```
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`

1 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 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.

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 execution trace of a call to `rmin`. The actual arguments have to be stored in a structure that is passed as a single argument.

2 Opportunities for Specialization

We consider here the partial evaluation of the client 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`.

```
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***

```
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***

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 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.

```
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`

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`.

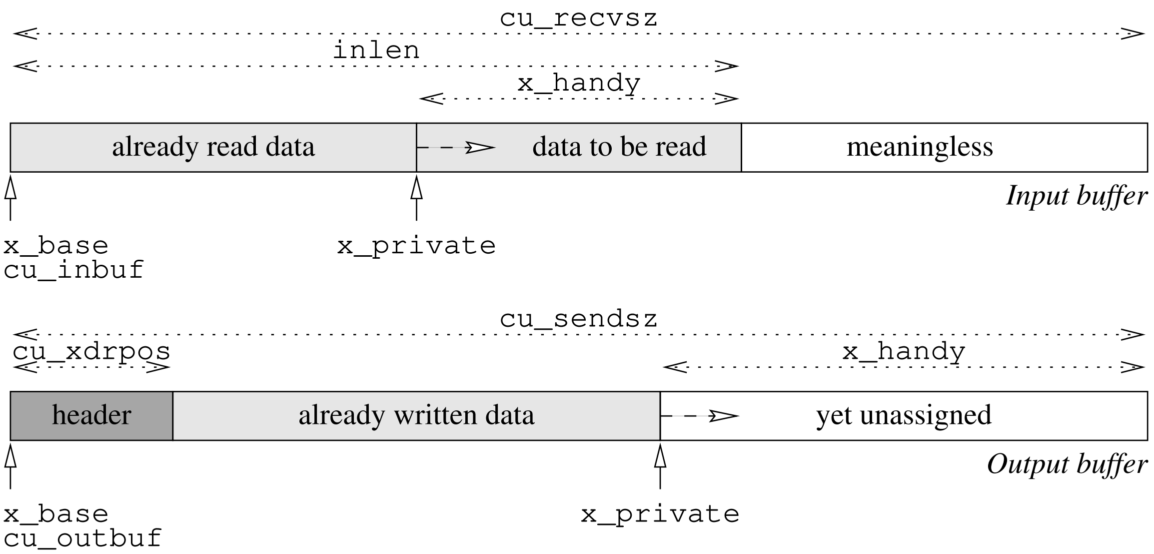

Figure 5: Buffer management