INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Uniform and Automatic Approach to Copy Elimination in System Extensions via Program Specialization

Eugen-Nicolae Volanschi, Gilles Muller, Charles Consel, Luke Hornof, Jacques Noyé & Calton Pu

N° 2903

Juin 1996

THÈME 2

Rapport de recherche

ISSN 0249-6399

A Uniform and Automatic Approach to Copy Elimination in System Extensions via Program Specialization

Eugen-Nicolae Volanschi, Gilles Muller, Charles Consel, Luke Hornof, Jacques Noyé & Calton Pu*

Thème 2 — Génie logiciel
et calcul symbolique
Projet LANDE

Rapport de recherche n° 2903 — Juin 1996 — 23 pages

Abstract: Most operating systems heavily rely on intermediate data structures for modularity or portability reasons. This paper extends program specialization to eliminate these intermediate data structures in a uniform manner. Our transformation process is fully automatic and is based on a specializer for C programs, named Tempo. The key advantage of our approach is that the degree of safety of the source program is preserved by the optimization. As a result, mature system code can be reused without requiring additional verification.

Our preliminary results on the automatically optimized RPC code are very promising in that they are identical to the results we obtained by manual specialization of the same code. In this last experiment, performance measurement of the specialized RPC fragments shows a minimal speedup of 30% compared to the non-specialized code.

Elimination of intermediate data structures is part of our research effort towards optimizing operating system components via program specialization. It improves on our previous work in that optimizations are now carried out automatically using our specialization tool. Furthermore, it shows how generic subsystems can be automatically specialized into specific system extensions by exploiting application constraints.

Key-words: Program specialization, partial evaluation, copy elimination, extensible operating systems, remote procedure call

(Résumé : tsvp)

This research is supported in part by France Telecom/SEPT, ARPA grant N00014-94-1-0845 and contract F19628-95-C-0193, NSF grant CCR-92243375, and Intel.

*Oregon Graduate Institute of Science and Technology, *e-mail:* calton@cse.ogi.edu

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

Une approche uniforme et automatique à l'élimination des copies dans les extensions systèmes par spécialisation de programmes

Résumé : La plupart des systèmes d'exploitation utilisent intensivement des structures de données intermédiaires afin de mettre en œuvre les propriétés de portabilité et modularité. Cet article propose une extension à la spécialisation de programmes pour éliminer de manière uniforme les copies intermédiaires. Notre processus de transformation est automatique et repose sur un spécialiseur de programmes C appelé Tempo. L'avantage majeur de notre approche est que le degré de sûreté du programme source est préservé par le processus d'optimisation. En conséquence, il devient possible de réutiliser du code système mature sans mettre en œuvre des vérifications additionnelles.

Nos résultats préliminaires sont très prometteurs : le produit de l'optimisation automatique de fragments de code de l'appel de procédure à distance est similaire au code spécialisé manuellement. Pour cette expérimentation, les mesures de performance sur le code spécialisé du RPC montrent un gain minimum de 30% en temps d'exécution par rapport au code non spécialisé.

L'élimination des structures de données intermédiaires s'intègre dans notre axe de recherche relatif à l'optimisation de composants de systèmes d'exploitation par spécialisation de programme. Cette étude enrichit nos travaux précédents dans la mesure u le processus d'optimisation est maintenant réalisé automatiquement par notre outil de spécialisation.

Mots-clé : Spécialisation de programme, évaluation partielle, élimination de copies, systèmes d'exploitation extensible, appel de procédure à distance

1 Introduction

Services offered by operating systems are by nature general. As new applications and hardware platforms emerge, this increasing generality penalizes performance. This conflict between generality and performance is the basis of several research projects which are aimed at designing operating systems which can adapt to usage patterns to treat specific cases efficiently [1, 2, 3, 4].

This paper explores a new approach for automatically generating system extensions in the context of extensible operating systems. Our main contribution is to provide a way of performing program specialization of general system layers exploiting the usage pattern of a particular application. This approach is suited for cases where information can be extracted from the user application in addition to information given by subsystems. Because such information varies from one application to another, the specialization process is used repeatedly; this makes its automation a necessity in the specialization of large production code such as operating systems. A typical example of this situation occurs in Remote Procedure Calls (RPC), where context information (*e.g.*, procedure signatures) is provided by an application and used by a tool (*e.g.*, stub compilers) to generate specialized code running at user level.

Our adaptation process is based on a specializer for C programs, named Tempo [5]. Tempo aggressively performs various program transformations such as elimination of unused code and variables, and procedure unfolding. Like any optimizer, if it is semantic preserving, the specializer will produce optimized code that can be trusted (if the non-specialized one is).

OS relevance. Synthetix, and the Synthesis kernel [6], have demonstrated significant potential gains of specialization in operating systems. However, both the Synthesis kernel and Synthetix experiments up to now have been done by hand. Our experience with these experiments shows that in order to scale up the experiments and apply specialization to an entire production operating system, we need powerful tools to automate the specialization process. This is particularly the case for production code with simultaneous requirements of performance, correctness, modularity, and portability.

A critical issue in the art of operating system optimization is avoiding physical memory copy. Many techniques have been introduced in the last fifteen years to optimize system structuring by eliminating copies [7, 8, 9, 10, 11]. Merging system layers together offers an opportunity to remove intermediate data structures which are normally needed for modularity reasons. However, copy elimination is beyond conventional program specialization; this optimization requires additional transformations.

Contributions. Tempo implements our general approach to automate system optimizations. The contributions of this paper can be summarized as follows.

- We have developed a new approach to combining and specializing existing system layers using program transformation.

- To achieve this integration of layers, we introduce a new technique to eliminate memory copy by constructing an abstract copy history and eliminating intermediate data structures.

Outline. The rest of the paper is organized as follows. Section 2 describes the Tempo specializer and the merging of existing system layers in a specialized system extension. Section 3 introduces the copy elimination technique and its integration into Tempo; this technique is first illustrated by automatic specialization of protocol stack layers which mimic RPC; then, we present preliminary results in applying our technique for optimizing RPC in the Chorus/ClassiX operating system. Section 4 presents related work in the specialization of operating systems. Finally, Section 5 gives concluding remarks.

2 Layer Integration via 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, re-usability, and generality. 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 protection domain crossing, data copying and format translation between layers, and heavily parameterized interface functions. Rather, modular design should 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. This automatic transformation process can collapse modules by such transformations as removing parameterization of interface functions and propagating constants. In this section we first give an overview of a program specialization system, named Tempo [5]. Then, we argue that our approach to developing Tempo makes it particularly well-suited to optimize system code. Finally, we discuss how the feature of Tempo are exploited to collapse layers.

Overview of Tempo

Tempo is a program transformation system based on program specialization [12, 13]. It takes a source program written in C and parts of its input, and produces a specialized program. This program specializer is off-line [12, 13] in that it processes a program in two steps: during the first phase only a known/unknown division of the input is given. Pointer and side-effect information of the subject program are first computed. Then, a *binding-time analysis* 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.

The result of the first phase can either be used for compile-time or run-time specialization. In the former case, a concrete value is given for each known input at compile time, and the program is specialized at compile time as well. In the latter case, the concrete values only become known at run time, and specialization relies on a strategy based on templates [14].

In both cases, the specialization process is guided by the transformations generated by the first phase.

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; they are handled in a dual way: they can be both dereferenced during specialization and residualized in the specialized program.

Tempo: a Specializer Dedicated to System Code Optimization

Besides the RPC experiment discussed in this paper, the techniques and tools we introduce are applicable to other operating system components. In fact, some of the optimizations included in Tempo have been specifically developed to address cases prompted by operating system experiments where manual specialization have been performed. These experiments are important to our work in that they introduce specialization techniques and methodologies, and assess their impact in the context of a real operating system. Specifically, Tempo has benefited from close collaboration with the Synthetix project which has manually specialized the Unix file system component of HP-UX [1]. Synthetix is actively testing and evaluating Tempo for the next generation specialization experiments on production operating system code.

Using Tempo to Collapse System Layers

Let us now describe some opportunities for specialization which appear when a given configuration of layers is to be optimized.

Fixed usage patterns of interfaces. When layers are composed in a fixed manner, some usage patterns of interface functions become available. For example, in the RPC application, the number and size of arguments are fixed by the client server interface. As a result, the size and layout of low-level messages become known. The specialization process can thus exploit this information to perform some optimizations: pre-allocation of communication resources, unrolling fragmentation and assembling loops in lower protocols.

Module instantiation. Because the combination of layers is fixed, the genericity of each layer can be eliminated. This genericity usually corresponds to options interpreted by the layer. Since some of these options are fixed for a given configuration of layers, they are identified as constants and thus propagated by the specializer. Unused functionalities then become dead code which is eliminated by the specialization process.

Unfolding of multiple entry points. Some layers provide several entry points for the same service. Each entry point offers a specific interface. For instance, a service may expect a sequence of arguments to be processed; however, several entry points corresponding to the most common cases may be available — typically, entry points for one or two arguments.

An example of this situation can be found in the Unix system calls `write` and `writev`, where the first operation is used to write a single value while the second operation writes a sequence of values.

These multiple entry points are eliminated by inlining their definitions. The resulting program directly calls the general routine with appropriately packaged arguments.

Session-oriented invariants. Subsystems often include three fundamental stages. The first stage involves the creation and initialization of a local state descriptor. The second stage consists of performing transactions which include repeated interpretation of components of the state descriptor. Usually, some components of this state descriptor do not change during these transactions. The last stage corresponds to the termination of the session and usually involves both releasing system resources and freeing the state descriptor.

Not only can the invariant components of a state be used to specialize operations which perform the transactions, but they can also be propagated to other modules to trigger further optimizations. A concrete example of this form of specialization is described in the HP-UX experiment [1].

3 Copy Elimination by Program Specialization

Achieving code fusion, as presented above, leads to performance gains. However, to obtain all the potential speedup from a complete layer integration, one must also eliminate intermediate data structures. With modern architectures, references to memory become more and more costly, relative to the speed of the processor. This is especially true when large data structures (*e.g.,* buffers) which do not fit in the first-level cache are manipulated. Traditionally, intermediate data structures are not eliminated by program specialization because this process is limited to propagating known *values*. If an intermediate data structure consists of unknown values, it will not be eliminated.

3.1 Principle of Copy Elimination

We propose an approach to integrating data structure elimination into program specialization by means of program instrumentation and specialization. The program is first instrumented to maintain a *history* of the copy of data structures. Then, the instrumented program is automatically transformed by specialization.

Instead of propagating the data, a symbolic version of the copy operation is invoked to propagate the *origin* of the data. The actual data is only copied once, from its original source to its final destination, when it is needed, that is, used in actual computations.

Figure 1-a illustrates this principle on a simple case. It shows how a copy history is used to eliminate the intermediate data structures. Notice that our strategy of copy elimination relies on some assumptions regarding the way intermediate data structures are used. In particular, the contents of the origin of the data structure must not change while it is in the copy history.

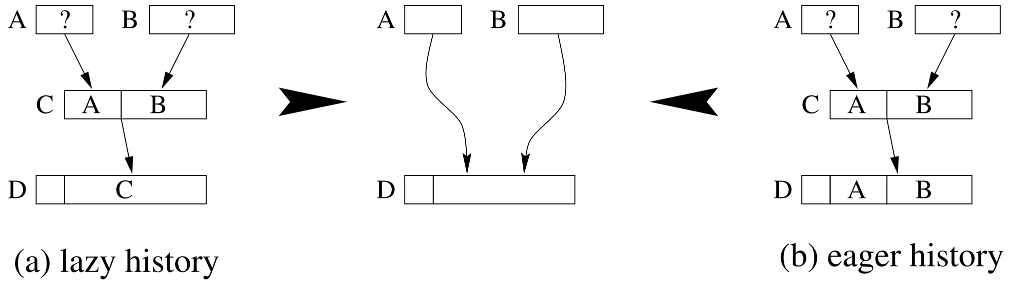

Figure 1: Keeping a history of the copies. The middle diagram represents the result of eliminating intermediate data structures using either a lazy (a) or an eager (b) history of the copies.

Modeling the copies. To introduce our model of copying, let us consider the *buffer* data type, together with two operations: an initialization function and a copy function. The initialization function is left unspecified; it is assumed that the initialization values are defined by a given application. For the sake of simplicity, it is assumed that it is a binary operation which is passed the address of a buffer and its length: `init(buffer, length)`. To allow some typical operations on buffers, like assembling and fragmentation, the copy function takes two offsets: `copy(dst, src, off_dst, off_src, length)`. No other operations allow buffers to be modified.

Furthermore, we assume that buffers are manipulated in a "single assignment" manner. That is, a buffer is first filled by initializations and/or copies. Once it is completely filled, the buffer can be used: its contents can be copied to other buffers. In other terms, copy operations into the buffer and out from it are serialized, and never intertwined. This condition ensures that the contents of the buffer will not change as long as it is part of the copy history.

In fact, various approaches can be used to construct a history of data copying: it can be built eagerly or lazily. The former approach, shown in Figure 1-a, amounts to limiting the propagation of a source buffer to one level. When the actual origin is needed, it is retrieved by following the history backwards. In contrast, eager history, shown in Figure 1-b, always points directly to the origin of the buffer. As can be seen in Figure 1-b, the history of buffer D directly points to A and B.

Notice that when all buffers are "single-assigned", both approaches to building histories are equivalent. However, this equivalence does not hold when an intermediate buffer is updated while being in the history. In this case, only the eager approach is correct and thus will be used in the rest of this paper.

In fact, the eager approach is applicable to most programs which include copying of data structures across layers, where these layers only read the data they are passed. A well-known example is protocol layers, where packets are concatenated, fragmented and dispatched, but usually not modified.

A naive implementation. A natural strategy to achieve elimination of data copying is to introduce a copy history at run time. Instead of copying data structures, only their addresses are copied. However, this approach introduces some overhead for each copy during run time, and intermediate buffers remain.

In contrast, we propose to eliminate intermediate buffers via a program transformation guided by a static analysis. This process is automatically achieved using Tempo.

3.2 Using Tempo to Eliminate Intermediate Buffers

The copy elimination process consists of two phases. Firstly, an *analysis phase* processes the program to be optimized; it is aimed at determining the history of each buffer. Secondly, if the history of a buffer can be statically determined, its intermediate instances are eliminated during the *transformation phase*. Otherwise they remain in the transformed program.

3.2.1 Analysis Phase

Before the analysis phase, the program is slightly transformed by a pass which replaces buffer operations by their symbolic version. The first step of this transformation consists of substituting standard buffers by symbolic ones: each element contains a pointer to the origin of its actual value, instead of the value itself (*e.g.*, bytes). This transformation is performed automatically. Then, the implementation of the copy operation is replaced by a symbolic one which initializes all the elements of the destination buffer with the address of the source. This symbolic operation is a library function included in our system.

Analysis phase via alias analysis. In fact, a copy history consists of pointers linking the destination to its source. One approach to collecting pointer information could be to use an alias analysis (see [15], for instance). However, existing alias analyses are not accurate enough to determine distinct pointer information about each element of an array. For instance when a buffer is built from several buffers, pointer information about the source of a specific component is lost because it is merged with the source of the other components. As a consequence, using an alias analysis would lead to overly conservative optimizations.

Analysis phase via binding-time analysis. The problem with the alias analysis is that it attempts to compute pointer information which is too accurate: not only does it determine whether pointers between objects are unconditional, but it also attempts to compute the actual pointers. In the context of copy history only the former property is needed. When the copy history of a buffer is statically known, this buffer can be eliminated. Furthermore, this property is already captured by the information produced by the binding-time analysis of Tempo. More specifically, binding-time information tells whether or not the origin of a buffer

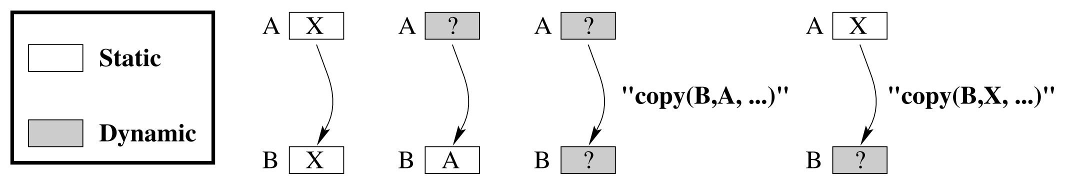

Figure 2: The different cases of buffer copying

is known. As for the actual pointers, they are directly propagated during the transformation phase. Considering again the example in which a buffer is built from several buffers, the binding-time analysis will first determine whether all copies are unconditional. If so, the actual pointers will be used during the transformation phase to eliminate the intermediate buffer. This situation is illustrated by buffer C in Figure 1-b.

3.2.2 Transformation Phase

After the binding-time analysis, buffers are divided in two types: those which have a known history — they are called *static* buffers —, and the others, called *dynamic* buffers. As a consequence, the implementation of the copy operation will differ depending on the type of the buffers which are manipulated. Some of the copies will be executed during the transformation phase, and thus will be eliminated, while other copies will remain. In fact, the copy operator could be seen as overloaded with respect to the types of its buffer arguments. Resolution of this overloading is actually done during the analysis phase. Let us consider the various cases of this overloading (see Figure 2):

Static destination/static source: the copy operation can be completely performed, and thus eliminated, during the transformation phase because the destination and the source of the buffer are known.

Static destination/dynamic source: although the origin of the source is unknown, the copy operation can still be performed. It consists of creating a new history starting from the address of the source, that is, the root of the history.

Dynamic destination/dynamic source: no information is available. The copy invocation remains in the transformed program.

Dynamic destination/static source: the destination buffer is not eliminated. As well, the copy operation to this buffer remains; its source will correspond to the root of the source buffer history.

Notice that if the source buffer itself is built from several origins, code must be generated for several copies into the destination buffer.

```
typedef char buftype;                                /* The type of buffer elements */

extern init(buftype v[], int len);                   /* initialize a buffer */

void copy(buftype dst[], buftype src[], int off_d, int off_s, int len) {
  bcopy(dst+off_d, src+off_s, len*sizeof(buftype));
}

int Dyn;                                             /* Variable unknown until execution */

buftype A[1], B[2], C[2], D[3],
  E[4], F[4], H[1], K[1];

void main() {

  init(A,1); init(B,2); init(H,1); init(K,1);

  copy(D,A,0,0,1);
  copy(C,B,0,0,2);                                   /* assemble A and B */
  copy(D,C,1,0,2);                                   /* into buffer D */

  copy(E,D,1,0,3);                                   /* E will contain D plus a header */

  if(Dyn)                                            /* add header H or K, but */
    {copy(E,H,0,0,1);}
  else                                               /* will not know which one until execution */
    {copy(E,K,0,0,1);}

  copy(F,E,0,0,4);
}
```

Figure 3: A simple program

Once the appropriate implementation is selected for each copy, the program is ready to be transformed using Tempo. This phase essentially amounts to propagating buffer addresses and executing copy operations. An example illustrating all the aspects of our approach is presented below.

3.3 A Simple Example

Consider the program shown in Figure 3, which manipulates buffers of characters. The data flow of the buffers is shown in Figure 4. Buffer D consists of data from buffers A and B. Then, the first component of buffer E is filled with either H or K; in fact, the decision as to which buffer is to be selected is under dynamic control (*i.e.,* the test of the conditional is not known until run time). Its second component contains buffer D.

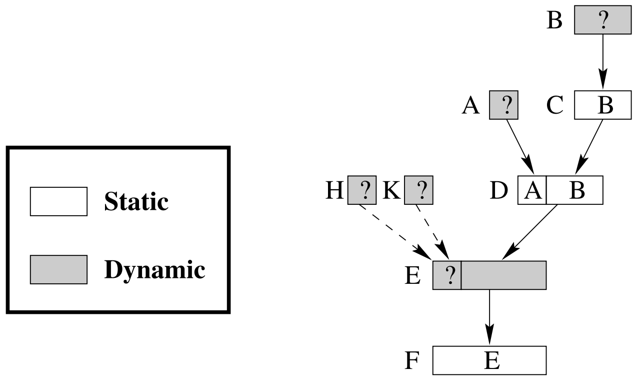

Figure 4: Data flow of the buffers for the simple program

Recall that before the analysis phase the buffers are substituted by symbolic ones. This substitution amounts to first redeclaring the data type of buffer elements (`buftype` in Figure 3) as `void *`. Then, as discussed previously, a symbolic version of `copy()` is provided.

The program yielded by the analysis phase is shown in Figure 7. The dynamic buffers (shown in boldface) are those for which the origin is unknown. This includes buffer E whose first component is unknown. Furthermore, the analysis phase has selected the appropriate implementation for the copy operator depending on the different types of buffers involved.

The four implementations of the copy operation are shown in Figure 5. It is important to notice that they are part of our approach and do not change from one application to another. Operations `copy_SS()` and `copy_SD()` correspond to calls to copy where the destination buffer is static; such calls are completely executed during the transformation. Operation `copy_DD()` is used when both destination and source buffers are dynamic; in such a context, the call is reproduced, but it refers to the standard copy routine. Operation `copy_DS()` corresponds to a call to copy where the destination buffer is dynamic and the source buffer is static; such a call yields several copy invocations, one per element. The function `NAME()` simply translates a known origin (which is the value of a pointer) to the name of its corresponding buffer — its definition is omitted. It is automatically generated from the list of buffers manipulated by the program.

Note that this version of `copy_DS()` systematically generates len copies of one byte, that is, one copy for each byte of the destination buffer. This strategy can be improved when n additional bytes come from the same buffer; in such a situation the copy does not need to be split. An optimized implementation which exploits this property is presented in Figure 6. Before generating copy operations, the maximum number of bytes coming from the same origin is accumulated and then a single copy operation is generated.

The final, optimized program is shown in Figure 8. Three static buffers out of eight have been eliminated. As well, three copies out of seven have been eliminated. In this example,

```
/* Static destination, Static source */

void copy_SS(void **dst, void **src, int off_d, int off_s, int
len)
  {
  int i;

  for(i=0; i<len; i++)
    dst[off_d+i]=src[off_s+i];
}

/* Static destination, Dynamic source */

void copy_SD(void **dst, void **src, int off_d, int off_s, int
len)
  {
  int i;

  for(i=0; i<len; i++)
    dst[off_d+i]=&src[off_s+i];
}

/* Dynamic destination, Dynamic source */

void copy_DD(void **dst, void **src, int off_d, int off_s, int
len)
{
  copy(dst, src, off_d, off_s, len);
}

/* Dynamic destination, Static source */

void copy_DS(void **dst, void **src, int off_d, int off_s, int
len)
{
  int i, source_off;
  void **source;

  for(i=0; i<len; i++) {
    NAME(&source, &source_off, src[off_s+i]);
    copy(dst, source, off_d+i, source_off, 1);
  }
}
```

Figure 5: The implementation of the copy operations

```
/* Dynamic destination, Static source */

void copy_DS(void **dst, void **src, int off_d, int off_s, int len)
{
   int i, source_off, old_i;
   void **source;

   for(i=0; i<len; i++) {
      NAME(&source, &source_off, src[off_s+i]);
      old_i=i;
      while(i<len && ((void**)src[off_s+i+1])-((void**)src[off_s+i])==1)
         i++;
      copy(dst, source, off_d+old_i, source_off, i-old_i+1);
   }
}
```

Figure 6: An improved implementation of the copy operation

```
typedef void * buftype;

int Dyn;

buftype A[1], B[2], C[2], D[3],
   E[4], F[4], H[1], K[1];

void main() {

   init(A,1); init(B,2); init(H,1); init(K,1);

   copy_SD(D,A,0,0,1);
   copy_SD(C,B,0,0,2);
   copy_SS(D,C,1,0,2);

   copy_DS(E,D,1,0,3);

   if(Dyn)
      {copy_DD(E,H,0,0,1);}
   else
      {copy_DD(E,K,0,0,1);}

   copy_SD(F,E,0,0,4);
}
```

Figure 7: The simple program after the analysis phase

```
/* TEMPO Version 1.105, 4/22/96, Copyright Irisa */

int Dyn;

void *A[1], *B[2], *E[4], *H[1], *K[1];

void main() {

  init(A, 1); init(B, 2); init(H, 1); init(K, 1);

  copy(E, A, 1, 0, 1);
  copy(E, B, 2, 0, 2);

  if (Dyn ≠ 0)
    copy(E, H, 0, 0, 1);
  else
    copy(E, K, 0, 0, 1);
}
```

Figure 8: Final version of the simple program

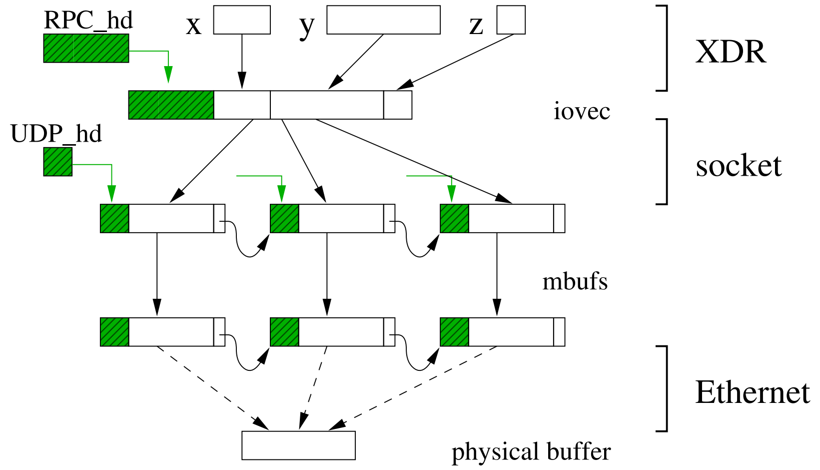

Figure 9: Data flow of the buffer in the RPC protocol stack

buffer F is optimized away, because, in this simple program, its contents is not assumed to be used. If F were used further, Tempo would have detected it and kept this buffer in the transformed program.

3.4 An RPC-like Example

The RPC protocol stack is a typical example where redundant copies and intermediate buffers can be eliminated. In the SUN stub compiler (`rpcgen`), RPC are implemented over the BSD socket abstraction, which is itself a complex stack of protocols. A sketch of the corresponding data flow is given in Figure 9. The XDR layer assembles data in a contiguous buffer. The socket layer fragments it into a chain of `mbufs`. Lower layers may copy it again. Finally, data is copied in the physical buffer. Each layer may add some headers to each packet.

We present an example which simulates three layers of the RPC implementation: the RPC layer, the XDR layer, and the socket layer (Figure 10). Compared to the last example, this one addresses more complex issues like a fragmentation loop, and dynamic memory allocation. Notice that the code given in Figure 10 has already been processed by the analysis phase.

Function `minimum()` is the client skeleton, automatically generated by `rpcgen`, from the interface definition. It takes two integer arguments and returns their minimum, computed on a remote site. The arguments are first packed in the `minargs` structure, also generated by `rpcgen`. Then, part of the RPC header (`struct rpc_msg`) is initialized. The RPC library function `clntudp_call()` implements a generic UDP client. It copies the arguments and the header into a contiguous buffer, named `iovec`. The main function of the socket layer is

```
struct rpc_msg call_msg;
struct minargs values;
bufelem iovec[RPC_SIZE];
struct mbuf heap_mbuf;

void minimum(int x, int y) {
  void clntudp_call(struct rpc_msg *cmsg, struct minargs *argsp);

  values.i1=x;
  values.i2=y;

  call_msg.rm_direction=CALL;
  call_msg.cb_rpcvers=2;
  call_msg.cb_prog=MINIMUMPROG;
  call_msg.cb_vers=MINIMUMVERS;

  clntudp_call(&call_msg, &values);
}

void clntudp_call(struct rpc_msg *cmsg, struct minargs *argsp) {
  void copy_SD(void **dst, void **src, int off_d, int off_s, int len);
  void sosend(bufelem xdrs[], int len);

  cmsg→rm_xid++;
  cmsg→cb_proc=MINIMUM;

  copy_SD(iovec, cmsg→buf, 0, 0, sizeof(struct OLD_rpc_msg));
  copy_SD(iovec, argsp→buf, sizeof(struct OLD_rpc_msg), 0,
          sizeof(struct OLD_minargs));
  sosend(iovec, sizeof(struct OLD_rpc_msg)+sizeof(struct OLD_minargs));
}


void sosend(void **xdrs, int len) {
  void copy_DS(void **dst, void **src, int off_d, int off_s, int len);
  int i=0,sz;
  struct mbuf *m, *l;

  l=0;
  while(i<len) {
    MGET(&m);
    m→next=l; l=m;
    sz=min(len-i,MBUF_SIZE);
    copy_DS(m→buf, xdrs, 0, i, sz);
    i+=MBUF_SIZE;
  }
}
```

Figure 10: RPC-like program, after analysis phase

```
/* TEMPO Version 1.105, 4/22/96, Copyright Irisa */

extern void minimum(int x, int y) {
  values.i1=x;
  values.i2=y;

  call_msg.rm_direction=CALL;
  call_msg.cb_rpcvers=2;
  call_msg.cb_prog=MINIMUMPROG;
  call_msg.cb_vers=MINIMUMVERS;
    { /* unfolded clntudp_call() */

      call_msg.rm_xid++;
      call_msg.cb_proc=MINIMUM;
      { /* unfolded sosend() */
        struct mbuf *m;

        MGET(&m);
        m→next=(struct mbuf *)0;
        { /* unfolded copy_DS() */

          copy(m→buf, call_msg.buf, 0, 0, 24);

          copy(m→buf, values.buf, 24, 0, 8);
        }
      }
    }
  }
```

Figure 11: RPC-like specialized program

called `sosend()`; it is responsible for copying the data from user space to system space. But, in the case of an optimized system extension, this protection domain crossing disappears. As a result, the remaining effect of this copy is the translation of the data in a standard system format by fragmenting it in a chain of `mbufs`.

Dealing with data structures. In this example, unlike the previous one, arbitrary data structures are involved in copy operations (*e.g.*, `struct rpc_msg`). The instrumentation before the transformation phase requires replacing these data structures by symbolic versions. This transformation is not as simple as redeclaring the type of an array, as discussed before. Our solution is to add to each such structure a new field containing the history. This field will be a symbolic buffer, that is, a vector of pointers with as many elements as the size of the original structure. All the occurrences of a transformed structure which do not involve copying are unchanged. It is only when a structure occurs in a copy invocation that its history field is used. This transformation is automatically performed.

Notice that scalar variables are not handled with this automatic transformation. Indeed, the introduction of a history component to a scalar variable requires a thorough rewrite of the source code. This is the topic of further work.

Let us now examine the transformed version of the RPC-like program (Figure 11). As can be noticed, the resulting program is highly optimized: all layers have been collapsed into one layer; buffer `iovec` has been eliminated; the fragmentation loop has been unrolled completely; only two copies, directly to the `mbuf`, are left.

One key advantage of our approach to integrating copy elimination into program specialization is that their close interaction enables synergistic optimizations to be performed. For instance, loop unrolling has made copy invocations explicit and thus enabled their elimination.

3.5 RPC Specialization in Chorus/ClassiX

We are currently applying our technique to the complete RPC in the Chorus/ClassiX operating system [16]. ClassiX is made from the combination of the Chorus V3.5 kernel and a subset of the BSD operating system [17]. Our goal is to merge together the stub, socket and UDP layers, and run the resulting optimized code in a *supervisor domain of protection* [18]. The latter mechanism provides Chorus with support for extensibility.

Manual specialization. As we develop the Tempo specializer to digest the Chorus/-ClassiX production code, we have manually specialized part of Chorus/ClassiX code by simulating the Tempo algorithms. This process is useful for two reasons: (1) we need the results of automatic specialization for the validation of Tempo, and (2) we want to estimate how much we can gain with the automatic approach. We have evaluated the sending side of RPC, as shown in Figure 12. We estimate to have the results of automated specialization using Tempo by July 1996.

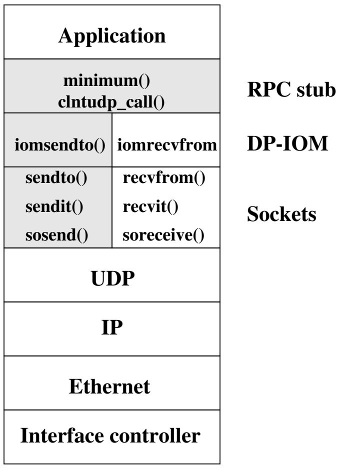

Figure 12: The protocol stack underlying the RPC implementation. The grayed part indicates the code fragment which has been manually specialized.

Preliminary results. We measured the speedup of the manually specialized version on Pentium 90 PC's running Chorus/ClassiX, connected by a 10M-bit/s Ethernet network. The timings are given in Figure 13.

As can be noticed, at the socket level, the specialized send operation is about 4 times faster than the non-specialized one (the time was measured by replacing the call to inferior layers by an immediate return). The importance of this speedup decreases to a factor of 1.3 when considering all the specialized layers (stub to socket). This is because the time spent in the stub and in the DP-IOM levels has not changed, except for $3\mu s$ which comes from the optimized packing function for the arguments.

The speedup obtained by manual specialization clearly demonstrates that our approach is promising

As of now, about 40% of the real BSD code is processed by Tempo. This corresponds to the analysis and the specialization of about a 2000-line program.

4 Related Work

The specialization techniques presented in this paper relate to many studies in various research domains such as manual system optimization and specialization, operating system structuring, and program transformation. Let us outline the salient aspects of the research in these domains.

Previous specialization experience. The techniques included in Tempo closely relate to manual optimizations which have been developed by the operating system community.

	Original	Specialized	Speedup
Socket level (emission only)	43.75	11.90	$\times 3.9$
Stub to socket (emission only)	109	74	$\times 1.3$

Figure 13: Performance comparison. Times are given in μs. Both versions execute the specialized code in a DP supervisor within the application itself.

For example, layer collapsing was first mentioned in the first Synthesis paper [6]. The elimination of intermediate data structures is useful in much of operating system data handling, including RPC, network protocols, file systems, and other forms of I/O. Granted, there have been proposals of specialized implementation techniques to eliminate copying for particular situations, for example, **`fbufs`** for networking protocol stack [10]. These hand-specialized techniques, however, usually entails compromises in other system properties such as maintainability and portability. Furthermore, these techniques typically cannot be generalized to other operating systems components. Consequently, automated elimination of intermediate copying can be valuable in operating system design and implementation, by simplifying the written code while preserving run-time performance.

General-purpose optimizations. More generally, it is well recognized that physical memory copy is one of the important causes of overhead in computer systems. Finding solutions to avoid or optimize copies is a constant concern of operating system designers. For instance, copy-on-write [7] was the technique which made message passing efficient enough to allow operating systems to be designed based on a micro-kernel architecture [8, 19]. 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 implementation [9]. 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 [11]. Mosberger *et al.* describe techniques for improving protocols by reducing the number of cycles stalled to wait for memory access completion [20].

Safety issues in 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. While the previous systems did not include a safety verification mechanism, 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 [4] 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 [21] which rewrites the binary code to insert software-based memory protection instructions.

Program transformation. It has long been known that program transformation is a key technology to reconcile generality and efficiency. It has been used successfully for specializing programs in domains such as computer graphics [22]. 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. Our specializer, Tempo, relies on partial evaluation [12, 13], a form of program transformation which is now reaching a level of maturity which makes it possible to develop specializers for real-sized languages like C [5, 23] and apply these specializers to real-sized problems.

5 Conclusion

In this paper we have described a uniform and automatic approach to (1) merging existing system layers in a specialized extension and (2) eliminating static intermediate data structures. Our approach relies on the use of a program specializer named Tempo, which preserves the semantics of the program. As a result, it allows existing system layers to be re-used and optimized safely without modification.

Our techniques and tools have been successfully applied to a reduced RPC implementation which was derived from the BSD implementation. We are currently applying our technique to the complete RPC in the Chorus/ClassiX operating system [16]. As of now, about 40% of the real BSD code is processed by Tempo. This corresponds to the analysis and the specialization of about a 2000-line program.

More generally, Tempo offers a unique opportunity to explore various forms of optimizations in real-sized systems. This transformation system can be particularly crucial for the optimization of operating system components which provide applications with generic services. For example, in addition to the application on RPC outlined above, the Synthetix team at OGI is designing specialization experiments using these techniques and the Tempo specializer.

References

[1] 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 [24], pages 314–324.

[2] 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.

[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 [24], pages 267–283.

[4] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [24], pages 251–266.

[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 Workshop, Dagstuhl Castle*, Lecture Notes in Computer Science, February 1996. To Appear.

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

[7] 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.

[8] 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.

[9] 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.

[10] P. Druschel and L.L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In SOSP93 [25], pages 189–202.

[11] C. Maeda and B.N. Bershad. Protocol service decomposition for high-performance networking. In SOSP93 [25], pages 244–255.

[12] 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.

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

[14] 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.

[15] 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.

[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.

[17] Chorus/ClassiX product description. Technical Report CS/TR-94-44.3, Chorus Systemes, 1994.

[18] 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.

[19] 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.

[20] D. Mosberger, L.L. Peterson, P.G. Bridges, and S.W. O'Malley. Analysis of techniques to improve protocol processing latency. Technical Report 96-03, Department of Computer Science, The University of Arizona, 1996.

[21] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. In SOSP93 [25], pages 203–216.

[22] 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.

[23] 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, Hew Haven, CT, USA. Technical Report YALEU/DCS/RR-909.

[24] *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.

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

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, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
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)
ISSN 0249-6399