, what has drastically improved is the power of the analysis that trigger the transformations, not only treating more difficult constructions (higher order, states), but also yielding finer results.

The heart of off-line partial evaluators lies in the binding-time analysis (BTA). The BTA problem has a spectrum of solutions, from the trivial one (everything is dynamic) to finer analysis: the more static statements or expressions, the more computation can be factorized. Refinements of early BTA in functional programming languages have led to:

- *partially static data structures*,
- *flow sensitivity*: binding time of a variable may depend on program point,
- *context sensitivity*: several instances of a function with different arguments (and store) binding times may coexist (*i.e.*, binding-time polyvariance).

Because of the complexity of the C language, those extensions were not included in early partial evaluators for C [1, 2]. BTA was not really improved further because it seemed to fit encountered problems, which were mainly toy examples or carefully (re)written programs.

As explained in the following subsections, trying to run a traditional BTA on the RPC code failed miserably. Understanding the reasons why it failed led us to reconsider two facts that we had taken for granted:

- An expression must be *either* static *or* dynamic.
- Static expressions may evaluate to a non-liftable value, *i.e.*, a pointer, a structure, or an array, which do not have a textual representation in C. Such expressions need to be residualized. But then, the use of such an expression in a dynamic context has to be considered dynamic, forcing all other uses and corresponding definition to become dynamic as well.

Yet, realistic programs such as RPC make a heavy use of non-liftable values. That had to be addressed.

4.1 Tempo

Our group is developing a partial evaluator for C, named Tempo [7]. To make sure that the analysis and optimizations that it performs address realistic programs, Tempo has been targeted towards a specific and very demanding application area: system software. The Sun RPC case has been one of the driving test-examples of Tempo's research, design and implementation [25, 26].

Tempo is an off-line specializer [6]: partial evaluation is split into a preprocessing phase that performs alias, side-effect, binding-time and action (*i.e.*, program transformation) analysis and, given some input values, a processing phase which does code generation. Tempo supports traditional compile-time specialization as well as run-time specialization [8]. Both share the same common core analysis [7].

Tempo's BTA includes the above-mentioned refinements : partially static data structures (interprocedurally), flow sensitivity and context sensitivity. Though those features were already well-understood in other partial evaluation contexts like functional languages, they had to be adapted for C imperative programming.

We found actual uses for these features in our RPC case study. Partially static data structures are totally indispensable throughout the code. Additionally, context sensitivity is useful for the integer encoding function. This function is usually called with dynamic data, representing the RPC arguments. However, there is one encoding of a static integer in each sending: the marshaling of the procedure identifier. Differentiating between the two call contexts preserves a specialization opportunity.

Other needs in the BTA emerged only after more experience with systems applications. They are covered in the following subsections. Additional comparisons may be found in [13, 12].

4.2 Use Sensitivity

In most situations, the data structures used within the XDR layer are partially static. Typically, these structures are passed to a procedure by means of a pointer. If this pointer is static (*e.g.*, because the structure was allocated during the client creation), one would expect to statically access the known fields of the structure, *and* to dynamically access the unknown fields.

This is the case, for example, in `xdrmem_putlong()` where the buffer descriptor is passed via the static pointer `xdrs` (see Figure 4). The field `x_handy` is static, because the size of the message is statically determined. On the other hand, accesses to field `x_private` must be residualized, because the output buffer is filled with dynamic values.

Because of the dual behavior of the pointer `xdrs`, a traditional BTA would conservatively treat it as dynamic. This would inhibit the specializer to eliminate the test for overflow, resulting in poor

```
bool_t xdr_pair(xdrs, objp)          // Original function generated by rpcgen
  XDR  *xdrs;                        //    XDR handle
  pair *objp;                        //    Pointer to arguments structure
{
  if( !xdr_int(xdrs, &objp->int1) ) // Encode (or, potentially, decode) first argument
    return FALSE;                    //    Abort on failure
  if( !xdr_int(xdrs, &objp->int2) ) // Encode (or, potentially, decode) second argument
    return FALSE;                    //    Abort on failure
  return TRUE;                       // Report success
}
```

Figure 9: Specialized encoding (and, potentially, decoding) routine `xdr_pair()`

```
void xdr_pair_spec(xdrs,objp)        // Function now returns void
  XDR  *xdrs;                        //    XDR handle
  pair *objp;                        //    Pointer to arguments structure
{                                    // Overflow check eliminated
  *(xdrs->x_private) = objp->int1;  // Inlined call for writing first argument
  xdrs->x_private += 4u;             //    Point to next copy location in buffer
  *(xdrs->x_private) = objp->int2;  // Inlined call for writing second argument
  xdrs->x_private += 4u;             //    Point to next copy location in buffer
}                                    // Return code eliminated
```

Figure 10: Specialized encoding (but not decoding) routine `xdr_pair()`

optimization. This issue is crucial in realistic programs, that make extensive use of non-liftable values, manipulating large nested data structures including pointers and arrays.

Motivated by several examples like this one, an enhanced BTA has been implemented in Tempo. The conceptual problem was identified to be the *use insensitivity* [13] of traditional analyses, meaning that dynamic uses of a dual pointer pollute all the static uses. The new, *use-sensitive* BTA is able to take into account a dual binding time for a variable or structure field: the static facet is used in all static contexts, and the dynamic facet is used otherwise. In other words, a *static and dynamic* expression can be evaluated and exploited at specialization time. However, it is also present in the residualized program.

The analysis annotations produced by Tempo on function `xdrmem_putlong()` are shown in Figure 8. Note the different binding times due to different uses of the pointer `xdrs`, which allow the overflow condition to be reduced.

4.3 Return Sensitivity

In the same example, the function `xdrmem_putlong()` returns a (boolean) static value selected under a static condition. However, `xdrmem_putlong()` contains dynamic side-effects on the output buffer. Consequently, all the calls to this function must be residualized. Then, a traditional BTA considers that the return value is dynamic. This thus inhibits specialization of the caller. In our case, the caller (actually `xdr_int`) is always testing the result of `xdrmem_putlong()` to eventually trigger an error. Residualizing the call forces the test to be kept as well. As a result, a significant improvement of the encoding process is lost: each single scalar buffer copy involves an additional and superfluous test.

An extension was added to Tempo in order to successfully specialize such cases. In the implementation, the program is automatically rewritten, so as to return the result through a global static variable. The rewritten function becomes a procedure (`void` function), and is called in the residual program for the dynamic side effects. The expression containing the call is specialized with respect to the returned static value.

As an example, consider the original `xdr_pair()` function generated by `rpcgen` (see Figure 9). The specialized version is shown in Figure 10. Syntax has been cleaned up and standard compiler optimizations like copy propagation have been performed manually in order to make the specializer output readable. Note that the return code has been eliminated and that the calls to `xdr_int()` have been specialized (no more overflow check) and inlined. The specialized caller (function `clntudp_call()`, not shown), also exploits the known returned `TRUE` value (no buffer overflow). Note also that the static XDR implementation with function pointers is reduced and inlined.

4.4 User Interface

In order to treat real-size applications, special support had to be added in Tempo to ease the specialization "tuning" phase.

In principle, an off-line specializer has the advantage of a certain predictability, in the sense that the program transformations are decided at analysis time, before specialization actually takes place. The results of the analysis phase have to be output in a easy-to-read format. Currently, Tempo represents various kinds of information in a colored picture of the program, visualized in MIME format through an `emacs` interface. This information includes: binding times, polyvariance, aliases, side-effects, used global variables, and program transformations (*i.e.*, actions). Colors provide the same kind of information as can be seen in Figure 8.

Array Size	Original	Specialized	Speedup	Folded	Speedup
20	0.047	0.017	**2.75**	–	–
100	0.20	0.057	**3.50**	–	–
250	0.49	0.13	**3.75**	–	–
500	0.99	0.30	**3.30**	0.26	**3.80**
1000	1.96	0.62	**3.15**	0.53	**3.70**
2000	3.93	1.38	**2.85**	1.13	**3.50**

Table 1: Client marshaling performance (in milliseconds)

Array Size	Original	Specialized	Speedup
20	2.32	2.18	**1.05**
100	3.32	2.89	**1.15**
250	5.02	4.02	**1.25**
500	7.86	5.99	**1.30**
1000	13.58	10.05	**1.35**
2000	25.24	18.80	**1.35**

Table 2: Round trip RPC call performance (in milliseconds)

4.5 Further Improvements

A current limitation of Tempo forces all the instances of a same structure type to have the same binding time for a specific field and program point. This approximation greatly simplifies the implementation of both the analysis and the specializer. Furthermore, from an intuitive point of view, we expected all instances of a structure type to follow a common behavior in system programs. Typically, all file descriptors should have the same static fields (likely, the open mode, the permissions, etc.). However, this intuitive uniformity is broken in some cases, including network software. In our RPC case, the behavior of system descriptors tends to be different between the send path and the receive path. We thus had to split the client encoding and decoding into two different functions, that are specialized independently. Removing this limitation is being worked on.

The alias analysis implemented in Tempo is very similar to the *points-to* model of aliasing [9, 22]. It is interprocedural, flow-sensitive but context-insensitive. For specialization, the finer the alias analysis, the less (possibly dynamic) wrong target locations are considered, hence the less conservative (*i.e.*, dynamic) binding times are assigned. In this experiment, the computed alias information was fine enough not to prevent specialization to take place. However, other on-going experiments with system code suggest that context sensitivity as well as exact structure layout (as opposed to simply field names) might be needed for alias analysis.

5 Benchmark

This section analyzes the performance we obtained by specializing the RPC layer with Tempo.

Our test program emulates the behavior of scientific parallel programs that exchange large chunks of structured data. The test program loops on a simple remote procedure call that sends and receives an array of integers. We have made two different kinds of measurements, comparing the Tempo specialized client with the non specialized one: (i) a micro-benchmark of the sending (*i.e.*, encoding) layer in the client, and (ii) a full round-trip remote procedure call. The interest of this second experiment is to take into account architectural machine behavior such as cache, memory and network bandwidth which highly affect global performance. Additionally, we consider different array sizes.

The whole client test program specialized by Tempo is about 1500 lines long (without comments), including 500 lines of declarations. The reason why it might seem large is that a lot of initializations are needed and that there exist many small functions (which are seldom used all at the same time) due to the generic micro-layer structure.

Measurements have been done on two Suns 4/50 connected with a 100 Mbits ATM link. All programs have been compiled using `gcc`, with option `-O2`.

Summary of Results

On the encoding layer, the specialized code is up to 3.75 times faster than the non specialized one (see table 1). On the round-trip RPC execution, we have a speedup of up to 1.35 (see table 2). It must be noted that in our experiment, only the client program is specialized. It is realistic to think that the speedup can be doubled by also specializing the server.

Micro-benchmark

Table 1 gives results of the micro-benchmark. Speedup varies from 2.35 to 3.75. Surprisingly, the speedup decreases with the size of the array of integers. When the array size grows, most of the marshaling time is spent in encoding the array of integers. 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. The reason for which the speedup decreases with the size is that, on our test machine, instructions execution time is dominated by memory accesses.

During specialization, the array encoding loop is unrolled. Unrolling large loops is sometime nasty because it breaks the locality of instructions accesses in the cache. In order to analyze unrolling effect on the cache, we have partially folded back the code into a loop (see two last rows of table 1). We have kept an unrolled loop body corresponding to an array size of 250 (*i.e.*, array sizes 500, 100 and 2000 correspond to 2, 4 and 8 iterations of the loop). This operation is manual. When the size of the array grows, the folded loop becomes faster than the unrolled generated one. This clearly shows the break of the cache locality.

Round-trip RPC

The specialized round-trip RPC runs up to 1.35 faster. Like for the micro-benchmark, the speedup decreases with the size of the data because of memory accesses. In addition to these memory accesses, the RPC implementation includes a call to `bzero()` that initializes the input buffer on both the client and server sides. These initializations further increase memory access overhead as the data size grows. Note that the marshaling micro-benchmark code does not contain any call to `bzero()`.

It must be noted that the ATM cards and drivers used in our experiment are three years old and quite inefficient (100 Mbits) compared with up to date products, in term of latency and bandwidth (155 Mbits, and even 622 Mbits). Therefore, we expect to have much better results in the future.

6 Related Work

Partial Evaluators for C. To our knowledge, the only other partial evaluator for C programs is *C-Mix* [2]. Like Tempo, *C-Mix* is an off-line evaluator, based on inter-procedural analyses, and able to deal with complex data structures and side-effects.

While being a powerful tool, *C-Mix* was not specially written to deal with system programs. More precisely, the following approximations of its analyses make it unsuitable for our XDR example. The BTA of *C-Mix* is program-point insensitive, which means that a variable is considered dynamic as soon as it is dynamic in a marginal part of the program (*e.g.*, an exception treatment). Also, the BTA is mono-variant, resulting in a unique analysis of each function, with respect to the union of all the calls in the program. Furthermore, structure splitting (approach used in *C-Mix* to treat partially static data structure) is intraprocedural for arguments, which also makes it unsuitable for the Sun RPC case.

Another practical aspect is that *C-Mix* systematically duplicates code after a static conditional. This improves the precision of the BTA, but can easily cause an exponential code explosion, which is much more difficult to control than the code duplication coming from loop unrolling.

Finally, *C-Mix*' BTA is use-insensitive. As a consequence, it will systematically consider as dynamic any pointer to a partially-static data structure. In order to circumvent this problem, *C-Mix* attempts at automatically splitting such structures into a static component and a dynamic one. However, this strategy seems to be applicable only in some particular cases. More importantly, structure splitting would modify the global type declarations which belong to the operating system's interface. This makes it impossible to separately specialize an application module, and, in particular, our XDR example.

General RPC Optimizations. A considerable amount of work has been dedicated to optimize existing RPC implementations (see for example [23, 16, 24]). In 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. Some of these consist of manual optimizations on a specific layer of the RPC protocol stack. Our approach aims precisely at automating such optimizations.

Other techniques aim at minimizing the operating system overhead in the critical path, typically by eliminating some context switches or data copies. Even in the cases where data copies cannot be eliminated, they are eventually replaced by cheaper operations, like page re-mapping. All these techniques are orthogonal to our study, and should indeed give best results when combined with our kind of optimizations.

Optimizing Stub Compilers. Clark and Tennenhouse [5] 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. Among those principles, the Application Level Framing (ALF) and the Integrated Layer Processing (ILP) are directly relevant to the presentation layer.

Thekkath and Levy [24] generate argument marshaling code at run-time, when a client is bound to a server. This code, especially built for the given client-server pair, is obtained by assembling simple, hand-generated code templates, corresponding to elementary data types. Their use of dynamic code generation is not targeted to build very efficient code by exploiting run-time information. Rather, they observe that, by dynamically generating this code, and executing it in the kernel, arguments can be directly copied to the network buffer. In other implementations, marshaling code is running in user space, so it must first assembles the arguments in a user-level buffer, which is then copied by the kernel into the network buffer. Another difference from our study is that they generate this specialized code only on the send path. For the receive path, a generic, user-level marshaling code is executed.

Hoschka and Huitema [14] convert marshaling code from a table-driven implementation to a procedure-driven implementation. In the former, a generic interpreter is selecting among several elementary decoding procedures, organized as a function table, while the latter 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 decision.

O'Malley *et al.* [18] present another stub compiler, called USC. As opposed to XDR, which converts between a fixed host format and another fixed extern representation, USC is able to convert 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 this 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 [3] manages external data formats which allow variable encoding, such as Q.93B [10] or ASN.1 [15]. In these representations, each data field is tagged to indicate its actual format, chosen between several possible ones. Since unmarshaling code cannot be generated at compile time, Blackwell builds a special-purpose on-line compiler, which generates specialized marshaling code for the formats that are frequently encountered 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 building 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.

7 Conclusion and Future Work

This experiment has taught us several things.

Partial evaluation can be applied to realistic industrial-strength programs and yields non-trivial results. We automatically obtained a 1.35 speedup on complete (including network transport) remote procedure calls and a 3.75 speedup on the pure marshaling process. Some work is still required to solve a few problems. However, first results are very encouraging. We can now consider the automation of previous operating systems specialization that have been obtained manually [20, 19].

We are currently working on the following improvements. As for RPC, we are working on the specialization of the server. The hypothesis are similar to those of the client. We also plan to specialize the lower level network layers integrated in the system kernel, such as sockets and UDP. As for Tempo, we are considering removing the constraint that give the same binding time to all instances of the same data structure and providing a finer control over the loop unrolling. Improving the Tempo's alias analysis has less priority but is nonetheless unavoidable.

We observed that finding potential invariants and opportunities of specialization requires a good knowledge of the application domain. This observation is coherent with other experiments realized in our group. Specialization of complex real cases cannot be totally automated. More precisely, heavy analysis and transformations can be automated, but there are some cases where it must be guided or helped by a expert in the application domain. This stresses the importance of a user-friendly interface (see §4.4).

This experiment should encourage people to write (or keep on writing) generic applications, letting partial evaluation take care of performance issues. In particular, in the operating systems domain, people should keep on trying to write generic modules without worrying too much about performance. Thanks to partial evaluation, adaptability, maintainability and reuse, should be considered more important than immediate efficiency.

Partial evaluation is reaching a relative level of maturity. Still, acknowledged successes in a realistic context are very scarce. While continuing to explore new and indispensable theoretical basis, partial evaluation community must realize the importance of large scale experiments.

Acknowledgments

The authors would like to thank the other designers and implementors of Tempo (Charles Consel, Jacques Noyé, Luke Hornof, Julia Lawall, Scott Thibault, François Noël, Alan Sayle, Sandrine Chirokoff *et al.*) for fruitful discussions, patient attention, and unsparing efforts.

References

[1] L.O. Andersen. Binding-time analysis and the taming of C pointers. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 47–58. New York: ACM, 1993.

[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] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[9] M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256. ACM SIGPLAN Notices, 29(6), June 1994.

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

[11] 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.

[12] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[13] L. Hornof, J. Noyé, and C. Consel. Accurate partial evaluation of realistic programs via use sensitivity. Research Report 1064, IRISA, Rennes, France, June 1996.

[14] 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.

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

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

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

[18] 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.

[19] 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. ACM Operating Systems Reviews, 29(5),ACM Press.

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

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

[22] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22. ACM SIGPLAN Notices, 30(6), June 1995.

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

[24] 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.

[25] E.N. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: the RPC case study. In *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, pages 24–28, Tucson, AZ, USA, February 1996.

[26] E.N. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noyé, and C. Pu. A uniform automatic approach to copy elimination in system extensions via program specialization. Research Report 2903, INRIA, Rennes, France, June 1996.