Fast, Optimized SUN RPC Using Automatic Program Specialization

Gilles Muller, Nic Volanschi, Renaud Marlet

COMPOSE Group

IRISA/INRIA - Rennes

Calton Pu, Ashvin Goel

Oregon Graduate Institute

Motivation of the Study

Why Optimize the Sun RPC ?

- Performance is crucial to RPC implementation
- The Sun RPC is a well recognized standard:
  - distributed system services: NFS, NIS
  - distributed computing environments: PVM, Stardust
- Automatic optimization is a valuable alternative to reimplementation:
  - reuse of existing code
  - optimized code remains compatible with the standard
  - manual optimization is error prone

Outline

- Opportunities for specialization in the SUN RPC
- Optimizing the RPC using the Tempo specializer
- Performance
- Conclusions and perspectives

Architecture of the SUN RPC

- The Sun RPC consists of a set of micro-layers
- The code has been designed in a highly generic way
- Micro-layers:
  - management of the transport protocol
  - representation of data (XDR)
  - marshaling / unmarshaling
  - buffer management

A Simple Example

- Compute the minimum of two integers / calling part of the client

```
arg.int 1 = ...
arg.int2 = ...
rmin (&arg)
  clnt_call (...)                    // protocol switch
    clntudp_call ()                  // generic procedure call
      xdr_pair()                     // rpcgen top function
        xdr_int ()                   // integer representation
          xdr_long ()                // encoding/decoding
            XDR_PUTLONG ()           // protocol switch
              xdrmem_putlong ()      // check of buffer overflow
                htonl ()             // big/little endian
```

Opportunities for Optimization

- Elimination of dispatch constructs
  - function pointers
    - ✔ protocols are chosen during initialisation
  - marshaling / unmarshaling
    - ✔ x_op is assigned a known value when starting encoding/decoding

```
xdr_long (xdrs, lp)
{
    if (xdrs->x_op == XDR_ENCODE)
        return XDR_PUTLONG (xdrs,lp)
    if (xdrs->x_op == XDR_ENCODE)
        return XDR_GETLONG (xdrs,lp)
    if (xdrs->x_op == XDR_FREE)
        return TRUE;
    return FALSE;
}
```

❑ Elimination of buffer overflow checking

✔ size of buffer is given by the procedure signature

```
xdrmem_putlong (xdrs, lp)
{
        if ((xdrs->x_handy -= sizeof (long)) < 0)
           return FALSE;
        *(xdrs->x_private) = htonl (*lp);
        xdrs->x_private += sizeof (long);
        return TRUE;
}
```

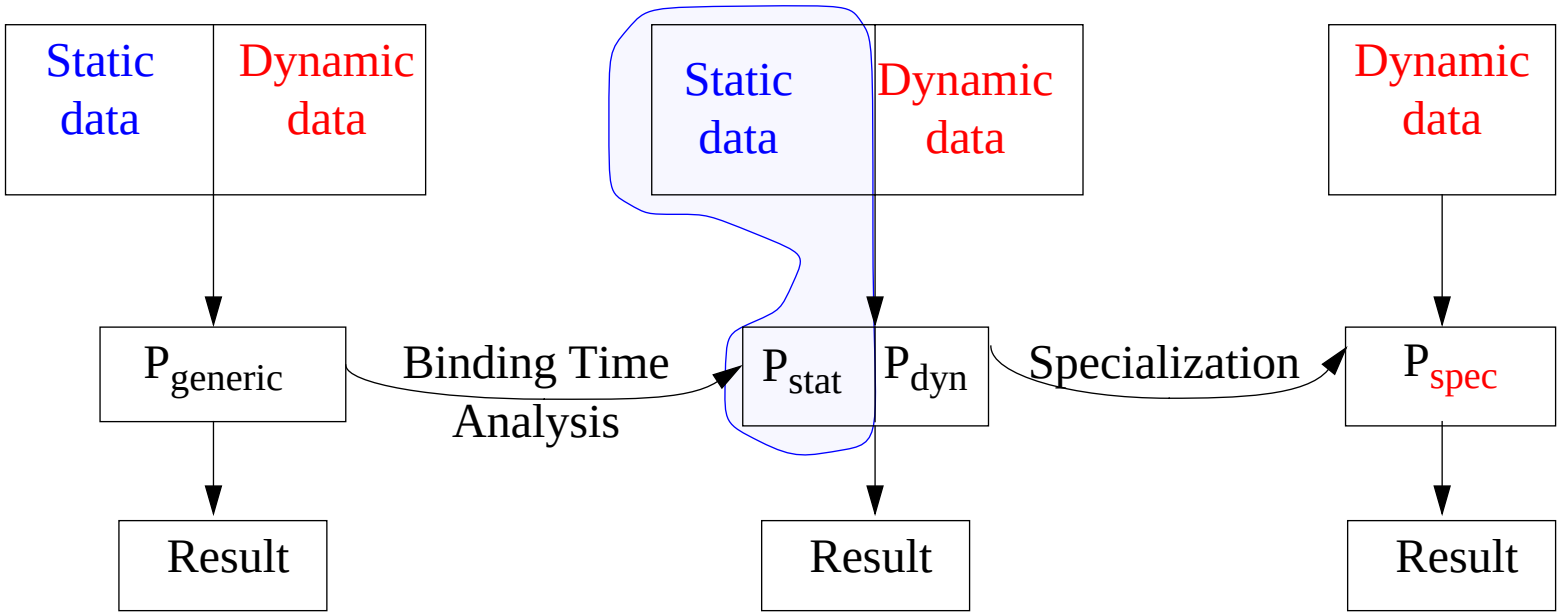
❑ Elimination of exit status

✔ error code can be computed statically

➢ The optimization process is systematic

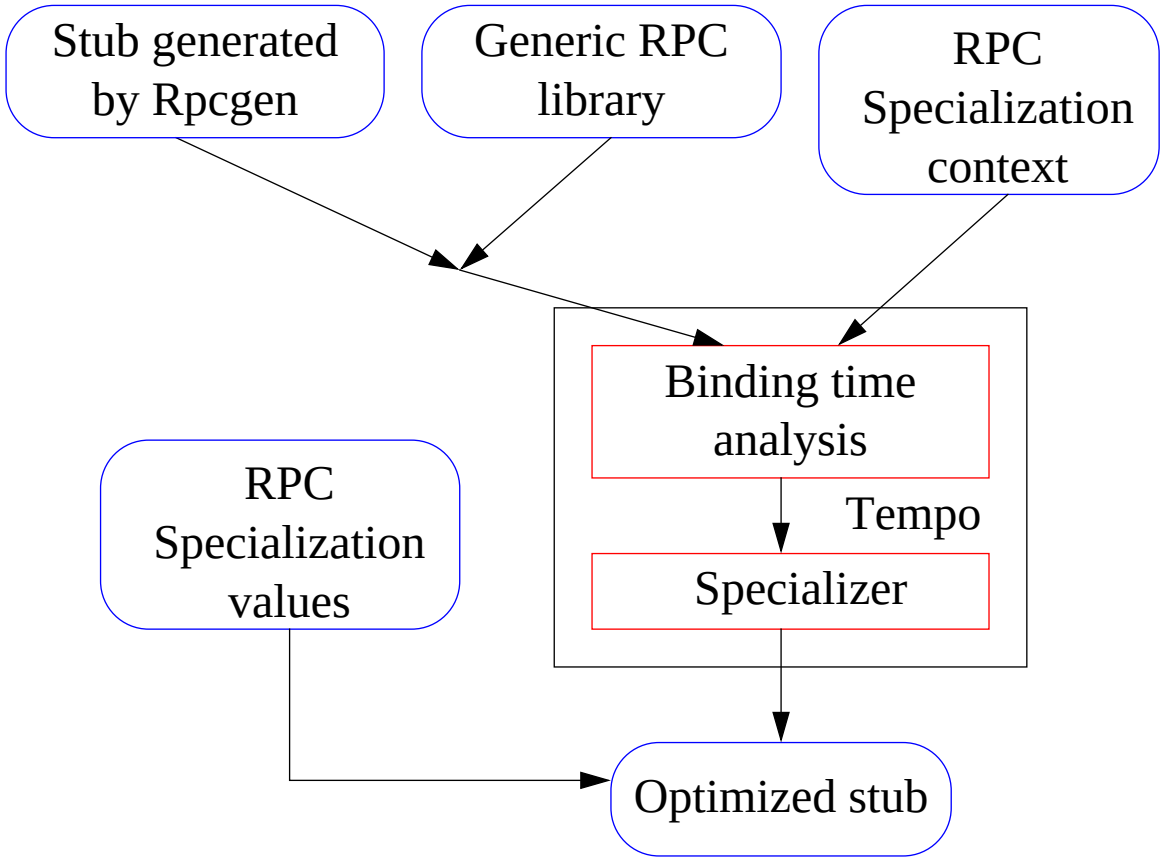
➢ We need a tool for doing it automatically

Automatic Specialization using the Tempo Partial Evaluator



- Binding time analysis:
  - program inputs are described to be either static or dynamic
  - binding time of inputs are propagated through the whole program
  - static parts are eliminated, dynamic parts are reconstructed in the specialized program

Specializing the RPC



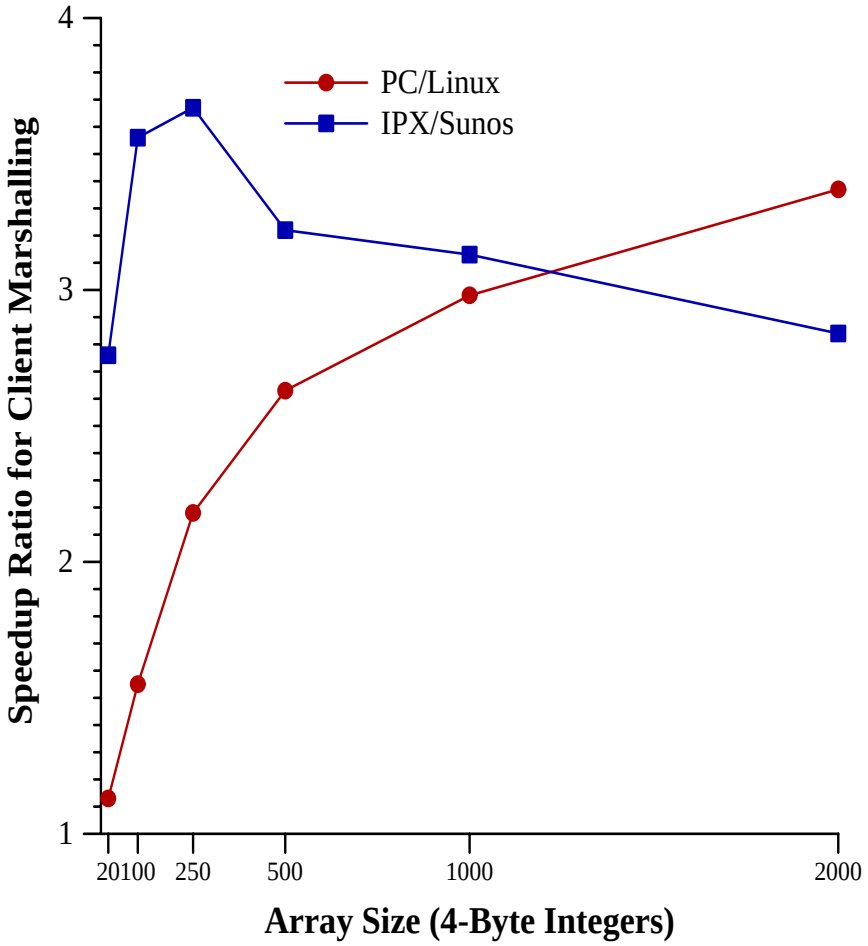
Binding-Time Analysed Code

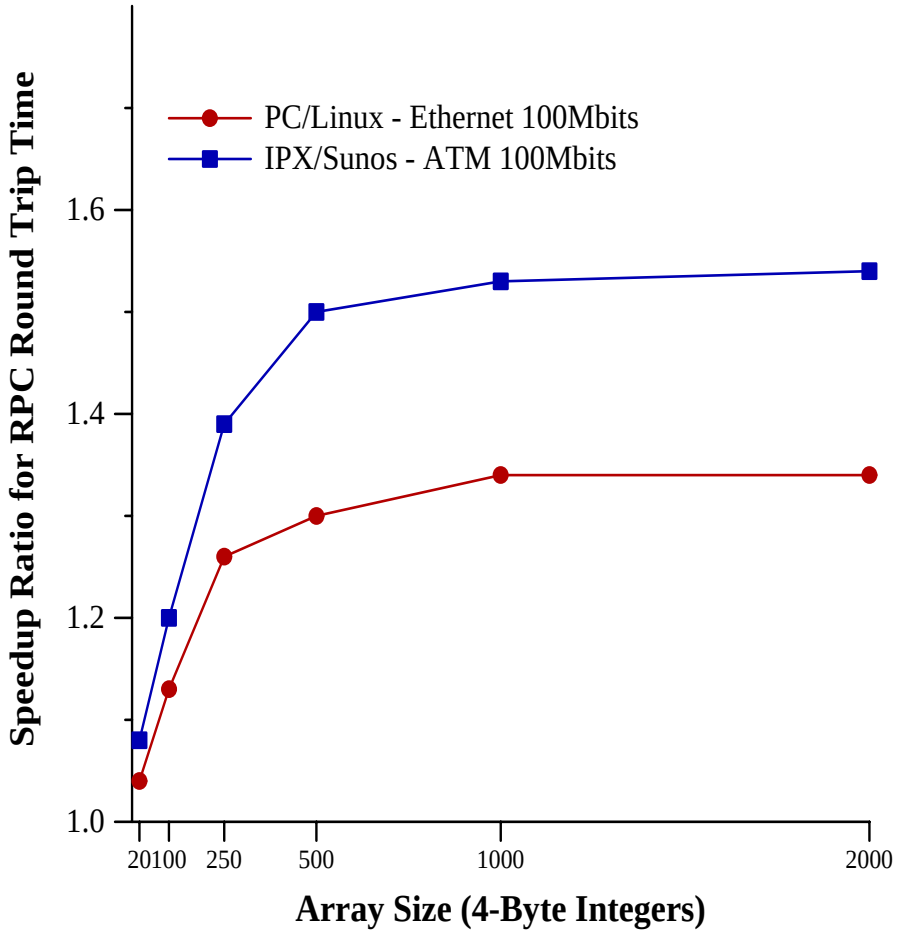
```
extern int Xdrmem_putlong (struct __tmp_struct2 *xdrs, int *lp)
{
    int *suif_tmp0;
    int suif_tmp1;
    char **suif_tmp3;

    suif_tmp0 = &(*xdrs/* cu_data.cu_outxdrs */).x_handy;
    suif_tmp1 = (unsigned int)(*suif_tmp0/*__tmp_struct2.x_handy */) - 4u;
    *suif_tmp0/* __tmp_struct2.x_handy */ = suif_tmp1;
    if (suif_tmp1 < 0)
      return 0;
    *((int *)(*xdrs/* cu_data.cu_outxdrs */).x_private)/* *__tmp_struct2.x_base */ = *lp /* *V, Proc */;
    suif_tmp3 = &(*xdrs/* cu_data.cu_outxdrs */).x_private;
    *suif_tmp3/* __tmp_struct2.x_private */ = 4u + suif_tmp3/* __tmp_struct2.x_private */;
    return 1;
}
```

Specialized Encoding Code xdr_pair()

```
{
  char **suif_tmp3;

  *((int *)(*xdrs).x_private) = (*((struct min1 *)argsp)).ent1;
  suif_tmp3 = &(*xdrs).x_private;
  *suif_tmp3 = *suif_tmp3 + 4u;
}

{
  char **suif_tmp3;

  *((int *)(*xdrs).x_private) = (*((struct min1 *)argsp)).ent2;
  suif_tmp3 = &(*xdrs).x_private;
  *suif_tmp3 = *suif_tmp3 + 4u;
}
```

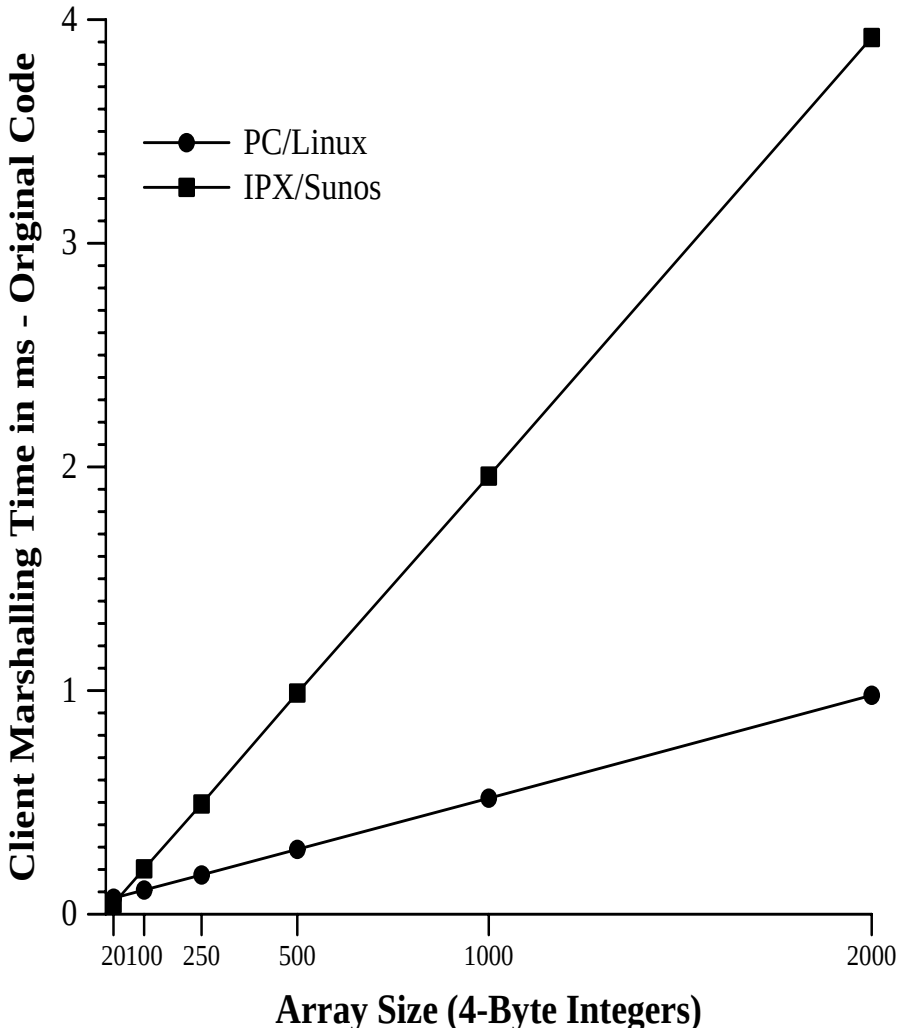
Speedups

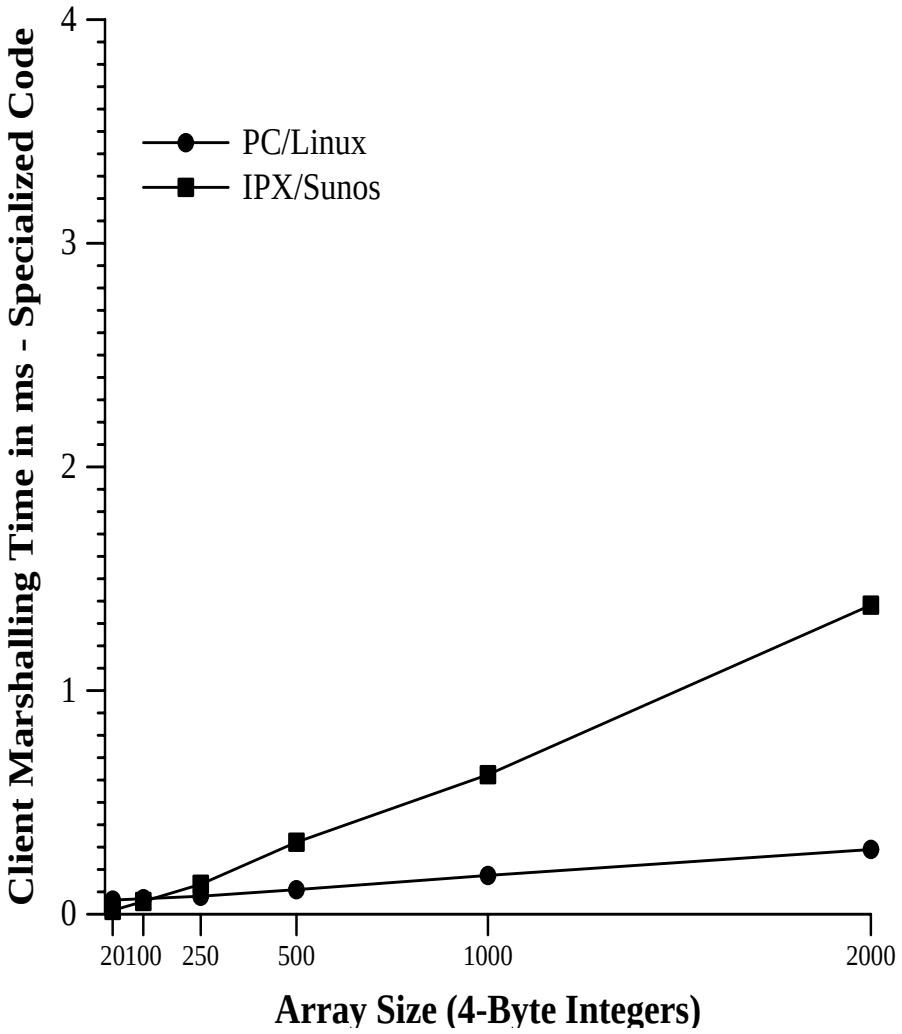




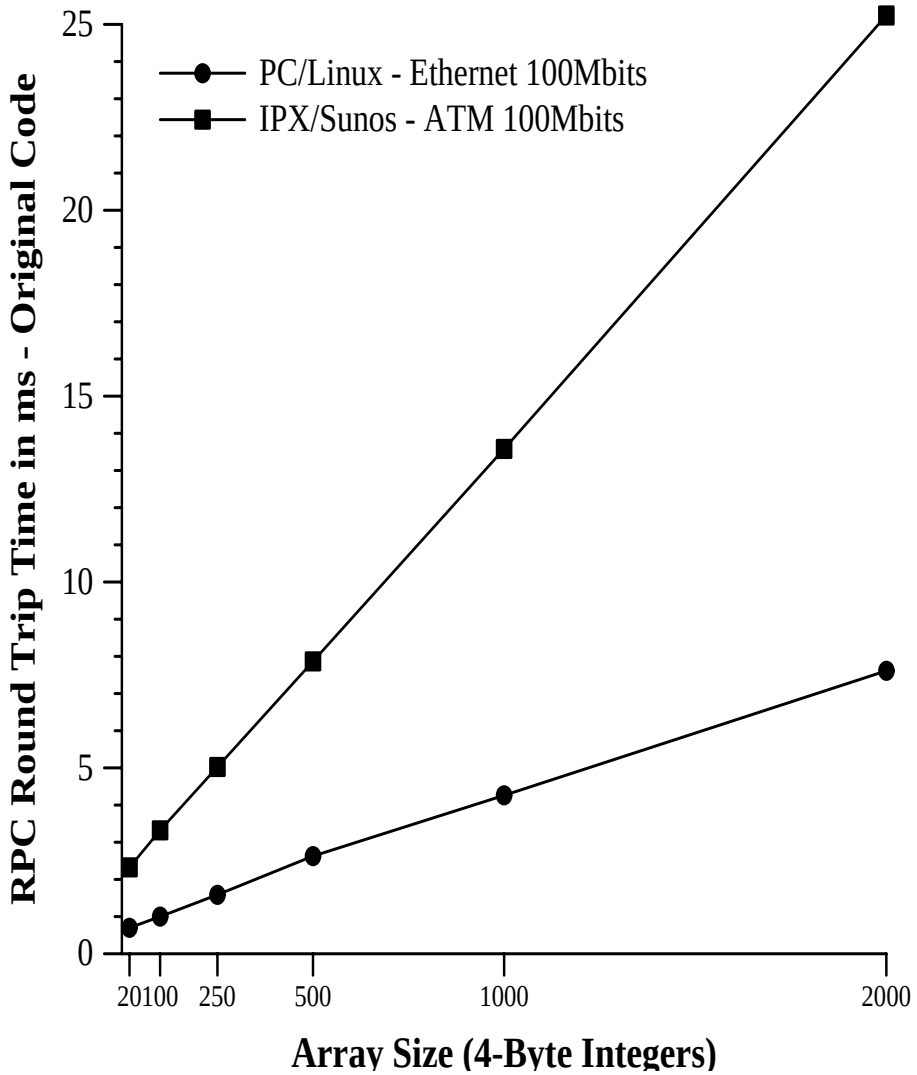
- Encoding of an integer array of size n

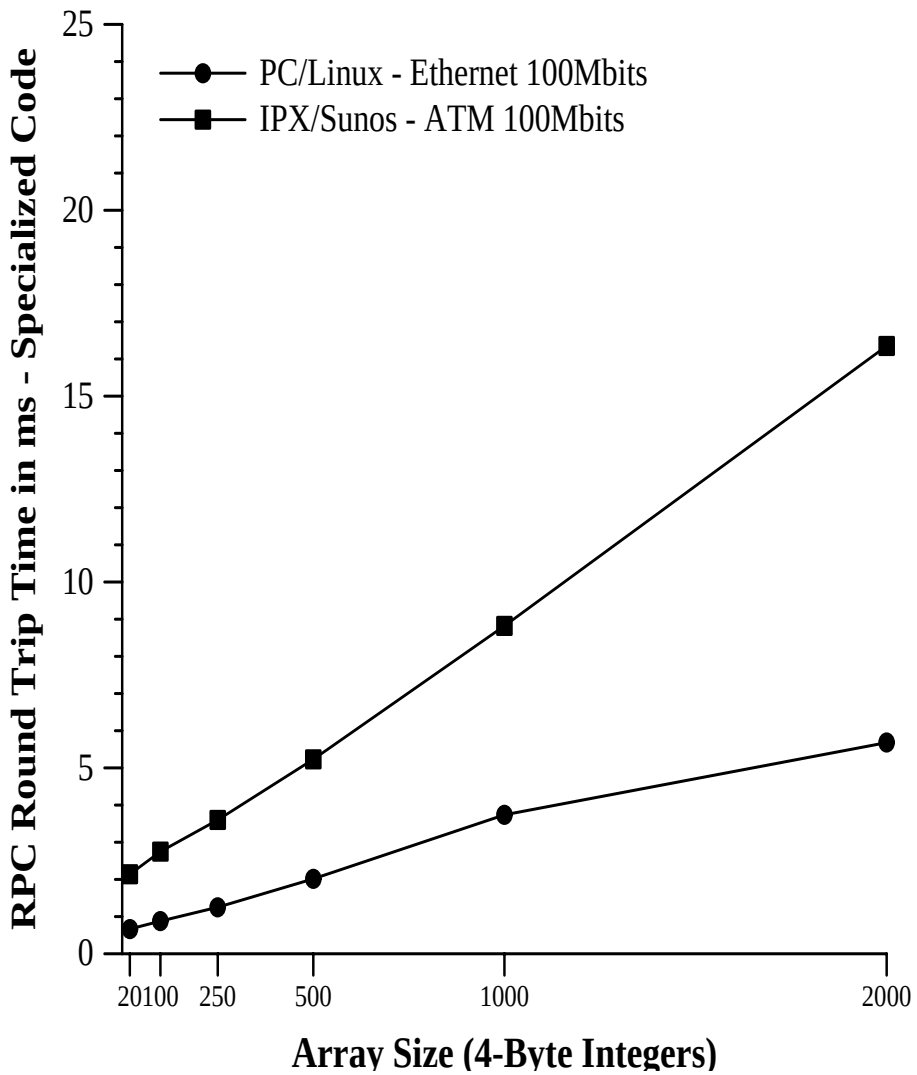
Marshaling Times





Round-Trip Times





Specialization is not Magic

The specialization process is automatic for the end user but...

- An “expert/user” has to write the specialization context
  - write once/many usages
- The “expert” may have to slightly modify the code so as to expose more specialization opportunities
  - code restructuring is always faster than implementing a dedicated optimizer/compiler
- Loop unrolling may lead to code explosion
  - on-going work in Tempo

Conclusions

- First automated specialization of an existing OS component
- Good speedups on heterogeneous platforms
- Tempo is publicly available (laptop demo on demand)

http://www.irisa.fr/compose/tempo

- On-going work: specialization of the Chorus IPC

Partial Evaluation/Specialization is a promising technique for designing and developing adaptable OS

“Think generic, don’t pay for it”