INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Mapping Software Architectures to Efficient Implementations via Partial Evaluation

Renaud Marlet , Scott Thibault , Charles Consel

N° 3217

25 juillet 1997

THÈME 2

Rapport de recherche

ISSN 0249-6399

Mapping Software Architectures to Efficient Implementations via Partial Evaluation

Renaud Marlet , Scott Thibault , Charles Consel

Thème 2 — Génie logiciel
et calcul symbolique
Projet LANDE / avant-projet COMPOSE

Rapport de recherche n° 3217 — 25 juillet 1997 — 18 pages

Abstract: Flexibility is recognized as a key feature in structuring software, and many architectures have been designed to that effect. However, they often come with performance and code size overhead, resulting in a flexibility *vs.* efficiency dilemma. The source of inefficiency in software architectures can be identified in the data and control integration of components, because flexibility is present not only at the design level but also in the implementation.

We propose the use of program specialization in software engineering as a systematic way to improve performance and, in some cases, to reduce program size. In particular, we advocate the use of partial evaluation, which is an automatic technique to produce efficient, specialized instances of generic programs. We study several representative, flexible mechanisms found in software architectures: selective broadcast, pattern matching, interpreters, software layers, and generic libraries. We show how partial evaluation can systematically be applied in order to optimize those mechanisms.

Key-words: Software architecture, software engineering, partial evaluation, flexibility, performance, selective broadcast, pattern matching, interpreter, software layer, generic library.

(Résumé : tsvp)

This work has been partly supported by FRANCE TELECOM, CNET 96-1B-027. It has been submitted for publication.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 02 99 84 71 00 – Télécopie : (33) 02 99 84 71 71

Implémentations efficaces d'architectures logicelles via évaluation partielle

Résumé : La flexibilité est reconnue comme une caractéristique majeure dans la structuration des logiciels ; de nombreuses architectures ont été conçues à cet effet. Cependant, cette flexibilité se paie : la performance est souvent médiocre et le code correspondant assez gros. Il en résulte un dilemme entre flexibilité et efficacité. Après examen, la source de l'inefficacité des architectures logicielles s'avère être l'intégration des données et du contrôle lorsque l'on réunit les divers composants d'un système. La raison tient à ce que la flexibilité n'apparaît pas uniquement au stade de la conception, mais se matérialise également lors du développement.

Nous proposons d'utiliser la spécialisation de programmes comme moyen systématique d'améliorer la performance et, dans certains cas, de réduire la taille des programmes. Plus particulièrement, nous préconisons l'usage de l'évaluation partielle, qui est une technique permettant de produire automatiquement des instances spécialisées efficaces de programmes génériques. Nous étudions pour cela plusieurs implémentations représentatives de mécanismes de flexibilité employés dans diverses architectures logicielles : diffusion sélective, recherche de motifs, interprètes, couches logicielles, librairies génériques. Nous montrons comment une application systématique de l'évaluation partielle conduit à l'optimisation de ces mécanismes.

Mots-clé : Architecture logicielle, génie logicielle, évaluation partielle, flexibilité, performance, diffusion sélective, recherche de motifs, interprète, couche logicielle, librairie générique.

1 Introduction

Software architectures express how systems should be built from various components and how those components should interact. It is widely accepted that as the size and complexity of software systems increase, the choice of software architectures becomes a major issue. This choice has a great impact on software engineering aspects such as the cost of development, validation and maintenance. Because it also affects the extensibility and interoperability of systems, it can have large impacts on time-to-market, providing the competitive advantage or disadvantage. In practice, typical concerns of software architectures include reuse and safety.

Additionally, a modern software system is characterized by its changing nature: computation may be distributed over a network of heterogeneous machines and components, where tasks can migrate at runtime; connections between software components can evolve in time and space; hardware platforms offer vastly different functionalities and performance; software environments provide applications with changing services; etc. This calls for programs that are able to adapt to changing parameters.

Therefore, *flexibility* can be identified as a key feature of software architectures. Instances of flexibility include re-usability, extensibility, genericity and adaptability. Many approaches aimed at achieving software flexibility have been proposed and put into practice, including pipes and filters [2], layered systems [25], data abstraction and object-oriented organization, event-based communication [33, 20], coordination [9] and domain-specific languages [44], and software buses [29].

However, flexibility causes non-negligible overhead. In practice, it requires having lots of components, or that the components should be generic. Therefore, computation often traverses software connectors, and some amount of code and execution time is devoted to gluing components together rather than spent in the components themselves. Similarly, using a generic component, in which many cases have been anticipated, is less efficient than using a specific component that only provides the required service for the given context. Whereas efficiency requires a tighter integration of components, flexibility calls for a greater separation. Actually, the reason why flexibility usually impairs efficiency is that flexibility is not only present at the design level but also in the implementation.

Efficiency is a fierce rival to flexibility in the concerns of software engineers. Manual optimizations have been proposed and used [24, 32]. But they are ad hoc (as opposed to systematically applicable), tedious, error prone and unpredictable; they do not scale up. In addition, they conflict with other software engineering concerns like maintenance and extensibility. In order to get the best of both worlds, there has been a major research effort aimed at achieving efficient generation and composition of building blocks [3, 4]. However, these approaches are also specific to the architectures and the domains of components. Moreover, they do not fully exploit all integration opportunities.

We propose a *systematic* method to improve the efficiency of software architectures, known as *program specialization*. Program specialization is a general program transformation that exploits the gluing information and the parameterization of components in order to better integrate them. Thus, it is not specific to a peculiar software architecture style. *Partial evaluation* is the technique that *automates* program specialization. It does not conflict with the purposes of software engineering, as opposed to manual optimizations.

In order to show that program specialization actually applies to many architecture styles, and that it indeed improves efficiency, we have studied several typical integration mechanisms used in software architectures, including selective broadcast, pattern matching, interpreters, layers, and generic libraries. We have applied partial evaluation to representative instances of those mechanisms, leading to a gain in efficiency.

Our contributions can be summarized as follows:

- We identify the fundamental reasons why mapping flexible software architectures into implementations leads to efficiency problems.
- We propose a systematic and automatic technique (partial evaluation) to turn software architectures into efficient implementations.
- We study five representative instances of mechanisms used in software architectures (selective broadcast, pattern matching, interpreters, layers, and generic libraries) and show how partial evaluation does indeed improve efficiency while retaining flexibility.

The rest of paper is organized as follows. Section 2 identifies the general sources of inefficiency in software architectures. Section 3 discusses program specialization and partial evaluation. Section 4 considers in turn several mechanisms used in software architectures and the applicability of partial evaluation. In the conclusion, we give some research directions for further improvements.

2 The Flexibility vs. Efficiency Dilemma

Many approaches have addressed the need for flexibility in software engineering, with various trade-offs with efficiency. They may be characterized by the way software components interact, *i.e.* what data they exchange and how they communicate. After examining these issues in turn, this section elaborates on the flexibility vs. efficiency dilemma.

Data Integration

Software systems are made of components that exchange or share data. The components may not use the same data representation. Such a situation occurs for example when an existing software component is being reused in a different context, when components are programmed using different languages, when components run on different systems or hardware platforms, or when the execution environment is distributed (network encapsulation of data). *Data integration* addresses the problems caused by data heterogeneity.

When a system is heterogeneous, data communication between components requires conversions. Two main approaches have been proposed. One is to systematically convert local data into a universal format that is used in all inter-component communications. The universal format may be provided by an intermediate data description languages such as ASN.1 [19] or IDL [35]. Each data communication necessitates two conversions. On the other hand, data converters are small because each component only needs a conversion between its own internal representation and the universal one.

Another solution is to push all conversions onto the callee. The called component examines a tag included in the received data to determine whether these data need to be converted; at most one conversion is needed. However, the number of converters is not linear but quadratic in the number platforms. Flexibility is reduced because adding a new data format requires more work.

Control Integration

Besides unifying data formats, composing software components also involves strategies to make these components communicate. This process is often referred to as *control integration*. Numerous styles have been proposed and put into practice, including pipes and filters [2], layered systems [25], data abstraction and object-oriented organization, event-based communication [20, 33], coordination [9] and domain-specific languages [44], and software buses [29].

There exist basically two main ways to achieve communication between software components: states and events. A *state* consist in some information available at any time. The access to a state may be explicit (*e.g.*, reference to a global variable) or implicit (*e.g.* with blackboarding techniques [28]). On the other hand, an *event* is a transfer of information that occurs at a discrete time, for example via a procedure call or sending a message. Event mechanisms may further be categorized into implicit and explicit invocation. For example, broadcasting a message is only an *implicit invocation* of procedures in other components. This is in contrast with systems in which the interface of components only consist of a collection of routines; communication is then based on direct, *explicit invocation* of those routines (*e.g.*, as in a system organized around a main program calling subroutines). In an object-oriented system, invocation of methods is functionally explicit but actually involves an implicit object dispatch indirection. Generic components can also contain aspects of implicit invocation at a finer-grain, where parameters of the component are used to select various behaviors of the component.

It is obvious that explicit procedure invocation and direct state referencing is faster than the implicit mechanisms. On the other hand, implicit mechanisms often offer more flexibility. Likewise for generic components, parameters provide a flexible mechanism of controlling control flow within a component in a black-box manner. Here efficiency is lost because execution time is spent in testing options and checking assertions about the arguments, as opposed to actually providing the required service. Manually building specific components reduces this overhead but also lessens flexibility and goes against many software engineering goals.

Control integration also has a significant impact on code size. In fact, adaptability calls for the anticipation of many use contexts. Given a specific usage context, only a few cases are needed, the other cases can be viewed as dead code. If this dead code cannot be eliminated, the code size of the whole system is unnecessarily large. This issue is very important for embedded and mobile code.

Improving Efficiency While Retaining Flexibility

As seen above, the reason why flexibility introduces overhead is that genericity and adaptability are usually present not only in structuring a system (*i.e.*, at the design level) but also in its implementation. Therefore, a natural approach to remove this overhead is to keep flexibility in the design but to obtain somehow an efficient implementation that is not necessarily flexible.

In practice, the integration of data and control that is expressed in the architecture should be made tighter in the implementation: the number of conversions must be reduced; implicit control must be turned into explicit control; generic components must be adapted to specific uses. Also, flexibility might actually be used at different stages of the assembly of a whole software system. Therefore, efficient implementations are needed at different times: configuration time, compile time, link time, opening session/initialization time, and runtime. In practice, the later adaptation is needed, the more difficult it is to implement it efficiently.

Currently, some software engineering environments can generate code from a flexible specification, especially in the domain of libraries [3, 4]. However, the generated code may still contain aspects of the software architecture in the specification. These techniques are also ad hoc in that they are specific to a given architecture. Furthermore, they only address compile-time code generation.

Actually, the central idea in optimizing components integration is *specialization*. It ranges from the specialization of the connection between components to the complete merging of the components functionalities. In fact, there is a technique known as *program specialization* (or *program adaptation*) that precisely has that goal. This technique is detailed in the next section.

3 Program Specialization and Partial Evaluation

Specialization is a program transformation that adapts programs with respect to known information about their inputs. We first give a short overview of specialization and present how it has been put into practice. In particular, we focus on partial evaluation, a process that automates specialization.

3.1 Specialization in a Nut Shell

Principles. Let us consider a program p, taking some argument data d and producing a result r, which may be written as $p(d) = r$. If d may be split into $d = (d_1, d_2)$ where d_1 is a *known* (*i.e.*, it does not vary) subset of the input and d_2 is yet *unknown*, we may form a new program (p, d_1) that waits until d_2 is available and then calls the original p program on (d_1, d_2) to produce the same result r. In other words, $(p, d_1)(d_2) = p(d_1, d_2) = r$. However, now that d_1 is known to p, computations relying on d_1 can be performed before d_2 is actually available. Therefore, we can form a new program p_{d_1}, equivalent to (p, d_1), where computations depending on d_1 have been exploited. We thus have $p_{d_1}(d_2) = p(d_1, d_2) = r$. The program p_{d_1} is called a *specialization* of p with respect to the *invariant* d_1. More generally, specialization exploits any invariant present in the code, not only input values. The idea is to factor out computations from the specialized program.

Example. Let us consider the simple example shown in Figure 1. On the left-hand side stands the definition of `mini_printf`, a simplified version of the Unix `printf` text formatting function. On the right-hand side stands a specialized version of `mini_printf` with respect to the string format `fmt` = `"n = %d"`. Note that all computations depending on `fmt` (*i.e.*, interpretation of format string) have been removed. Bold face font is used here (and in the rest of the paper) to highlight parts of the original program that rely only on the known `fmt` value. All those computations disappear in the specialized program.

```
mini_printf(char fmt[], int val[])
{
  int i = 0;
  while( *fmt != '\0' ) {
    if( *fmt != '%' )
       putchar(*fmt);
    else
      switch(*++fmt) {
        case 'd' : putint(val[i++]); break;
        case '%' : putchar('%'); break;
        default  : abort();  /* error */
      }
    fmt++;
  }
}
```

```
mini_printf_fmt(int val[])
{
  putchar('n');
  putchar(' ');
  putchar('=');
  putchar(' ');
  putint(val[0]);
}
```

Figure 1: Specialized `mini_printf` with respect to `fmt = "n = %d"`

Advantages. Program specialization may reduce both execution time and code size. Indeed, running $p_{d_1}(d_2)$ is usually *faster* than running $p(d_1, d_2)$ because computations involving d_1 are already performed. In addition, cases written to treat other inputs than d_1 can be removed from p_{d_1}; they are dead code. Program p_{d_1} is thus *smaller*. On the other hand, specialization can also involve loop unrolling, which may increase code size. For example, in Figure 1, the whole loop has been unrolled; if the format string had been longer, many `putchar()` calls would have appeared in the specialized program. Besides, if building the specialized program p_{d_1} comes with a certain price, it is worth it only if $p_{d_1}(d_2)$ is run enough times to amortize the cost of building p_{d_1}.

Concerning software engineering issues, specialization produces monolithic, specialized code from a modular, generic system. Most software engineering qualities of the original code are lost in this process. Thus, the specialized code should be considered as an opaque pre-compilation rather than the starting point of further manual developments.

3.2 Manual, Ad Hoc Approaches to Specialization

Various studies have demonstrated that significant optimizations could be obtained via program specialization. Examples of such experiments can be found in different areas such as graphics [24] and operating systems [32]. However, these studies have been limited to *manual* code transformation. Because it is tedious and error prone, manual specialization is generally local, *i.e.* restricted to a small "window" of code (as opposed to inter-procedural optimizations); it does not scale up to large systems. In addition, because it is not automatic, manual specialization trades safety, maintainability and extensibility for efficiency, which defeats software engineering purposes. Finally, techniques proposed are ad hoc (*i.e.*, not systematic); it is not clear how they could be extended and applied in general.

There already exists tools that provide some primitive support for specialization. For example, some software and hardware particularities may be expressed at configuration time using tools like `configure`. Others particularities can be handled at compilation time using macro facilities, in addition to simple compiler optimization.

In addition, some smart *compilers* can achieve intra-procedural propagation and folding of scalar constants, as well as inlining. While this is enough to optimize simple parametrization, it does not scale up for the full program adaptation needed for software component integration. Moreover, it is not easy to predict the effect of such optimizations.

3.3 Partial Evaluation

Partial evaluation is "the" technique that *automates* the specialization process [10, 22]. Partial evaluation is also *systematic*, as opposed to *ad hoc* specializations that are restricted to specific cases. Using the same notations as above, a *specializer* (or *partial evaluator*) is a tool that automatically produces the specialized program p_{d_1}, given a program p and a known input subset d_1. Therefore, it improves speed and, in some circumstances, may reduce code size. Roughly speaking, standard partial evaluation can be thought of as a combination of aggressive *inter-procedural* constant propagation (applied to *all* data types instead of just scalars), constant folding, inlining and loop unrolling.

In contrast with manual specialization, partial evaluation is safe and preserves code genericity. It does not conflict with the purposes of software engineering. On the contrary, because it automatically takes care of efficiency issues, it encourages programmers to write generic code. In addition, optimizing code using partial evaluation is much less tedious than doing manual specialization. It is intrinsically made to scale up, as opposed to manual specialization. Optimizations are also more predictable.

Long confined in functional or logic programming, partial evaluation has now been put into practice for imperative languages. It is reaching a level of maturity that makes it applicable to real-sized systems. In fact, not only are there now partial evaluation systems for languages like C, but the program specialization approach is at the basis of the development of adaptable system in a number of major research projects and in different areas such as networking [26, 40], graphics [23], and operating systems [6, 16, 31]. What we are interested in is to use partial evaluation to generate context-specific efficient instances from generic components. As will be demonstrated in the following case studies, partial evaluation systematically and automatically improves implementations of software architectures.

Our claim concerning the applicability of partial evaluation to software engineering is not specific to a language or a partial evaluator. However, we had to use a real tool in our case studies. In the following, we actually use *Tempo*, a partial evaluator for C programs [11] developed in our group. To make sure that Tempo performs optimizations that address realistic cases, it has initially been targeted towards a very demanding application area: system software. There exists another system for C specialization named C-Mix [1]. (See [27] for comparison details.)

Tempo is an *off-line* specializer [10]: partial evaluation is split into two phases. First, a preprocessing phase performs an abstract propagation of known information throughout the code. The output of this analysis can be visualized in a form which is very similar to the font decoration in Figure 1. The user can thus assess the benefits of applying partial evaluation. Second, a processing phase actually performs code generation, given some partial input values. Tempo may exploit values when they are known, at compile time and/or at runtime [14]. Whereas compile-time specialization is a source-to-source program transformation, runtime specialization relies on binary template assembly for fast code generation. Dynamic selection of variously specialized routines is presented in [39]. In this framework, specializations with respect to values that can vary during program execution (*i.e.*, *quasi-invariants*) may be triggered at run time.

4 Case Studies

In order to support our assessment, we consider in turn five mechanisms that are common in software architectures. For each one, (i) we give a short description of the mechanism, taking as an example an architecture and a real system that actually relies on it, (ii) we point out efficiency problems inherent in the mechanism, and (iii) we show how partial evaluation can automatically improve performance and, in some cases, reduce code size.

Specialized code listed in this section has been automatically produced by Tempo, apart from the following manual simplifications aimed at clarity: some transformations, like copy propagation performed by optimizing compilers, were done by hand on the specialized source; code has also been manually pretty-printed; some initializations, as well as type and variable definitions have been omitted. In addition, some comments have been added to the original and specialized code.

All partial evaluation examples displayed in this section are presented as compile-time source-to-source program transformations for readability reasons. When specialization values are known at run time, and even vary during program execution, runtime partial evaluation can generate binary specialized routines (that we cannot display) on the fly. Consequently, partial evaluation does not generally limit the use of flexibility in software architectures. However, partial evaluation techniques cannot be applied to code which is dynamically loaded because all the program (not values) must be known at analysis time.

4.1 Optimizing Selective Broadcast

The Mechanism. Our first case study deals with *selective broadcast*, also called *reactive integration* [34]. In such a context, components are independent agents that interact with each other by sending broadcast events. Components in the system that are interested in peculiar messages register "callback" procedures to be called each time such messages are broadcast. This mechanism is also called *implicit invocation* because broadcasting events "implicitly"

```
my_execution_context()
{
  register_for_event( DEBUG_AT, editor_goto );
  register_for_event( DEBUG_AT, cfg_highlight );
  debug_run();
}
debug_run()
{
  do_stuff();
  broadcast( BUS_ERROR, (char *)NULL );
  dbg_info->line = line;
  dbg_info->fname = fname;
  broadcast( DEBUG_AT, (char *)dbg_info );
  do_more_stuff();
}
```

```
register_for_event(int event, void (*func)(char*))
{
  handler[no_handlers].func = func;
  handler[no_handlers].event = event;
  no_handlers++;
}
broadcast(int event, char *arg)
{
  for (i = 0; i < no_handlers; i++)
    if (handler[i].event == event)
      (*handler[i].func)(arg);
}
```

```
debug_run()
{
  do_stuff();
  dbg_info->line = line;
  dbg_info->fname = fname;
  editor_goto((char*)dbg_info);
  cfg_highlight((char*)dbg_info);
  do_more_stuff();
}
```

Figure 2: Optimization of Registration and Broadcast

invokes procedures in other components. Blackboarding techniques may also be based on similar indirect access mechanisms [18].

The Field programming environment is a typical, representative example of such an architecture [33]. It is an open system that integrates many programming tools. Let us consider a system containing an editor, a debugger and a viewer of control flow graphs. The example in Figure 2 (left-hand side) models a typical communication between those tools. The editor and the flow-graph viewer register their interest in the DEBUG_AT event, which is emitted by the debugger when an execution is stepped or when a breakpoint is reached. When the DEBUG_AT event is received, the editor wants to set the cursor on the line where the debugger stopped, and the flow-graph viewer wants to highlight the name of the current function in the graph. In order to separate concepts well, events are identified here using an integer, and data associated to events is a structure pointer (manipulated as a "dummy" character pointer). Hence, this section only models the bare broadcast mechanism. The next section considers the real selection and communication mechanism of Field that relies on string messages and pattern matching.

Efficiency Problems. Such a broadcast mechanism suffers from a performance problem related to control integration. Since invocation is implicit, broadcasting a message is clearly slower than explicitly calling the callback procedures. Worse, the complexity of broadcast is linear in the number of registered events because the whole registration table must be scanned. This could be optimized with hash-tables for simple event identifiers, but not for a pattern-matching-based selection mechanism (see the next section), which would require a more complex automaton encoding.

Application of Partial Evaluation. The right-hand side of Figure 2 shows the optimization of registration and broadcast using partial evaluation. All indirect, implicit invocations of callback procedures have been turned into direct, explicit calls. Note that broadcasting an event like BUS_ERROR, for which no component has registered any interest, is turned into a "no-operation". Whereas the complexity of broadcast in the original program was linear in the number of registered events, the specialized program achieves broadcast in constant time, only depending on the event; at runtime, it is not necessary anymore to lookup the handler table.

The applicability of this optimization requires that the registered and broadcast events be known at specialization time. The example in Figure 2 illustrates compile-time specialization but a similar specialization can be done at run time, using a run-time specializer. Ad hoc user-aided specialization has already been considered for run-time compilation of event dispatch in extensible systems [8] but the approach is less automatic and less systematic.

As a by-product, if there is an application-dependent policy such that all broadcast messages should be received by at least one component (*i.e.*, no uncaught event), then inconsistencies between event registrations and broadcasts can be detected at specialization time. Assuming a `warning()` function is called in `broadcast()` whenever there is no registered receiver for a message, then partial evaluation replaces all occurrences of such void broadcasts by a warning invocation. Testing the above policy then only amounts to looking for `warning()` calls in the specialized program, which can easily be checked. In particular, this allows the detection of typos in registrations and broadcasts.

4.2 Optimizing Pattern Matching

The Mechanism. Selection of broadcast events may involve pattern matching rather than just comparison of event identifiers. In this case, when a message is broadcast, the system invokes all the procedures that are associated with registered patterns matching the message. In an environment like Field [33], a pattern identifies not only the type of the message but also the parts of the message that correspond to the arguments of the callback routine, and the format of those arguments. Pattern matching thus serves two purposes: selection of a message (string comparison) and, if matching succeeds, invocation of the callback routine with arguments decoded into the proper internal form.

In Field, patterns and messages exchanged by tools are all strings. The format of patterns is very similar to the Unix `scanf` facility. Basically, escape sequences for argument matching and decoding consist of a percent sign, an integer specifying the position of the argument in the callback routine and a type character: '`d`' for an integer, '`s`' for a string, etc. For example, after registering pattern `"DEBUG AT %2s line %1d"` with callback procedure `handle_debug_at`, broadcasting message `"DEBUG AT ./tree.c line 24"` eventually invokes `handle_debug_at(24,"./tree.c")`.

Brown University has given us access to the sources of the Field implementation. Because the pattern matching code is more than a thousand lines long, what we show on the left-hand side of Figure 3 is only a small representative excerpt. The top-left part of Figure 3 illustrates a typical call to the pattern matcher, with the pattern argument `"DEBUG AT %1s %2d"`. First, a pattern descriptor (a C structure) must be computed. It contains (among other things) the desired types and positions for the arguments to decode. For efficiency reasons, it also stores the length of the longest prefix of the pattern, that does not contain escape sequences. In our case, the length is 9, *i.e.* the size of `"DEBUG AT "`. It also converts the `scanf`-like pattern into a type-free pattern: `"DEBUG AT %A\001 %A\002"`. Then, actual pattern matching really starts: string comparison of the constant prefix with the message, string conversions of arguments according to escape sequences, and literal character comparison for embedded string constants. In the end, if the message actually matched, the callback routine is invoked with the decoded arguments.

Efficiency Problems. As stated by the author [33, p. 64], "All Field messages are passed as strings. While this introduces some inefficiencies, it greatly simplifies pattern matching and message decoding and eliminates machine dependencies like byte order and floating point representation." As patterns and messages are more complex, selection (*i.e.*, pattern matching) may become the bottleneck of broadcast. The phenomenon can be amplified if the complexity of the broadcast stays linear (see section §4.1). The efficiency problem here is a mixture of data integration (converting data back and forth to and from strings according to the given formats) and control integration (broadcast selection using pattern matching).

Application of Partial Evaluation. We have extracted the pattern matching routines from the Field implementation and run our partial evaluator on various pattern samples. In order to keep the original and specialized program small enough to fit in the paper, we only present in Figure 3 a simplified version of the code. For example, numbers are only read in decimal notation, not in octal nor hexadecimal. The right-hand side of Figure 3 shows the specialized pattern matcher obtained by partial evaluation (including inlining) from the general pattern sketched on the left-hand side.

```
my_execution_context(msg)
{
  p = PMATmake_pattern(
        "DEBUG AT %1s %2d", 2, NULL);
  process_message(msg, p, my_handler);
}
```

```
PMATmake_pattern(str,ct,defaults) { ... }

process_message(msg, pattern, handler)
{
  n = PMATmatch(msg, pattern, args);
  if (n >= 0)
    if (pattern->retargs == 2)
      (*handler)(args[0], args[1]);
}
PMATmatch(txt,pp,args)
{
  if (pp->prefix_len != 0)
    if (strncmp(txt,pp->pattern,
        pp->prefix_len) != 0) return -1;
  rslt = TRUE;
  txt += pp->prefix_len;
  for(p=pp->pattern+pp->prefix_len;*p!=0;++p){
    if (*p != '%')
      if (*txt++ != *p) rslt = FALSE;
    else {
      ++p;
      if (*p=='A') {
        i = (*++p) - 1;
        ap = (args == NULL) ? NULL : &args[i];
        if (!match_arg(&txt,&pp->arg[i],ap))
          rslt = FALSE; }
      else if (...)
        ...
    if (!rslt) break; }
}
match_arg(sp,pa,argp)
{
  s = *sp;
  if (pa->type==PMAT_TYPE_INT) {
    mode = 0;
    v = 0;
    while (TRUE) {
      if(!isdigit(*s)) break;
      v = v*base + *s++ - '0';
      mode = 1; }
    if (mode == 0) return FALSE; }
  else if (pa->type==PMAT_TYPE_STRING) {
    bufp = buf;
    len = 0;
    while (*s != 0 && !isspace(*s)) {
      if (len++ < MAX_ARG_SIZE) *bufp++ = *s;
      ++s; }
    *bufp = 0;
    v = (argp != NULL) ? strdup(buf) : 0; }
  else if (...)
    ...
  if (argp != NULL) *argp = v;
  return TRUE;
}
PMATfree_pattern(pp) { ... }
```

```
my_execution_context(msg)
{                       //-----Scan "DEBUG AT "
  if(strncmp(msg,"DEBUG AT %A\001 %A\002",9)){
    n = -1; goto end; }
  rslt = TRUE;
  msg += 9;
  bufp = buf;           //--------------Scan %s
  len = 0;
  while (*msg != 0 && !isspace(*msg)) {
    if (len++ < MAX_ARG_SIZE) *bufp++ = *msg;
    ++msg; }
  *bufp = 0;
  args[0] = strdup(buf);
  if (*msg++ != ' ') rslt=FALSE; //---Scan " "
  if (!rslt) goto stop;
  mode = 0;             //--------------Scan %d
  v = 0;
  while (TRUE) {
    if(!isdigit(*msg)) break;
    v = v*10 + *msg++ - '0';
    mode = 1; }
  if (mode == 0) rslt = FALSE;
  else args[1] = v;
stop:                   //---------------Match?
  if (!rslt) n = -1; else n = 2;
end:                    //-----------Invocation
  if (n >= 0)
    my_handler(args[0],args[1]);
}
```

Figure 3: Optimized Pattern Matcher

What must be noted is that the call to `PMATmake_pattern()` has been totally evaluated away: all pattern information has been inter-procedurally propagated and exploited so that the specialized program only performs the basic literal comparison and conversion operations. In terms of integration overhead, the optimization can be understood as follows. Because the type formats have been fused into control flow in the specialized pattern matcher, the data integration overhead now only reduces to string conversions. Moreover, control integration overhead is now restricted to raw pattern matching. Of course, this can be combined with the optimization of selective broadcast (see section §4.1).

In addition, as mentioned above, there exists a manual optimization in the original source code: the length of the constant prefix of the pattern is saved so that only a simple string comparison with the initial characters of the message is needed; then the full pattern matching machinery is set in motion. This situation burdens the code and the data structures; this is a drawback from a software engineering point of view. Yet, the same optimization could have been obtained automatically from the general pattern matching code via partial evaluation. That is in fact very general; programmers often do manual optimizations without always being aware that it corresponds to program specialization.

4.3 Tight Integration of Software Layers

The Mechanism. A layered system is a hierarchical organization of a program where each layer provides services to the layer above it and acts as a client to the layer below. The most widely known examples of this kind of architecture are layered communication protocols [25].

As an example of such an architecture, we have considered the Sun implementation of the remote procedure call (RPC), that makes a remote procedure look like a local one : the *client* transparently calls a function that is executed on a distant *server*. This protocol has become a *de facto* standard in the design and implementation of distributed services (NFS, NIS, etc.). It manages the encoding/decoding of data to a network-independent format, standardized by the eXternal Data Representation protocol (XDR). The user specifies the interface of the function, and "stub" routines are automatically generated for the client (encoding of arguments, emission, reception and decoding of result) and the server (reception and decoding of arguments, computation, encoding and emission of result), using generic RPC functions.

The Sun implementation is divided into many micro-layers, each one being devoted to a small task: generic client procedure call, selection of transport protocol (UDP, TCP, etc.), cases depending on scalars data size, choice between encoding and decoding, generic encoding/decoding (to/from memory, stream, etc.), reading/writing in network input/output buffers with overflow checks, selection between big and little endian. The left-hand side of Figure 4 shows the bottom of the stack of layers. As may be seen, the implementation is highly parameterized. For example, a function like `xdr_long` can achieves both encoding and decoding, depending on a flag provided in the arguments. A typical execution context for client encoding is displayed in the top-left corner. The `xdr_pair()` is a stub function that has been generated automatically; it encodes or decodes a pair of integers.

Efficiency Problems. Layered systems have several good properties: their design follows incremental abstraction steps; they favor extensibility and reuse; different implementations of the same layer can be interchanged. However, as noted in [34, p. 25], "considerations of performance may require closer coupling between logically high-level functions and their low-level implementation". This is precisely what partial evaluation achieves automatically.

More precisely, in our example, data integration is fixed by the protocol. On the other hand, control integration seems relatively important: invocations are all explicit, apart from the indirect call (through a function pointer) in `XDR_PUTLONG()`. However, invocation are numerous and exit statuses are propagated (and sometimes checked) through each micro-layers. Moreover, a dispatch function like `xdr_long()` does not really "produce" anything; it merely acts as a switch. In addition, output buffer is checked for overflow for each single integer encoding, rather than once and for all. All this introduces an important overhead.

Application of Partial Evaluation. Yet, the information driving the switch in `xdr_long()` and the number of integer written in the output buffer can be known from the execution context. Consequently, the exit status of the inner-most layer can be known (buffer overflow or not). Propagating this information to each layers makes the tests unnecessary.

```
my_execution_context()
{
  ...
  xargs = xdr_pair;  // arguments encoding
  xdrs->x_ops->x_putlong = xdrmem_putlong;
  xdrs->x_op = XDR_ENCODE;
  if(!(*xargs)(xdrs,argsp))
    return cu->cu_error.re_status;
  sendto(...);
  ...
}
```

```
xdr_pair(xdrs,objp)      //----------User generated
{
  if (!xdr_int(xdrs,&objp->int1)) {
    return (FALSE);
  }
  if (!xdr_int(xdrs,&objp->int2)) {
    return (FALSE);
  }
  return (TRUE);
}
xdr_int(xdrs,ip)         //------Read/write integer
{
  if (sizeof(int) == sizeof(long)) {
    return xdr_long(xdrs,(long *)ip);
  else
    return xdr_short(xdrs,(short *)ip);
}
xdr_long(xdrs,lp)        //---------Read/write long
{
  if( xdrs->x_op == XDR_ENCODE )
        return XDR_PUTLONG(xdrs,lp);
  if( xdrs->x_op == XDR_DECODE )
        return XDR_GETLONG(xdrs,lp);
  if( xdrs->x_op == XDR_FREE )
        return TRUE;
  return FALSE;
}
#define XDR_PUTLONG(xdrs, longp) \
   (*(xdrs)->x_ops->x_putlong)(xdrs,longp)

xdrmem_putlong(xdrs,lp) //----Write long to memory
{
  if((xdrs->x_handy -= sizeof(long)) < 0)
    return FALSE;
  *(xdrs->x_private) = htonl(*lp);  // buffer copy
  xdrs->x_private += sizeof(long);  // ptr increm
  return TRUE;
}
#define htonl(x) x
```

```
my_execution_context()
{
  ...
  *(xdrs->x_private) = objp->int1;
  xdrs->x_private += 4u;
  *(xdrs->x_private) = objp->int2;
  xdrs->x_private += 4u;
  sendto(...);
  ...
}
```

Figure 4: Tight Integration of Micro-Layers

```
(rec-msg(calc(Exp)) . snd-eval(run(Exp)) .
 rec-value(Val) . snd-do(prn(Val))) * delta
```

```
void interp(pe_t *pe, rterm_t *(*eval)(term_t*))
{
  if (pe->op == CHOICE) {
    if (try(pe->e1))
      interp(pe->e1,eval);
    else
      interp(pe->e2,eval);
  } else if (pe->op == SEQUENCE) {
    interp(pe->e1,eval);
    interp(pe->e2,eval);
  } else if (pe->op == ITERATION) {
    ok=try(pe->e1);
    while (ok) {
      interp(pe->e1,eval);
      ok=try(pe->e1);
    }
    interp(pe->e2,eval);
  } else if (pe->op == ATOMIC) {
    if (pe->atomic->action == SND_MSG) {
      SB_prod_term(pe->atomic->term);
      SB_snd_msg(pe->atomic->term);
    } else if (pe->atomic->action == REC_MSG) {
      ok=SB_cons_term(pe->atomic->term,
                      SB_rec_msg());
      if (!ok) fail();
    } else if (pe->atomic->action == SND_EVAL) {
      SB_prod_term(pe->atomic->term);
      result=(*eval)(pe->atomic->term);
    } else if (pe->atomic->action == REC_VALUE) {
      ok=SB_cons_term(pe->atomic->term,result);
      if (!ok) fail();
    } else if (pe->atomic->action == SND_DO) {
      SB_prod_term(pe->atomic->term);
      (*eval)(pe->atomic->term);
    }
  }
}
```

```
void interp(pe)
{
  ok = try(pe->e1);
  while (ok) {
    petmp1 = pe->e1;
    petmp2 = petmp1->e1;
    SB_cons_term(petmp2->atomic->term,
                 SB_rec_msg());
    petmp3 = petmp1->e2->e1;
    SB_prod_term(petmp3->atomic->term);
    result=handle(petmp3->atomic->term);
    petmp4 = petmp2->e2->e1;
    SB_cons_term(petmp4->atomic->term,
                 result);
    petmp4 = petmp3->e2;
    SB_prod_term(petmp4->atomic->term);
    handle(petmp4->atomic->term);
    ok = try(pe->e1);
  }
}
```

Figure 5: Interpreter Optimization

Right-hand side of Figure 4 shows what partial evaluation does automatically on such an architecture. Note that the dispatches, the propagation of exit status and the buffer overflow checking have all been removed. As a matter of fact, benchmarks have shown [27] that the specialized code of RPC encoding routines is up to 3.75 times faster.

4.4 Compiling Coordination Languages Interpretation

The Mechanism. *Scripting languages* [30] are intended to glue together a set of powerful components (building blocks) written in traditional system programming languages. Scripting languages simplify connections between components and provide rapid application development. Coordination [9] and domain specific languages [44] exploit the same idea. The Toolbus coordination architecture [5] uses this concept. It consist of independent tools (seen as processes) communicating via messages. However, communication of messages is not performed by the tools; it is carried out by a script that coordinates the processes. Toolbus also relies on the selective broadcast mechanism (see section §4.1) and pattern matching (see section §4.2); messages are tree-like terms and patterns are terms with variables.

The top-left part of Figure 5 shows a sample script written in *T Script*, the script language of Toolbus. As described in [5], a T script consist of a composite process formed from builtin atomic processes. The atomic processes are combined using choice (`+`), sequence (`.`), and iteration (`*`). Atomic rules take terms as arguments. Terms are constructed from lower case literal identifiers and capitalized variable names. Each script is associated with a tool and evaluates terms of the atomic processes `snd-eval` and `snd-do` by calling an evaluation function for that tool. The sample script specifies a simple calculator, which receives expressions from other tools, evaluates them and then prints their results. The evaluation function for the calculator treats terms of the form `calc(`X`)` by evaluating the expression specified by the term X and evaluates terms of the form `prn(`X`)` by printing the term X. The script consists of an iteration of the four atomic processes `rec-msg`, `snd-eval`, `rec-value`, and `snd-do`, which respectively

1. wait for a message and match it to the term `calc(Exp)`,
2. build the term `run(Exp)` and pass it to the evaluation function for the the tool,
3. place the return value of the previous `snd-eval` into the `Val` variable, and
4. build the term `prn(Val)` and pass it the evaluation function.

Iteration continues as long as the `rec-msg` continues to succeed (*i.e.*, there are messages that match the term `calc(Exp)`).

The left-hand column of Figure 5 shows the core of an interpreter for T scripts. The interpreter accepts a script in abstract syntax form and traverses the tree executing each construct. The process operators `+` and `*` use a "try" function in order to predetermine if a process expression will fail. This is used, for example, to determine when to terminate iteration. The atomic processes are implemented with some basic functions: `SB_cons_term()` matches a message to a term and assigns values to variables in the terms; `SB_prod_term()` expands variables in a term with their values; `SB_snd_msg()` send a message and `SB_rec_msg()` receives one.

Efficiency Problems. Most often, scripts are interpreted and type-less. This provides more flexibility to the gluing language. However, that introduces performance overhead that become significant when the building blocks are small. As stated in [5, p. 82] "There are many methods for implementing the interpretation of T scripts, ranging from purely interpretative methods to fully compilational methods that first transform the T script into a transition table. The former are easier to implement, the latter are more efficient. For ease of experimentation we have opted for the former approach." The interpretation overhead is actually due to a poor control integration. Interpreting the script leads to a significant latency in communications.

Application of Partial Evaluation. The right-hand column of Figure 5 demonstrates how partial evaluation successfully eliminates the interpretation, producing a program similar to what one would write by hand to implement the example script. A C-structure representation of the above script (argument for `pe`) and the evaluation function `handle` (argument for `eval`) yields a specialized program with one while loop resulting from the `*` iteration construct; its body consists of the implementation of the four atomic processes used in the script. Basically, the script has been compiled by partial evaluation. For clarity, the definition of functions `SB_`xxx were not specialized. However, since the `SB_cons_term()` and `SB_prod_term()` functions consist of basic pattern matching, partial evaluation could be applied to them in a similar manner as in section §4.2.

Partial evaluation has already be advocated as a general tool to help building domain specific languages software environment [38]. Its application to interpreters has also been extensively studied [21]. In fact, constructing compilers from interpreters in one of the standard use of partial evaluation.

4.5 Efficient Instances of Generic Libraries

The Mechanism. General libraries like libg++, NIHCL, COOL, or the Booch C++ Components [7] have had a large success in achieving reuse. However, for performance reasons, they implement a large number of hand-written specific components that represent a unique combination of features (*e.g.* concurrency, data structure, memory allocation algorithms). As a consequence, the library implementation itself achieves little reuse. It has been argued that this way of building data structure component libraries is inherently unscalable. Another approach is to provide only primitive building blocks and have a generator combine these blocks to yield complex custom components [4]. However, the techniques and the generator are not general purpose. In some cases, computer algebras may also automatically

```
my_execution_context()
{
  norm = v_get(3);
  light = v_get(3);
  n_dot_l = _in_prod(norm,light,0);
}
```

```
double _in_prod(VEC *a, VEC *b, u_int i0)
{
  if ( a==(VEC *)NULL || b==(VEC *)NULL )
    error(E_NULL, "_in_prod");
  limit = min(a->dim, b->dim);
  if ( i0 > limit )
    error(E_BOUNDS, "_in_prod");
  return __ip__(a->ve+i0, b->ve+i0,
                (int)(limit-i0));
}

double __ip__(Real *dp1, Real *dp2, int len)
{
  sum = 0;
  for( i = 0; i < len; i++ )
    sum += dp1[i]*dp2[i];
  return sum;
}
```

```
my_execution_context()
{
  ...
  n_dot_l = norm->ve[0] * light->ve[0] +
            norm->ve[1] * light->ve[1] +
            norm->ve[2] * light->ve[2];
}
```

Figure 6: Optimization of a Call to a Maths Library Function

generate parts of libraries from given mathematical models. However, that is very restricted and specific to a model and a computer algebra system.

We have taken as an example the Meschach Library [37] developed at the Australian National University, which provides a wide range of matrix computation facilities. It it very general in its design and implementation. For example, many functionalities in Meschach are implemented using two routines. The first one provides a clean interface; it controls the validity of arguments and performs bound checking. The second one does the actual computation on raw data. Such an example is shown in Figure 6: function `_in_prod()` provides the safe encapsulation to function `__ip__()`. The top-left corner gives an example use of the library: two three-dimension vectors are allocated and used for a inner-product operation.

Efficiency Problems. It is clear that the software protection provided by the `_in_prod()` interface function is achieved at the expense of performance loss. Moreover, because the function may apply to vectors of any size, inner-product computation involves a loop management overhead. In terms of control integration, the communication between the caller and the library function seems explicit. However, only the invocation of `__ip__()`, that performs the actual computation, is significant. Communication rather must be considered as implicit. The components need tighter integration.

Application of Partial Evaluation. As may be seen in Figure 6, partial evaluation uses available information (*i.e.*, the size of the vectors) to eliminate all verifications concerning the validity of the arguments: the safety interface layer is compiled away. That is analogous to the elimination of buffer overflow checking in the RPC experiment. In addition, the raw computation itself is here slightly improved using loop unrolling. When an application heavily relies on a general library, such optimizations become crucial.

5 Conclusion

As discussed in this paper, the literature of software architectures presents many approaches which, according to their authors, trade efficiency for flexibility. The reason why flexibility introduces overhead is that genericity and adaptability are not only present at the design level but also in the implementation.

We have identified the fundamental efficiency problems in flexible architectures as being related to data and control integration of software components. We have proposed to use a systematic and automatic program transformation (*i.e.*, partial evaluation) to turn flexible implementations into efficient ones while retaining flexibility at the structuring level. In order to assess our claim, we have studied five common mechanisms used in software architectures (selective broadcast, pattern matching, interpreters, layers, and generic libraries) and successfully applied partial evaluation to them, yielding efficient implementations. Because this optimization can also be performed at run time, depending on run-time values, flexibility is not constrained to compile-time structuring.

While standard partial evaluation suits control integration very well, it does little concerning data integration. A more complex partial evaluation technique, known as *deforestation* [42], is needed to combine successive data conversions. However, to our knowledge, it has not been applied yet to imperative programming. Semi-automatic approaches to copy elimination in inter-layer communications have been considered [41] but not yet put into practice. Because specialization needs *actual values*, there is also a limit to the type of control and software protection overhead that partial evaluation can eliminate. In particular, traditional partial evaluation cannot exploit *properties about values*, such as interval ranges. Several extensions to partial evaluation exploiting properties have been proposed: *parameterized partial evaluation* [12] and *generalized partial computation* [17]. However, they have not been yet put into practice on realistic applications.

References

[1] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*, chapter 5, pages 111–119. Software Series. Prentice Hall, 1986.

[3] Don Batory. Intelligent components and software generators. In *Proceedings of the Software Quality Institute Symposium on Software Reliability, Austin, Texas*, April 1997. Also available as Technical Report 97-06, Department of Computer Sciences, University of Texas at Austin, February 1997.

[4] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable Software Libraries. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 191–199, December 1993.

[5] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In Ciancarini and Hankin [9], pages 75–88.

[6] 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 [36], pages 267–283.

[7] Grady Booch. The design of the C++ booch components. *ACM SIGPLAN Notices*, 25(10):1–11, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[8] C. Chambers, S. Eggers, J. Auslander, M. Philipose, M. Mok, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In WCSSS'96 [43], pages 118–126.

[9] Paolo Ciancarini and Chris Hankin, editors. *Coordination and models, Proceedings of the first international conference, Cesena, Italy*, number 1061 in LNCS. Springer Verlag, 1996.

[10] 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.

[11] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [15], pages 54–72.

[12] C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993. Extended version of [13].

[13] C. Consel and S.C. Khoo. Parameterized partial evaluation. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 92–106, Toronto, Ontario, Canada, June 1991. ACM SIGPLAN Notices, 26(6).

[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] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, February 1996.

[16] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [36], pages 251–266.

[17] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *J. Theoretical Computer Science*, 90:60–79, 1991.

[18] D. Garlan, G.E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE journal Computer*, 25(6):30–38, June 1992.

[19] ISO. Specification of abstract syntax notation one (ASN.1). ISO standard 8824, 1988.

[20] Ian Jacobs, Janet Bertot, Francis Montagnac, and Dominique Clement. The SOPHTALK reference manual. Technical Report RT 150, INRIA, February 1993.

[21] N.D. Jones. What *not* to do when writing an interpreter for specialisation. In Danvy et al. [15], pages 216–237.

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

[23] T.B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225. ACM SIGPLAN Notices, 31(5), May 1996. Also TR MSR-TR-96-04, Microsoft Research, February 1996.

[24] 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.

[25] G.R. McClain. *Open Systems Interconnection Handbook*. Intertext Publications, McGraw-Hill, New York, 1991.

[26] 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.

[27] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[28] H. Penny Nii. Blackboard systems. *AI Magazine*, 7(3):38–53, 1986.

[29] OMG. *CORBA: The Common Object Request Broker: Architecture and Specification*. Framingham, 1995.

[30] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. White paper, Sun Labs, 1997. Available at `http://www.sunlabs.com/people/john.ousterhout/`.

[31] 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 [36], pages 314–324.

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

[33] Steve P. Reiss. Connecting tools using message passing in the filed environment. *IEEE Software*, 7(4):57–66, July 1990.

[34] M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996.

[35] R. Snodgrass. *The Interface Definition Language: Definition and Use*. Computer Science Press, Rockville, MD, 1989.

[36] *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.

[37] D. R. Stewart. *MESHCHAC: Matrix Computations in C*. University of Canberra, Australia, 1992. Documentation of MESCHACH Version 1.0.

[38] S. Thibault and C. Consel. A framework of application generator design. In *Proceedings of the Symposium on Software Reusability*, May 1996.

[39] E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, Atlanta, USA, October 1997. ACM Press. To appear.

[40] E.N. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: the RPC case study. In WCSSS'96 [43], pages 24–28.

[41] E.N. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noyé, and C. Pu. A uniform automatic approach to copy elimination in system extensions via program specialization. Research Report 2903, INRIA, Rennes, France, June 1996.

[42] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 90.

[43] *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, Tucson, AZ, USA, February 1996.

[44] *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.

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, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
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)
http://www.inria.fr
ISSN 0249-6399