Automatic Optimization of the Sun RPC Protocol Implementation via Partial Evaluation

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 an experiment of using partial evaluation on a realistic program, namely the Sun commercial RPC protocol. RPC is a highly generic software that offers several opportunities of specialization. We used Tempo, a partial evaluator for C programs targeted towards system software.

Our specialized marshaling layer is up to 3.5 times faster than the non-specialized one. On a complete RPC call, we have a speedup of 18%.

This work shows that partial evaluation is reaching a relative level of maturity: it can now be applied to real system software.

Keywords: partial evaluation, generic system software, RPC protocol.

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 various layers of the protocol [14, 3, 8, 15, 10], but they lead to protocols which are incompatible with an existing standard such as Sun's RPC. Moreover, even reimplementation of the present protocol could lead to incompatibility because of existing but yet accepted bugs (i.e. features). Our point is that the high genericity of Sun's standard RPC implementation is an invitation to specialization.

Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their inputs [4]. Our group is currently developing a partial evaluator for C named Tempo [5]. It is targeted at realistic programs, as opposed to toy examples or specially (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 optimized the client part of the RPC protocol, reusing the existing software layers. On the marshaling layer only, we run up to 3.5 times faster. On a complete RPC call, we have a 18% speedup. We expect to double that speedup when also specializing the server side.
- We have illustrated the fact that partial evaluation is a very appropriate tool to suppress modularity in generic software.
- Finally, we have shown that partial evaluation is reaching a level of maturity that makes it suitable to treat realistic (system) programs.

The remainder of the paper is organized as follows. Section 2 presents partial evaluation and describes the Tempo specializer. Section 3 recalls the internal architecture of the Sun RPC. Section 4 explores opportunities for partial evaluation and shows specialization results. Section 5 gives some benchmarks on a real example. The conclusions of our experiment opens up new perspectives.

2 Partial Evaluation

The conflict between modularity and performance is now a well-identified problem in system software development. Modularity in system code has many advantages such as maintainability, reusability, and adaptability. However, these advantages come at the expense of performance if the modular design is mapped directly to an implementation. Indeed, it often introduces some overhead due to operations such as data copying and format translation between layers, and heavily parameterized interface func-

```
arg.int1 = ...                          // Set first argument
arg.int2 = ...                          // Set second argument
rmin(&arg)                              // 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(longp)           //    Write in output buffer and check overflow
          htonl(lp)                     //    Switch between big and little endian (macro)
      xdr_min(argsp)                    // Stub function generated by rpcgen
                                        //    Write first argument
        xdr_int(&argsp->int1)           //       Machine dependent switch on integer size
          xdr_long(intp)                //       Switch between encoding and decoding
            XDR_PUTLONG(lp)             //       Generic marshaling to memory, stream... (macro)
              xdrmem_putlong(longp)     //       Write in output buffer and check overflow
                htonl(*lp)              //       Switch between big and little endian (macro)
                                        //    Write second argument
        xdr_int(&argsp->int2)           //       Machine dependent switch on integer size
          xdr_long(intp)                //       Switch between encoding and decoding
            XDR_PUTLONG(lp)             //       Generic marshaling to memory, stream... (macro)
              xdrmem_putlong(longp)     //       Write in output buffer and check overflow
                htonl(*lp)              //       Switch between big and little endian (macro)
```

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

tions. Modular design should rather be the basis of a stepwise refinement process to derive an implementation, instead of being directly mapped to an implementation.

We claim that this refinement process can be achieved by program specialization [4].

Overview of Tempo. In our experiment, we have used a partial evaluator of C named Tempo [5]. Form a source program and parts of its input, Tempo produces a specialized program. Tempo also allows run-time code generation [6].

Tempo is an off-line [4] specializer in that it processes a program in two steps: analysis and specialization. A known/unknown division of the input is given to the analysis phase. Pointer and side-effect information of the subject program are first computed. Then, a *binding-time analysis* (BTA) determines the computations which rely on the known parts of the input. Finally, an action analysis determines a program transformation for each construct in the program.

More precisely, every construct is assigned one of the following annotations:

- *static*: known at specialization time, hence can be exploited and eliminated in the specialized program,
- *dynamic*: only known a run time, hence must be residualized (i.e., must remain in the specialized program),
- *static & dynamic*: known at specialization time but forced to be residualized (usually for values such as pointers), hence can be exploited in specialization but must nonetheless appears in the specialized program.

The analysis phase handles partially-static structures, that is, data structures where only some but not all the fields are known. It also treats pointers to partially-static structures.

3 The Sun RPC Standard Protocol

The Sun Remote Procedure Call 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 [9] and NIS [13]. It implements mainly three functionalities:

(1) It makes a remote procedure look like a local one. It provides an interface between a client and a server through a *stub* functions. Those functions are automatically generated from the signature of the called procedure.

(2) It marshals / unmarshals (i.e. encodes / decodes) data from a local machine dependent representation to a network independent one.

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

```
xdr_long(xdrs,lp)                          //  Generic send or receive long integer
{
  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 int: `xdr_long()` — Key: *Static*, **Dynamic**, ***Static&Dynamic***

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

More precisely, the RPC implementation is composed of a set of 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/to the marshaling buffers. The micro-layers may have several implementations. Thus the overall implementation stays modular 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 call on the client's side and the dispatch of procedures on the server's side.

Figure 1 shows an abstract[1] execution trace of a call to `rmin`. The actual arguments have to be stored in a structure that is passed as a single argument.

4 Opportunities for Specialization

We consider here the partial evaluation of the client[2] 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`[3].

[1] In the following code listings, irrelevant items are removed for clarity: declarations, "uninteresting" arguments and statements, error handling, casts, some levels of functions calls.

[2] 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.

[3] 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).

The following subsections discuss various optimizations examples that a specializer can perform.

Elimination of dispatch constructs. A smart compiler can optimize the equality test in a function like

```
xdr_int(...)
{
  if (sizeof(int) == sizeof(long))
    return xdr_long(...);
  else
    return xdr_short(...);
}
```

It replaces the whole `if` construction with one of its branches. However, there is nothing it can do in cases such as `xdr_long()` (see figure 2) where, *a priori*, nothing is known about `xdrs->x_op`.

We must resort here to partial evaluation. The interprocedural analysis of the specializer takes note when `xdrs->x_op` is assigned a known (i.e. static) value. Thus, when we reach function `xdr_long()`, we may exploit the actual value of the field `x_op`, which is computed somewhere above in the call tree. As a result, the whole specialized function can be reduced to only one of the return constructs. Actually, the specialized `xdr_long()` function, being small enough, will disappear after inlining: the dispatch on `x_op` is totally eliminated.

Elimination of buffer overflow checking. The binding time annotations that Tempo produces for function `xdrmem_putlong()` (used to encode a long integer) are shown in figure 3.

The field `x_handy` stores the remaining space in the buffer. Like `x_op` in the example above, the analysis sees that it is first initialized (i.e. given a static value), then decremented and tested several times (for each call to `xdrmem_putlong` and related functions). Hence, it can be considered as static and the whole buffer overflow checking can be performed at specialization time; only the

```
xdrmem_putlong(xdrs,lp)                          //  Copy long int into output buffer
{
  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 int: `xdrmem_putlong()` — Key: *`Static`*, **`Dynamic`**, ***`Static&Dynamic`***

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

Figure 4: Specialized encoding routine `xdr_min`

buffer copy will remain in the specialized version (unless a buffer overflow is discovered).

Note the different uses of the pointer `xdrs`: in a static context with `xdrs->x_handy`, but in dynamic one with `xdrs->x_private`. It is assigned a "static & dynamic" annotations that allows both the overflow condition to be reduced and the copy code to be residualized.

Elimination of returned values. In the two above examples, all the return statements are static. This means that the result of calls to those functions may be known and exploited at specialization time.

Figure 4 shows the specialization of function `xdr_min` in the context of the specialized version of `rmin`. The type of the function has been turned to `void`; its result, which is always `TRUE` independently of dynamic `argsp` argument, is exploited to reduce an extra test in `clntudp_call` (not shown). Note the inlined specialized calls to `xdr_long`.

5 Benchmarks

This section analyzes the performance that can be expected in specializing the RPC layer with Tempo.

Due to the increasing speed of off-the-shelf processors and the increasing throughput of modern networks such as ATM, network of workstations can now be used as large scale multiprocessors. Because large networks are often heterogeneous, environments for communicating machine independent data involves encoding. Such environments often rely on Sun's XDR. Examples of these environments are PVM [7] for a message passing model and Stardust [2] for a Distributed Shared Memory model.

Our test program emulates the behavior of parallel programs which exchange large chunks of structured data. The test program loops on a simple RPC which sends and receives an array of integers. We have made two different kinds of measurements: (i) a micro-benchmark, which evaluates only the speedup of the sending (encoding) layer in the client, and (ii) a round-trip RPC benchmark which measures the real total time of a complete RPC 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.

The client test program specialized by Tempo is 1500 lines long (without comments), including 400 lines of declarations. Measurements have been done on two Suns 4/50 connected with a 100 Mbits ATM link. All programs have been compiled using `gcc`, with the option `-O2`.

Summary of results. On the sending layer (see table 1), the specialized code is up to 3.5 times faster than the non specialized one. On the round-trip RPC execution (see table 2), we have a speedup of up to 18%. It must be noted

Array size	250		500		1 000		2 000	
	time	speedup	time	speedup	time	speedup	time	speedup
Non specialized code	1 450	-	1 870	-	2 660	-	4 300	-
Specialized code	410	**3.5**	570	**3.5**	880	**3.0**	1 860	**2.3**
Loop optimized code	430	**3.4**	590	**3.5**	890	**3.0**	1 470	**2.9**

Table 1: RPC client marshaling performance (in microseconds)

Array size	250		500		1 000		2 000	
	time	speedup	time	speedup	time	speedup	time	speedup
Non specialized code	6.5	-	8.7	-	11.7	-	22.4	-
Specialized code	5.3	**18%**	7.3	**16%**	10.1	**13%**	19.1	**14%**
Loop optimized code	5.2	**20%**	7.0	**19%**	10.1	**13%**	17.4	**22%**

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

that in our experiment, only the client program is specialized. It is realistic to think that speedup can be doubled by also specializing the server.

Micro-benchmark. Speedups of the micro-benchmark are given in table 1; they varies between 2.3 and 3.5. Surprisingly, the speedup decreases with the size of the array of integers. When the size grows, most of the encoding time is spent in encoding the array of integers. If 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 since it breaks the locality of instruction accesses in the cache. To analyze unrolling effect on the cache, we have manually rewritten the unrolled code with a loop (see third row of table 1). When the size of the array grows, the loop optimized code becomes faster than the unrolled generated one. This clearly shows the break of the cache locality. Automatic do-unroll/do-not-unroll loops strategies are being investigated in the Tempo group.

Round-trip RPC. On the round-trip RPC (see table 2), the specialized code is between 13% to 18% faster than the non-specialized one. Similar to the micro-benchmark, the speedup decreases with the size of the data. However, the loop optimized code is always faster than the Tempo specialized one, with a maximum gain of 22%.

Finally, we must say that the ATM cards and driver used in our experiment are two years old and quite inefficient compared to up to date products, both in term of latency and bandwidth (i.e., 155 Mbits / 622 Mbits). Therefore, we expect to have even better results in the future.

6 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. It still requires some work to solve specific problems. However, first results are very encouraging. We can now consider the automation of previous operating systems specialization that have been obtained manually [12, 11].

We are working at the moment on the specialization of the server. The hypothesis are pretty much the same as for the client. We also plan to specialize the lower level network layers integrated in the system kernel, such as sockets and UDP. This raises complex issues since the specialized code has to cohabit in the kernel with the non-specialized one. Our final goal is to merge the stub and system layers and run the resulting optimized code in a *supervisor domain of protection* [1, 16]. The latter mechanism provides support for extensibility in Chorus.

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 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. Adaptability, maintainability, reuse, must unquestionably be the main key points.

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 *et al.*) for fruitful discussions, patient attention, and unsparing efforts.

References

[1] C. Bryce and G. Muller. Matching micro-kernels to modern applications using fine-grained memory protection. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 272–279, San Antonio, TX, USA, October 1995. IEEE Computer Society Press.

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

[3] 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.

[4] 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.

[5] 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.

[6] C. Consel and F. Noël. A general approach for runtime 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.