

A Tour of Tempo: A Program Specializer for the C Language

Charles Consel^a Julia L. Lawall^b Anne-Françoise Le Meur^a

^a *Compose group, INRIA/LaBRI,
ENSEIRB, 1 avenue du docteur Albert Schweitzer,
33402 Talence Cedex, France
{consel,lemeur}@labri.fr*

^b *Dept. of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen, Denmark
julia@diku.dk*

Abstract

Tempo is a specializer for the C language that automatically customizes a program with respect to the values of configuration parameters. It offers specialization at both compile time and run time, and both program and data specialization. To control the specialization process, Tempo provides the program developer with a declarative language to describe specialization opportunities for a given program.

The functionalities and features of Tempo have been driven by the needs of practical applications. Tempo has been successfully applied to a variety of real-sized programs in areas such as operating systems and networking.

1 Introduction

Program specialization is an automatic technique that customizes a program with respect to the values of configuration parameters. Computations that depend on these early values are performed and the corresponding results are encoded in a specialized program. Extensive research and experiments have made program specialization a well-established approach to reconciling genericity and performance [2,3]. Program specialization has demonstrated its effectiveness in a variety of areas such as operating systems, networking, graphics, scientific computing and compiler generation [3–5].

Program specialization is typically performed in two stages: *preprocessing* and *processing*. The preprocessing phase consists of a form of dependency analysis, called a *binding-time analysis*, that takes a program and a description of the

configuration parameters available at specialization time. Given these inputs, the binding-time analysis determines the computations that solely depend on the configuration parameters; these computations are said to be *static*. The computations that may depend on non-configuration parameters are said to be *dynamic*. Binding-time information is used by the processing phase of program specialization to customize the program with respect to actual configuration values.

For a number of years following Jones' seminal paper on program specialization [6], researchers actively explored this technique in a number of directions without challenging its compile time and source-to-source nature. More than 10 years later, an approach to specializing programs at *run time* was proposed by Consel and Noël [7]. This new capability opened a number of new opportunities for specialization in applications where the values of configuration parameters are only available at run time. Another extension to program specialization is to specialize a program in *multiple stages*, *i.e.*, incrementally [8,9]. It was shown that, by factorizing the transformation process, specializing a program at each stage costs considerably less than specializing it once all the data are available. Multi-stage specialization has been proposed both at compile time and run time.

The dual notion to specializing programs is specializing data; *data specialization* encodes the results of early computations in data structures [10], instead of encoding them in a residual program. Program and data specialization are complementary: program specialization can be used if the number of results to encode in the specialized program does not cause a code blowup; otherwise, data specialization can compactly represent results in data structures. Chirokoff *et al.* show that both strategies can be integrated in a single specialization process; they investigate the benefits and limitations of combining program and data specialization [11]

Initially, Jones' main target application for program specialization was compiler generation from executable language specifications written in a denotational style [6]. As a result, research efforts then primarily focused on functional languages. The first program specializer, called Mix, processed untyped, first-order, side-effect free functional programs [6]. Many program specializers succeeded Mix and proposed new analyses and transformations to cope with higher-order functions and data structures [12,13]. A variety of other languages were explored ranging from logic to imperative languages [4,5]. As program specialization became more mature, research targeted real-sized languages such as Fortran [14], C [4,15] and Java [16].

This paper gives a tour of a specializer for the C language, named Tempo. A key feature of Tempo is that it offers the main forms of specialization: it specializes programs both at compile time and run time; run-time specialization can be performed incrementally; Tempo offers both program and data specialization; finally, it has been used as a back-end to specialize programs written in other source languages, namely, Java and C++. Interestingly, all these forms of specialization share the same preprocessing phase.

Unlike previous program specializers, the design of Tempo was targeted towards the needs of real-sized applications in systems and networking rather than compiler generation. To cope with this kind of programs, its design and implementation were iterated until it successfully optimized a set of representative programs. This process led to the introduction of new analysis features needed to achieve an expected degree of specialization [17–19]. The resulting program specializer significantly improved performance of industrial-strength code such as the remote procedure call developed by Sun [20].

To cope with both a realistic language such as C and real-sized applications, existing program specializers require the programmer to be proficient in using a number of complex parameters and mechanisms to finely configure the specialization process. Furthermore, even when properly configured, a program specializer may not produce a residual program that matches the programmer's expectations. Unsuccessful specialization may be caused by a program that is inappropriately structured and/or by a lack of accuracy in the analyses of the program specializer. To address these issues, Tempo provides a declarative language that enables the programmer to specify specialization scenarios for a given program [21]. Such scenarios are used to configure the specialization process and are checked during preprocessing to ensure that specialization matches the programmer's expectations.

Working Example

We illustrate our tour of Tempo by the Sun Remote Procedure Call (RPC) mechanism. The RPC is used to support the implementation of distributed services between heterogeneous machines [22]. There have been many efforts to manually optimize critical parts of the RPC, to reduce, or even to eliminate, the overhead of this mechanism whenever possible.

In this paper, we focus on the marshaling layer, called the XDR. This layer converts values into a machine independent format. Arguments of a remote procedure are marshaled by a client, before being sent, and unmarshaled by the

<code>xdr_int(&arg)</code>	<code>// Machine dependent switch on integer size</code>
<code>xdr_long(intp)</code>	<code>// Generic marshaling or unmarshaling</code>
<code>XDR_PUTLONG(lp)</code>	<code>// Generic marshaling to memory, stream...</code>
<code>xdrmem_putlong(lp)</code>	<code>// Write in output buffer and check overflow</code>
<code>htonl(*lp)</code>	<code>// Choice between big and little endian</code>

Fig. 1. Abstract trace of the marshaling of an integer value

server, when received. The procedure call result is similarly (un-)marshaled.

The Sun RPC implementation [22] is well-suited to illustrate program specialization because it consists of a set of highly generic micro-layers. Each layer is devoted to a small task, for example, marshaling some parameter and writing the result in memory. Roughly, a generic function implements each layer; it interprets its arguments to determine the function to invoke in the layer below.

Let us illustrate the layered architecture of the marshaling process by considering the treatment of an integer value as an argument to a remote procedure. The call chain caused by this marshaling is summarized in Figure 1, where the call argument is stored in the variable `arg`. We assume that the configuration parameters of the marshaling process (*e.g.*, transport protocol and coding direction) have been set higher in the call chain. These parameters are kept in some sort of a *marshaling state* that is omitted from Figure 1.

Initially, the function `xdr_int` is invoked with the address of the integer value to be marshaled. Because such a value is represented as a long integer on the host machine, this function invokes `xdr_long` to further process it. The function `xdr_long` examines the marshaling state to determine the coding direction and calls the function `XDR_PUTLONG` to perform the marshaling. The marshaling state is further examined to determine whether the integer value should be stored in memory or in a stream. The former option is taken and so the function `xdrmem_putlong` is invoked. This function writes the value in a buffer after using the function `htonl` to change the order of the bytes, if needed.

To further explore the marshaling process, let us examine the function `xdrmem_putlong`, displayed in Figure 2. This function takes the marshaling state, as defined by the type `XDR`, and a pointer to the data to be marshaled. As shown by this function, the marshaling state includes such information as the space left in the buffer (the field `x_handy`) and a pointer into the buffer (the field `x_private`). If there is enough space left, the function `htonl` is invoked to produce an integer value with an appropriate byte ordering.

```

bool_t xdrmem_putlong(XDR * xdrs, long * lp) {           // xdrs points to the marshaling
                                                         // state and lp points to data
    if ((xdrs->x_handy -= sizeof(long)) < 0)           // Decrement space left in buffer
        return (FALSE);                               // Return failure on overflow
    *(long *)xdrs->x_private = (long)htonl((u_long)(*lp)); // Copy to buffer
    xdrs->x_private += sizeof(long);                   // Point to next copy location
    return (TRUE);                                    // Return success
}

```

Fig. 2. Source code of the `xdrmem_putlong` function

Overview

The rest of this paper is organized as follows. Section 2 presents the various ways one interacts with Tempo; it includes specialization declarations, verification of these declarations, and tools to visualize analysis results. Section 3 describes the anatomy of Tempo, examining the analyses performed in the preprocessing phase, and the various alternative processing phases that can follow. Section 4 discusses the use of Tempo as a back-end specializer in the context of Java and C++. Section 5 lists some applications successfully optimized by Tempo and presents some performance measurements. Finally, Section 6 gives concluding remarks and proposes future directions.

2 Interacting with Tempo

Figure 3 illustrates the process of interacting with Tempo. The program developer provides the source code of the program to specialize and specialization declarations to the preprocessing phase. Based on these inputs, the preprocessing phase determines how the program should be specialized and yields specializable code as a result. Verifications are performed during this phase to guarantee that the specializable code satisfies the developer’s declarations. To further assist the developer, Tempo provides some visualization tools, allowing inspection of intermediate results. Once the developer has prepared the specializable code, he makes it available to users, who provide appropriate specialization values to the processing phase to generate the specialized code.

2.1 Declaring Specialization

Successful specialization typically requires that the program developer have an intuitive understanding of how static information should propagate through the program. This is, however, often not sufficient to obtain the desired degree of specialization. Specializers for realistic languages typically offer some

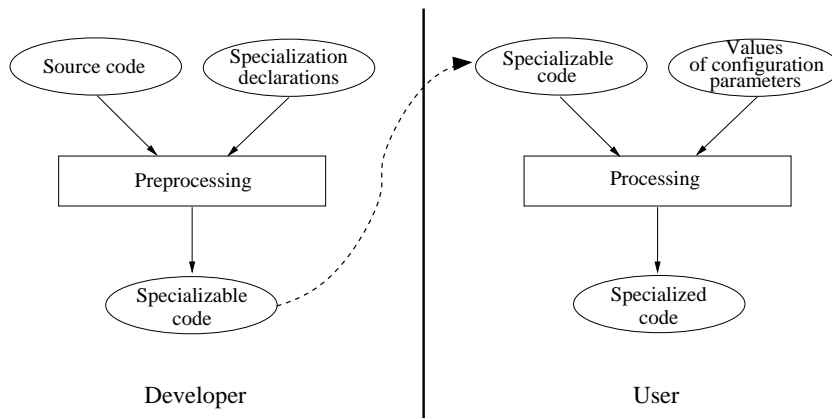


Fig. 3. Decomposition of the specialization process

configuration options; choosing the appropriate options for a given program requires some expertise. Furthermore, since a program specializer is an automatic tool, it may by default choose to specialize the code in a way that contradicts the developer’s intentions; consequently, the developer needs a way to describe his intentions to the tool in order to be able to control the specialization process. Nevertheless, specializers typically do not offer the proper abstractions to address either of these issues. As a result, residual programs are often under-specialized or over-specialized.

Tempo provides a declarative language for describing specialization opportunities [21,23,24]. These declarations are written by the developer in auxiliary specialization modules as the source code is being developed. The developer describes specialization opportunities as *specialization scenarios*. A specialization scenario is a declaration that identifies a function, global variable or data structure that is of interest for specialization, and declares the binding-time context, *i.e.* static or dynamic, in which specialization involving this construct should be carried out. A specialization scenario is analogous to a type declaration, where types express the developer’s intention, not about the values being manipulated, but about the moment of their availability.

We now consider the specialization opportunities provided by the function `xdrmem_putlong`. When the amount of data to marshal is known, the buffer overflow check and pointer increment performed by this function can be carried out at specialization time, leaving only the copying of the data into the buffer in the specialized code. Figure 4 shows specialization scenarios that describe these opportunities. To declare the binding-time properties of the value referenced by `xdrs`, we annotate the declaration of its type `XDR` with binding-time information. Specifically, the scenario `BtXDR`, defined in the module `xdr`, specifies that the `x_handy` and `x_private` fields are static. We then use this scenario to declare a specialization scenario for `xdrmem_putlong`, in the module `xdrmem`. This scenario specifies that the function `xdrmem_putlong`, defined in `xdr_mem.c`, can be specialized if the pointer `xdrs` is static, the result of

```

Module xdr {
  Imports { ... }
  Defines {
    From xdr.h {
      BtXDR :: struct XDR { ...
        D(char) S(*) x_private;
        S(int) x_handy;
        ...};
    }
    From xdr.c {
      Btxdr_int ...
      Btxdr_long ...
      ...
    }
  }
  Exports { BtXDR; Btxdr_int; Btxdr_long; ...}}

-----

Module xdrmem {
  Imports { From xdr.mdl { BtXDR; }
          From library.mdl { Bhtonl; }
  }
  Defines {
    From xdr_mem.c {
      Btxdrmem_putlong :: intern xdrmem_putlong (BtXDR(struct XDR) S(*) xdrs, D(long *) lp)
        { needs { Bhtonl; } };
    }
    ...
  }
  Exports { Btxdrmem_putlong; ... }}

```

Fig. 4. Specialization modules for the `xdr` and `xdrmem` layers

dereferencing this pointer satisfies the scenario `BtXDR`, and `lp` is completely dynamic. Furthermore, the `needs` clause specifies that invocation of `htonl` within the body of the `xdrmem_putlong` should satisfy the imported scenario `Bhtonl`.

In our example, the module definitions are organized according to the layer decomposition of the XDR library. Collecting scenarios into specialization modules centralizes specialization information related to a particular functionality and facilitates the understanding and reuse of specialization scenarios.

The developer compiles the specialization modules to produce declarations to configure Tempo. These declarations guide the binding-time analysis of Tempo according to the developer’s intentions.

2.2 Verifying Specialization

The binding-time analysis verifies that the computed properties are coherent with the binding-time properties declared by the developer. These checks are analogous to the checks that a compiler performs during type checking. Concretely, the verifications impact the preprocessing phase as follows. If a parameter declared static is found to be dynamic by the analysis, the preprocessing phase stops with an error message. If a parameter declared dynamic is

```

char * buf;
struct XDR *xdrs;
...
buf = (char *) malloc(sizeof(int));           // XDR marshaling state initialization
xdrs = (XDR*) malloc(sizeof(struct XDR));
xdrs->x_private = buf;
// overflow check removed                    // Integer marshaling
(int *)xdrs->x_private = htonl(number);
// pointer increment removed
...
free(buf);
free(xdrs);

```

Fig. 5. Specialized code for integer marshaling

found to be static by the analysis, its binding-time is forced to be considered dynamic by the analysis; in this manner the declarations guide the analysis. These checks avoid over-specialization and identify binding-time incoherence in cases of under-specialization.

Verifications performed during the binding-time analysis are critical to enable the developer to obtain specializable code that matches his expectations. Furthermore, as the declaration language allows the developer to decompose a complex specialization scenario into the combination of simpler ones (using the `needs` keyword), it is possible to incrementally develop code and scenarios that yield proper specialization. Decomposing the problem in this manner facilitates the specialization of large programs.

2.3 Visualizing Specialization

Tempo provides some support tools to assist the developer in making code specializable. Prior to preprocessing, graphic tools allow the developer to visualize his specialization declarations. One tool shows the hierarchy of the modules involved in a given specialization and another tool allows inspection of the scenario dependency graph, enabling visual checking of the propagation of static parameters throughout the code. After the preprocessing phase, the developer may check the intermediate results of the different analyses by inspecting the annotated files that are generated during this phase.

Once generated, the specializable code may be distributed to users who provide the values of the configuration parameters to customize the code to their needs. Figure 5 shows the specialized code performing an integer marshaling, which has been inlined in its calling context. Following the specialization scenario `Btxdrmem_putlong`, the overflow check and the pointer increment have been removed during specialization, leaving only the buffer copy.

3 A Tour of the Tempo Specialization Engine

The design of Tempo has been guided by the desire for a uniform approach to the different forms of specialization: compile-time and run-time program specialization, and data specialization. All three forms of specialization use the same analysis engine. We first present the analyses performed during the preprocessing phase and then give an overview of the different forms of specialization offered by the processing phase.¹

3.1 Preprocessing

The purpose of the preprocessing phase is to identify the constructs of the source program that can be simplified during specialization and those that must be reconstructed to form the specialized program (*i.e.*, *residualized*). Figure 6 shows the sequence of analyses and transformations that are performed during this phase. We focus on the last three analyses, *binding-time analysis*, *evaluation-time analysis*, and *action analysis*, which determine the structure of the specialized program. The result of these analyses, which forms the specializable code, is an annotated program that is ready for specialization.

3.1.1 Binding-time analysis

Binding-time analysis identifies expressions that depend only on the values of configuration parameters and on other static information (*e.g.*, constants) available in the program. We present only an overview of the analysis. Hornof and Noyé present the binding-time analysis of Tempo in more detail [19].

Tempo uses the binding times *static* and *dynamic*. The least-upper bound of static and dynamic is dynamic. An expression is dynamic if it is a variable that has been determined to be dynamic or if it has a dynamic subexpression. For example, if *s* is static and *d* is dynamic, then $d + (2 * s)$ is dynamic. An expression is static otherwise. In the previous example, the subexpression $2 * s$ is static.

A key point in the binding-time analysis of an imperative language is the flow of binding-time properties across assignments. When the left-hand side of an assignment is a simple variable, the binding time of the variable becomes the binding time of the right-hand side expression. For example, in the code

¹ Tempo uses a subset of C as an internal representation. We have translated all of the figures in this section from this internal representation to the form of the original source program, for readability.

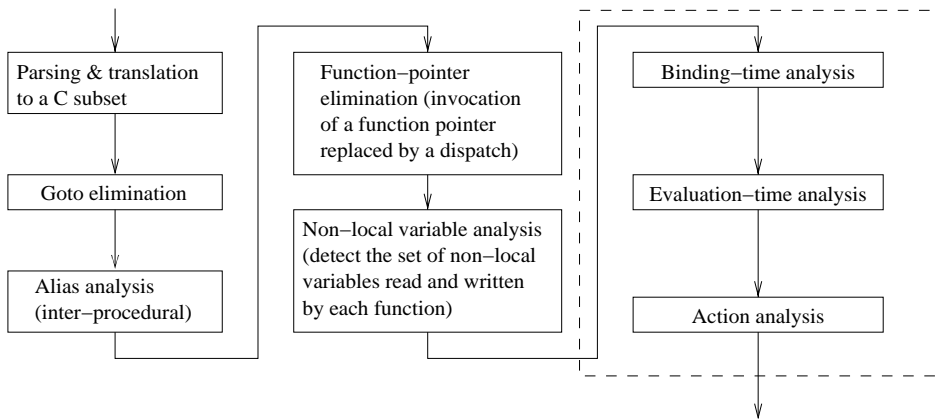


Fig. 6. Components of the preprocessing phase

following an assignment $x = 3$, the variable x is static. When the left-hand side is a dereference expression, as in $*x = 3$, the effect of the assignment depends on the alias information. If the dereferenced expression has only one possible alias, then this alias is again given the binding-time of the right-hand side. If the dereferenced expression has more than one possible alias, then only one of them is affected by the assignment at specialization time. In this case, the binding time of each alias is set to the least-upper bound of its current binding time and the binding-time of the right-hand side expression.

Binding times are affected both by data-flow dependencies, as in the examples above, and by control-flow dependencies, in the treatment of conditionals and loops (all gotos are encoded as conditionals and loops in the goto elimination phase shown in Figure 6, following the algorithm of Erosa and Hendren [25]). When a conditional has a static test, only one of the branches is specialized and the values of any static variables assigned in the chosen branch are available to specialization of the rest of the code. Nevertheless, because the binding-time analysis cannot determine which branch is chosen during specialization, the binding-time of each variable assigned within either branch (as determined by the alias analysis and the non-local variable analysis shown in Figure 6) is set to the least-upper bound of its binding times at the end of either branch. For example, in Figure 7a, x is assigned a static value in both branches and is thus considered static after the conditional, while in Figure 7b, x is assigned a static value in one branch and a dynamic value in the other and thus is considered dynamic after the conditional. When a conditional has a dynamic test, the specialization phase specializes both branches. In this case, a static variable modified in at least one of the branches potentially has two specialization-time values at the end of the conditional. Thus, all variables assigned in either branch are subsequently considered dynamic. For example, if the test expression in Figure 7 is dynamic, then the final reference to x is dynamic in both examples. Loops are treated by standard fixpoint techniques [4].

<pre> // test is static, d is dynamic if (test) x = 3; else x = 4; ... x ... /* x is static */ </pre> <p style="text-align: center;">(a)</p>	<pre> if (test) x = 3; else x = d; ... x ... /* x is dynamic */ </pre> <p style="text-align: center;">(b)</p>
--	---

Fig. 7. Binding-time analysis in the presence of conditionals

In a realistic application, only a portion of the program may present interesting specialization opportunities. When specialization is applied to a fragment of the original program, this fragment may call functions that affect global variables that are subsequently referenced by the specialized fragment. To describe such side effects, Tempo allows the program developer to create an *analysis context file* that describes the side effects of external functions. Currently, this file contains C definitions of these functions, which may abstract the original definition to the point that they only achieve the relevant binding-time effect, *e.g.* assigning dummy static or dynamic values to the affected variables rather than using the original right-hand side expressions. We are working on a more convenient notation for this information, as part of the language for declaring specialization opportunities, described in Section 2.

To provide the degree of precision needed for realistic applications, the binding-time analysis of Tempo adopts several standard dataflow analysis strategies: *flow sensitive*, *context sensitive*, and *polyvariant* treatment of data structures [19]. Flow sensitivity affects the treatment of assignments. Rather than maintaining a single binding time for all uses of a given variable, the binding time of a variable in Tempo depends on the most recent assignment of the variable or other binding-time effect (*i.e.*, control dependence). For example, in Figure 7b, *x* is considered static in the “then” branch of the conditional, even though it is considered dynamic elsewhere in the program. Context sensitivity affects the treatment of function calls. Tempo creates a variant of a function for each binding-time configuration that occurs at any of the possible call sites, taking into account both the binding times of the function’s parameters and the binding times of any variables referenced by the function, as determined by the non-local variable analysis. Polyvariance of data structures, which is optional in Tempo, affects the treatment of structure fields. When polyvariance is used, each declared data structure is associated with a separate binding-time description. Otherwise, all instances of each structure type share the same description. In either case, the description contains a separate binding-time for each structure field. Polyvariant analysis is more precise, but is also more expensive and gives no extra benefit for some programs.

Including these analysis features in the binding-time analysis of Tempo increases the availability of static information within a program at specialization time. Nevertheless, we found that these features were not sufficient to achieve the desired degree of specialization from many systems programs, such as the

```

bool_t xdrmem_putlong(XDR * xdrs, long * lp) {           (1)
    if ((xdrs->x_handy -= sizeof(long)) < 0)             (2)
        return (FALSE);                                 (3)
    *(long *)xdrs->x_private = (long)htonl((u_long)(*lp)); (4)
    xdrs->x_private += sizeof(long);                     (5)
    return (TRUE);                                     (6)
}                                                       (7)

```

Fig. 8. Result of binding-time analysis of `xdrmem_putlong`

XDR [26]. In systems code, it is common to return a status flag to indicate the success or failure of a computation, and the value of this flag often depends on the values of only a subset of the parameters. To address this issue, we developed the analysis feature *return sensitivity* for Tempo. When a function has a static return value, but contains dynamic computations, the return value is considered static at the call site, even though the function call must be residualized.

Finally, we consider binding-time analysis of the `xdrmem_putlong` function shown in Figure 2, where the parameter `xdrs` is static, the parameter `lp` is dynamic, and the fields `x_handy` and `x_private` in the structure referenced by `xdrs` are static. The result of binding-time analysis is shown in Figure 8. Dynamic constructs are overlined. Only the reference to `lp` and the result of passing this value to `htonl` are considered dynamic (line (4)); the remaining computations are considered to be static. In particular, the left-hand side of this assignment is not considered dynamic, because the affected address is known at specialization time. Furthermore, the subsequent use of the `x_private` field is static (line (5)), even though the assignment sets the location referenced by this field to the value of a dynamic expression. Thus, the precision of the binding-time analysis of Tempo is sufficient to support a static pointer to a dynamic value, which allows specialization-time computations to be performed on this pointer value, as shown here. Both values returned by this function are static (lines (3) and (6)). Return sensitivity allows specialization of the call site to use whichever return value is chosen during specialization.

3.1.2 Evaluation-time analysis

The result of binding-time analysis indicates the expressions that depend only on the static information and thus can be evaluated during specialization. This information is not, however, sufficient to ensure correct specialization. Problems arise when a static value is to be encoded in the residual program and when a variable assigned to a static value is subsequently considered to be dynamic.

When a static expression occurs in a dynamic context, the value of this expression must be residualized as part of the specialized code. Nevertheless, some

```

bool_t xdrmem_putlong(XDR * xdrs, long * lp) {           (1)
    if ((xdrs->x_handy -= sizeof(long)) < 0)           (2)
        return (FALSE);                               (3)
    *(long *)xdrs->x_private = (long)htonl((u_long)(*lp)); (4)
    xdrs->x_private += sizeof(long);                   (5)
    return (TRUE);                                    (6)
}                                                       (7)

```

Fig. 9. Result of evaluation-time analysis of `xdrmem_putlong` (underlined terms are static and dynamic, overlined terms are dynamic)

values are *non-liftable*, *i.e.* they cannot be meaningfully represented in the specialized code. A pointer to a local variable is always non-liftable. When specialization is carried out at compile time, pointers to global variables are also non-liftable. Floating-point numbers may be considered non-liftable, because of the difference between the precision of the textual and internal representations. In the rest of this section, we consider all pointers to be non-liftable.

The flow-sensitive binding-time analysis of Tempo implies that a variable can be considered to be static at some occurrences and dynamic at others. For example, a variable that is assigned a static value in a conditional with a dynamic test is considered dynamic at any reference after the conditional, even though there is no intervening dynamic (*i.e.* residualized) assignment. A similar situation arises when a global variable that is assigned a static value is referenced by the application that will use the specialized code. When such a static assignment reaches a dynamic reference, the static assignment must be treated as both static and dynamic to ensure that the dynamic reference is meaningful in the residual program.

Evaluation-time analysis addresses these issues [17]. The analysis reannotates every non-liftable static expression that occurs in a dynamic context as dynamic. This reannotation may in turn cause subexpressions to occur in a dynamic context, and thus provoke a sequence of such reannotations. The analysis also maintains a set of the variables for which there is a subsequent dynamic reference but no reaching dynamic initialization along the current control-flow path. Any static assignment reaching such a reference is reclassified as both static and dynamic.

Figure 9 shows the result of evaluation-time analysis of `xdrmem_putlong`. The expression `*(long *)xdrs->x_private` on the left-hand side of the assignment in line (4) is annotated as static in the result of the binding-time analysis (Figure 8) but occurs in a dynamic context, because the right-hand side of the assignment is dynamic. The value of the left-hand side of an assignment is always an address, and thus a non-liftable value. The evaluation-time analysis reannotates this expression as dynamic. Because the expression is dynamic, the subexpression `xdrs`, which is also a pointer, must be considered dynamic as well, as shown in Figure 9. Following this adjustment, there is a dynamic

```

bool_t xdrmem_putlong((XDR * xdrs)EV&RES, (long * lp)ID) {REB
    (if ((xdrs->x_handy -= sizeof(long)) < 0)
        return (FALSE);)EV
    (*(long *)xdrs->x_private = (long)htonl((u_long)(*lp)));ID
    (xdrs->x_private += sizeof(long);)EV
    (return (TRUE);)EV
}REB

```

Fig. 10. Result of action analysis of `xdrmem_putlong`

occurrence of `xdrs`, but no corresponding dynamic initialization. The only reaching initialization is the first parameter of `xdr_putlong`. This parameter becomes static and dynamic, implying that the corresponding argument becomes static and dynamic at each call site. Finally, if multiple values are to be marshaled, the pointer `xdrs->private` may be needed after returning from `xdrmem_putlong`. In this case, the incrementing of this pointer in line (5) would also be reannotated as static and dynamic.

3.1.3 Action analysis

The final analysis of the Tempo preprocessing phase is action analysis. This analysis determines how each construct should be specialized during the processing phase, based on the evaluation times of all of its subterms. Action analysis is not essential, but simplifies the construction of a dedicated specializer to perform compile-time or run-time program specialization, or data specialization.

Tempo uses five actions: evaluate (EV), reduce (RED), rebuild (REB), identity (ID), and evaluate/residualize (EV&RES). Most of these actions are illustrated by the result of action analysis of the `xdrmem_putlong` function (Figure 10). The parameter `xdrs` that is annotated as static and dynamic in Figure 9 is annotated as evaluate/residualize here. The parameter is thus bound at specialization time, but also appears in the residual program. The parameter `lp` that is annotated as dynamic in Figure 9 is annotated as identity here and thus only appears in the residual program. The body of the function is annotated as rebuild; this block must be reconstructed because it contains some dynamic computations. The statements in lines (2), (3), (5), and (6) are annotated as completely static in the result of the evaluation-time analysis. These statements are annotated EV by the action analysis, meaning that they are evaluated away during specialization. Finally, the assignment in line (4) is annotated as completely dynamic in the result of the evaluation-time analysis. It is annotated as ID by the action analysis, meaning that it is reproduced unchanged in the specialized program.

The treatment of `xdrmem_putlong` does not illustrate the reduce action. This action would be used if the code following the conditional were instead moved

to the else branch of the conditional. In that case, it would be possible to reduce the conditional during specialization, but not necessarily to completely evaluate it, because one of the branches would contain some code that should be residualized.

3.2 Processing

For each form of specialization (compile-time or run-time program specialization, or data specialization), Tempo creates a dedicated specializer, also known as a generating extension [4,27], based on the action-annotated program. This dedicated specializer is compiled and provided to users. If the dedicated specializer was created for compile-time specialization, a user links the specializer with any needed external functions and applies the resulting program to a set of configuration values to obtain specialized C source code. If the dedicated specializer was created for run-time specialization, a user links the specializer with the program that should use the specialized code, applies the specializer to the configuration values in the course of the program when they become available, and then invokes the generated code directly, as needed. Data specialization is also carried out at run time. We now examine each form of specialization in more detail.

3.2.1 Compile-time specialization

For compile-time specialization, the dedicated specializer consists of two kinds of functions: *code-generation functions* that construct the abstract syntax tree of the specialized program, and *evaluation functions* that contain the fragments annotated as EV by the action analysis. The essence of the dedicated specializer for `xdrmem_putlong` is shown in Figure 11 (this code is simplified to hide the complexities of constructing an abstract-syntax tree). Source program variables are represented as fields in the global structures `_local_sstore` and `_store`. Thus, the configuration parameter `xdrs` of `xdrmem_putlong` is initialized by setting the location `_local_sstore.xdrmem_putlong_xdrs` to the desired value. The entry point of specialization is the function `specialize_putlong`. In this simplified definition, this function calls `instantiate` to generate specialized code. The function `instantiate` is applied to a string, representing the code to be generated, and some function pointers, which are invoked in sequence to fill in the string positions indicated by `%s` with the results of specializing the subterms. In the example, there are three function pointers: `RES_1` for the EV code in lines (2) and (3) of the function `xdrmem_putlong` (Figure 10), `RES_2` for the ID assignment in line (4), and `RES_3` for the EV code in lines (5) and (6). The functions `RES_1` and `RES_3` call evaluation functions `EV_1` and `EV_2`, respectively, which execute the original code. The function

```

// Specialization state
struct _local_sstore_s {
    struct XDR *xdrmem_putlong_xdrs;
    ...
} _local_sstore;

struct _global_sstore_s {
    int xdrmem_putlong_return;
    ...
} _store;

// Code generation functions
char *specialize_putlong() {
    return
        instantiate(
            "void xdrmem_putlong(XDR * xdrs, long * lp) { %s %s %s }",
            RES_1, RES_2, RES_3);
}

char *RES_1() {
    EV_1();
    return "{}";
}

char *RES_2() {
    return "*(long *)xdrs->x_private = (long)htonl((u_long)(*lp));";
}

char *RES_3() {
    EV_2();
    return "{}";
}

// Evaluation functions
void EV_1() {
    if ((_local_sstore.xdrmem_putlong_xdrs->x_handy -= sizeof(long)) < 0) {
        _store.xdrmem_putlong_return = FALSE;
        return;
    }
}

void EV_2() {
    _local_sstore.xdrmem_putlong_xdrs->x_private += sizeof(long);
    _store.xdrmem_putlong_return = TRUE;
    return;
}

```

Fig. 11. Dedicated compile-time specializer for `xdrmem_putlong`

`RES_2` simply returns the dynamic assignment statement.

After specialization, the resulting abstract syntax tree is first passed to a post-processing phase that optionally performs inlining and some code simplifications, and then translated to C code that is returned to the user.

3.2.2 Run-time specialization

Unlike compile-time specialization, run-time specialization must generate executable code directly from the dynamic source-program constructs. To address this issue, Tempo collects the dynamic constructs into a *template file*, which is compiled in advance using `gcc` and used at run time as a repository of fragments of executable code, known as *templates* [28], from which to construct a specialized definition. Figure 12 shows the function corresponding to `xdrmem_putlong` from the template file. Because `xdrmem_putlong` contains

```

extern void tmp_xdrmem_putlong(struct XDR *xdrs, int *lp) {
    static void *LA_tmp_xdrmem_putlong[2] =
        {(void *)tmp_xdrmem_putlong,
         (void *)_tmp_xdrmem_putlong};

    *(long *)xdrs->x_private = (long)htonl((u_long)(*lp));
}

```

Fig. 12. Function from the template file corresponding to `xdrmem_putlong`

only a single dynamic assignment, this function contains only a single template. In general, a function can contain multiple templates, each corresponding to a sequence of REB or ID constructs and delimited by labels in the template file.

The actual specialization process is carried out by a dedicated run-time specializer, which evaluates the static constructs, copies selected templates into a buffer representing the specialized definition, and instantiates these templates with computed static values and with appropriate branch offsets. These operations are simple and thus the cost of specialization is typically amortized after only a few uses of the generated code [29]. The specializer for `xdrmem_putlong` is shown in Figure 13. This function allocates space for the specialized instance (line (1)), emits the only template associated with the function (line (2)), instantiates a library call (the call to `htonl`) in this template with the offset of the called function from the position of the call in the specialized code (line (3)), and then performs the static computations. In general, templates are generated throughout the run-time specializer, at points corresponding to the placement of the dynamic code in the source program. The result of the run-time specializer is a pointer to the specialized function. If the corresponding source function returns a static result, this value is returned in a global variable, here `_xdrmem_putlong_return`.

Consel and Noël [7] and Noël *et al.* [29] present more details about the run-time specializer in Tempo, from both a theoretical and a practical perspective. Refinements of this approach allow inlining of specialized functions within the specialized definition [30].

3.2.3 Data specialization

Data specialization separates the computation of the program into a *loader* and a *reader*. The loader is invoked once with the static configuration values and evaluates the static constructs of the source program. The reader is invoked as needed with the dynamic inputs and carries out the remaining computations. The loader and reader communicate via a cache that contains the values of static expressions that occur in a dynamic context; this cache amounts to the specialized data. Because both the loader and reader are generated at compile

```

void *rts_xdrmem_putlong(struct XDR *xdrs) {
    char *buffer = (*rts_alloc_code)(20000);           (1)
    char *data_buffer = (*rts_alloc_data)(20000);
    char *code_ptr = buffer;
    char *data_ptr = data_buffer;
    char *spec_ptr = code_ptr;
    char *temp_43_addr;

    temp_43_addr = 0;
    DUMP_TEMPLATE(code_ptr, temp_43_addr, (char *)tmp_xdrmem_putlong + 0, 48); (2)
    PATCH_LIB_CALL(temp_43_addr, (char *)tmp_xdrmem_putlong + 0, 20); (3)
    if ((xdrs->x_handy -= sizeof(long)) < 0) {
        _xdrmem_putlong_return = FALSE;
        return (void *)spec_ptr;
    }
    xdrs->x_private += sizeof(long);
    _xdrmem_putlong_return = TRUE;
    return (void *)spec_ptr;
}

```

Fig. 13. Dedicated run-time specializer for `xdrmem_putlong`

time, before the static data is known, data specialization does not simplify conditionals or unroll loops. The lack of these transformations implies that data specialization often gives less performance improvement than program specialization, in which new code is generated based on the static values. On the other hand, because data specialization cannot lead to code explosion, it is useful for some applications where excessive code would be generated by program specialization [11].

Figure 14 shows the loader and reader generated for `xdrmem_putlong`. We have slightly modified the code to place the code following the conditional statement in the original implementation (Figure 2) in the else branch of the conditional; cache management in the current implementation of data specialization in Tempo requires that if any branch contains a return statement then they all do. Although static and dynamic portions of `xdrmem_putlong` are largely disjoint, because static conditionals are not reduced by data specialization, the static value of the conditional test (line (2) of Figure 10) must be communicated from the loader to the reader via the cache. Lines (1) and (2) in the loader initialize the cache entry according to the value of the test. Line (3) in the reader accesses this value to choose the branch to execute, without re-evaluating the test expression. Both the loader and the reader return a pointer to the next free position in the cache. Because passing a value from the loader to the reader through the cache involves memory references, the cost of using the cache may be more expensive than the cost of simply re-evaluating a very simple expression, such as a variable reference. Tempo thus gives the program developer some control over the caching strategy.

Because data specialization has slightly different properties than program specialization, the preprocessing phase of Tempo must be directed specifically to prepare the code for data specialization. Nevertheless, the differences are minor, and the same analysis engine is used. More information about data

```

void **xdrmem_putlong_2_loader(struct XDR *xdrs, void **Cache) {
    if ((xdrs->x_handy -= sizeof(long)) < 0) {
        *((int *)Cache) = 1;           // test expression value      (1)
        _xdrmem_putlong_return = FALSE;
        return Cache + 1;             // new cache pointer
    }
    else {
        *((int *)Cache) = 0;           // test expression value      (2)
        xdrs->x_private += sizeof(long);
        _xdrmem_putlong_return = TRUE;
        return Cache + 1;             // new cache pointer
    }
}

void **xdrmem_putlong_2_reader(struct XDR *xdrs, int *lp, void **Cache) {
    if (*(int *)Cache)                 // access cached test value      (3)
        return Cache + 1;
    else {
        *(long *)xdrs->x_private = (long)htonl((u_long)(*lp));
        return Cache + 1;
    }
}

```

Fig. 14. Loader and reader for data specialization of `xdrmem_putlong`

specialization in Tempo is available in the work of Chirokoff *et al.* [11] and Lawall [30].

4 Tempo as a Back-End Specializer

Specialization has been explored for a variety of realistic languages. Nevertheless, developing a specializer for such a language remains a daunting task. A promising alternative is to explore specialization for a new language by translating programs into a language for which there already exists a program specializer. This approach exploits an existing and stable infrastructure, and thus permits the developer to concentrate on issues particular to the targeted language. In our exploration of this approach using Tempo, we have found that the C language is suitable to accurately express the semantics of a wide variety of programming languages. Indeed, a number of compilers use C as their target language [31–33]. Our study of back-end specialization mainly focused on two languages whose specialization had not previously been addressed, namely C++ and Java.

To translate C++ programs into C, we use the `cfront` compiler. The translated code is amenable to successful specialization, modulo minor transformations to reduce its size by eliminating unnecessary program fragments. Specialization of C++ was used to specialize industrial-strength systems code: parts of the Chorus inter-process communication subsystem [34]. The main limitation of this approach was its inability to produce residual programs in the initial source language, that is, C++.

This limitation was addressed by a subsequent project aimed to use Tempo as a back-end specializer for Java. The translation from Java to C is performed by the Harissa compiler, developed in the Compose group [31,35]. Numerous optimizations are built into the Harissa compiler and run-time system, including method inlining driven by a class hierarchy analysis. Furthermore, this translation exposed specialization opportunities concerning language features such as virtual calls, casts and array references. Yet, some specialization opportunities inherent to the source program appeared to be difficult for Tempo to exploit because of the heavy use of data structures and dynamic memory allocation in the code generated by Harissa. To cope with this kind of code, Tempo's analyses were extended with a more precise treatment of structures. Also, to make back-end specialization transparent to the programmer, a translator from C to Java was developed. This back translator only handles the subset of C (and program patterns) that can be generated by Harissa and output by Tempo. This project led to a program specializer named JSpec [16,36,37].

5 Applications

Tempo has mainly been used in two areas: operating systems and compiler generation. In this section, we give an overview of applications of Tempo in these areas and discuss the resulting performance improvements.

5.1 *Operating Systems*

An early success of the Tempo project was the specialization of the XDR layer of the RPC mechanism. The computations optimized away by Tempo mostly include the testing of the coding direction, the checking of buffer overflow, and the testing of return status. The last optimization critically relies on return sensitivity, allowing a static return value in a callee to flow back to a caller. This feature allows computations depending on the return value to be performed during specialization.

We specialized the XDR layer with respect to different values for the size of an array to be used as an RPC argument. The specialized versions of this layer ran between 165% and 235% faster than the original version on a PC under Linux. The overall roundtrip performance improvement of the RPC was 35% on the same platform [3,20]. These experiments were carried out using compile-time specialization. Kono and Masuda have applied run-time specialization using Tempo to the problem of marshaling data and obtain similar performance improvements [38].

Another systems application studied in the context of Tempo was UNIX signals. This mechanism allows processes to communicate events. Specialization opportunities occur when a process sends the same signal to the same destination process in the course of some collaborative work. In this situation a number of comparisons and conditionals can be eliminated; the residual code solely consists of operations to deliver the signal to the target process. When specialized, the resulting system call is 65% faster than the original code [3].

5.2 *Compiler Generation*

The long line of work on specializing interpreters, initiated by Jones, has found a number of follow-ups in our systems work. One such application is the specialization of a low-level interpreter for selecting packets from a network interface, named the Berkeley Packet Filter (BPF) [39]. This kernel-resident interpreter runs packet filter programs written in a bytecode by application programmers. The BPF, like any interpreter, can be specialized with respect to a particular program, thus eliminating most, if not all, of the interpretation overhead.

The BPF interpreter was specialized both at compile time and run time to account for opportunities existing at these two stages. On a Pentium, the compile-time specialized BPF interpreter was 3.4 faster than the original version, and 1.95 faster when specialized at run time [40]. The difference in performance is due to the fact that compile-time specialization produces a program that can be globally optimized by a conventional compiler. In contrast, the run-time specializer assembles binary templates that are only locally optimized.

Another case study is an interpreter for a language, named PLAN-P, that allows a programmer to define application-specific network protocols [41,42]. To achieve maximum flexibility, protocols need to be deployed dynamically over a network of programmable routers. In order to satisfy portability, safety and security constraints, interpretation appears to be a natural strategy to run programs. Yet, using run-time specialization, we demonstrated that Tempo could essentially achieve just-in-time compilation, thus reconciling the need to manipulate the source representation of programs for verification purposes with the need for efficiency. The specialized PLAN-P interpreter was found to be 35% faster than an equivalent compiled and optimized Java program [40].

6 Conclusions and Future Work

The goal of the Tempo project has been to make specialization a practical tool to optimize realistic programs. Examination of realistic specialization cases prompted us to develop new analyses and transformation techniques for Tempo. The resulting specializer has been continuously tested against a variety of applications in operating systems, networking and interpreters. The contributions of this research project reflect the breadth of its scope; it includes results in program analyses, program transformation, language design, software engineering, operating systems, and networking. Tempo's capabilities and features open a host of new research directions. We conclude by describing some of this current work.

Specialization modules have facilitated the use of Tempo by non-experts, including industry engineers. However, much work remains to enable the integration of Tempo in an industrial-strength software development process. Necessary improvements include finer-grained description of specialization scenarios and a high-level way to specify effects of external functions. Furthermore, it can be useful to control the size and time usage of specialized programs, for example in the domain of embedded systems. Consequently, we plan to investigate techniques to characterize the effects of specialization and to restrict some transformations to guarantee such properties as a maximum possible size of the specialized program.

Another direction consists of exploring the combination of program specialization and application generators. To ease their development, application generators usually produce generic code and rely on highly-parameterized libraries. Because both the libraries and the application generator are fixed, all possible generated applications can benefit from a fixed set of specialization scenarios. Encouraging results along this line have been obtained in the context of the RPC, which is implemented as a stub compiler and the XDR library.

Finally, we would like to couple specialization with declarations and mechanisms to enable specialized code to be integrated safely and correctly as the program executes. This is particularly interesting for long-running systems that critically rely on performance but whose execution cannot be suspended.

References

- [1] ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Portland, OR, USA, 2002.
- [2] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana,

- J. Walpole, K. Zhang, Optimistic incremental specialization: Streamlining a commercial operating system, in: Proceedings of the 15th ACM Symposium on Operating Systems Principles, ACM Operating Systems Reviews, 29(5), ACM Press, Copper Mountain Resort, CO, USA, 1995, pp. 314–324.
- [3] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, C. Goel, C. Consel, G. Muller, R. Marlet, Specialization tools and techniques for systematic optimization of system software, ACM Transactions on Computer Systems 19 (2001) 217–251.
- [4] N. Jones, C. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, International Series in Computer Science, Prentice-Hall, 1993.
- [5] C. Consel, O. Danvy, Tutorial notes on partial evaluation, in: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, ACM Press, Charleston, SC, USA, 1993, pp. 493–501.
- [6] N. Jones, P. Sestoft, H. Søndergaard, An experiment in partial evaluation: the generation of a compiler generator, in: J.-P. Jouannaud (Ed.), Rewriting Techniques and Applications, Vol. 202 of Lecture Notes in Computer Science, Springer-Verlag, 1985, pp. 124–140.
- [7] C. Consel, F. Noël, A general approach for run-time specialization and its application to C, in: Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, USA, 1996, pp. 145–156.
- [8] R. Glück, J. Jørgensen, An automatic program generator for multi-level specialization, Lisp and Symbolic Computation 10 (1997) 113–158.
- [9] R. Marlet, C. Consel, P. Boinot, Efficient incremental run-time specialization for free, in: Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99), Atlanta, GA, USA, 1999, pp. 281–292.
- [10] K. Malmkjær, Program and data specialization: Principles, applications, and self-application, Master's thesis, DIKU University of Copenhagen (Aug. 1989).
- [11] S. Chirokoff, C. Consel, R. Marlet, Combining program and data specialization, Higher-Order and Symbolic Computation 12 (4) (1999) 309–335.
- [12] A. Bondorf, J. Jørgensen, Efficient analyses for realistic off-line partial evaluation, Journal of Functional Programming 3 (3) (1993) 315–346.
- [13] C. Consel, A tour of Schism: a partial evaluation system for higher-order applicative languages, in: Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Copenhagen, Denmark, 1993, pp. 66–77.
- [14] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, R. Glück, Fortran program specialization, in: U. Meyer, G. Snelting (Eds.), Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen, Justus-Liebig-Universität, Giessen, Germany, 1994, pp. 45–54, report No. 9402.

- [15] C. Consel, L. Hornof, F. Noël, J. Noyé, E. Volanschi, A uniform approach for compile-time and run-time specialization, in: O. Danvy, R. Glück, P. Thiemann (Eds.), *Partial Evaluation, International Seminar, Dagstuhl Castle*, no. 1110 in *Lecture Notes in Computer Science*, 1996, pp. 54–72.
- [16] U. Schultz, J. Lawall, C. Consel, G. Muller, Towards automatic specialization of Java programs, in: *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, Vol. 1628 of *Lecture Notes in Computer Science*, Lisbon, Portugal, 1999, pp. 367–390.
- [17] L. Hornof, J. Noyé, C. Consel, Effective specialization of realistic programs via use sensitivity, in: P. Van Hentenryck (Ed.), *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, Vol. 1302 of *Lecture Notes in Computer Science*, Springer-Verlag, Paris, France, 1997, pp. 293–314.
- [18] L. Hornof, J. Noyé, Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity, in: *PEPM'97* [43], pp. 63–73.
- [19] L. Hornof, J. Noyé, Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity, *Theoretical Computer Science* 248 (1–2) (2000) 3–27.
- [20] G. Muller, R. Marlet, E. Volanschi, C. Consel, C. Pu, A. Goel, Fast, optimized Sun RPC using automatic program specialization, in: *Proceedings of the 18th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Amsterdam, The Netherlands, 1998, pp. 240–249.
- [21] A.-F. Le Meur, J. Lawall, C. Consel, Towards bridging the gap between programming language and partial evaluation, in: *PEPM'02* [1], pp. 9–18.
- [22] A. Birrell, B. Nelson, Implementing remote procedure calls, *ACM Transactions on Computer Systems* 2 (1) (1984) 39–59.
- [23] A.-F. Le Meur, *Approche déclarative à la spécialisation de programmes C*, Thèse de doctorat, Université de Rennes 1, France (Dec. 2002).
- [24] A.-F. Le Meur, C. Consel, B. Escrig, An environment for building customizable software components, in: *IFIP/ACM Conference on Component Deployment*, Berlin, Germany, 2002, pp. 1–14.
- [25] A. Erosa, L. Hendren, Taming control flow: A structured approach to eliminating goto statements, *ACAPS Technical Memo 76*, School of Computer Science, McGill University, Montreal, Canada (Sep. 1993).
- [26] G. Muller, E. Volanschi, R. Marlet, Scaling up partial evaluation for optimizing the Sun commercial RPC protocol, in: *PEPM'97* [43], pp. 116–125.
- [27] R. Glück, J. Jørgensen, Efficient multi-level generating extensions for program specialization, in: M. Hermenegildo, S. Doaitse Swierstra (Eds.), *Proceedings of the 7th International Symposium on Programming Language Implementation and Logic Programming*, Vol. 982 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, 1995, pp. 259–278.

- [28] D. Keppel, S. Eggers, R. Henry, Evaluating runtime compiled value-specific optimizations, Technical Report 93-11-02, Department of Computer Science, University of Washington, Seattle, WA (1993).
- [29] F. Noël, L. Hornof, C. Consel, J. Lawall, Automatic, template-based runtime specialization: Implementation and experimental study, in: International Conference on Computer Languages, IEEE Computer Society Press, Chicago, IL, 1998, pp. 132–142, also available as IRISA report PI-1065.
- [30] J. Lawall, Implementing circularity using partial evaluation, in: O. Danvy, A. Filinski (Eds.), Programs as Data Objects, Second Symposium, PADO 2001, Vol. 2053 of Lecture Notes in Computer Science, Springer, Aarhus, Denmark, 2001, pp. 84–102.
- [31] G. Muller, B. Moura, F. Bellard, C. Consel, Harissa: A flexible and efficient Java environment mixing bytecode and compiled code, in: COOTS97 [44], pp. 1–20.
- [32] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, S. Watterson, Toba: Java for applications - a way ahead of time (WAT) compiler, in: COOTS97 [44], pp. 41–53.
- [33] S. Feldman, D. Gay, M. Maimone, N. Schryer, A Fortran to C converter, Computing Science Technical Report 149, AT&T Bell Laboratories, Murray Hill, NJ (Mar. 1995).
- [34] E. Volanschi, Une approche automatique à la spécialisation de composants système, Thèse de doctorat, Université de Rennes I (Feb. 1998).
- [35] G. Muller, U. Schultz, Harissa: A hybrid approach to Java execution, IEEE Software (1999) 44–51.
- [36] U. P. Schultz, J. L. Lawall, C. Consel, Specialization patterns, in: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000), IEEE Computer Society Press, Grenoble, France, 2000, pp. 197–208.
- [37] U. P. Schultz, J. L. Lawall, C. Consel, Automatic program specialization for java, ACM Transactions on Programming Languages and Systems To appear.
- [38] J. Kono, T. Masuda, Efficient RMI: Dynamic specialization of object serialization, in: Proceedings of the 20th International Conference on Distributed Computing Systems, IEEE Computer Society Press, Taipei, Taiwan, 2000, pp. 308–315.
- [39] S. McCanne, V. Jacobson, The BSD packet filter: A new architecture for user-level packet capture, in: Proceedings of the Winter 1993 USENIX Conference, USENIX, San Diego, CA, USA, 1993, pp. 259–269.
- [40] S. Thibault, C. Consel, R. Marlet, G. Muller, J. Lawall, Static and dynamic program compilation by interpreter specialization, Higher-Order and Symbolic Computation 13 (3) (2000) 161–178.

- [41] S. Thibault, C. Consel, G. Muller, Safe and efficient active network programming, in: 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana, 1998, pp. 135–143.
- [42] S. Thibault, J. Marant, G. Muller, Adapting distributed applications using extensible networks, in: Proceedings of the 19th International Conference on Distributed Computing Systems, IEEE Computer Society Press, Austin, TX, 1999, pp. 234–243.
- [43] ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Amsterdam, The Netherlands, 1997.
- [44] Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, Usenix, Portland (Oregon), USA, 1997.