Automated Software Engineering, , 1–30 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Efficient Implementations of Software Architectures via Partial Evaluation

RENAUD MARLET, SCOTT THIBAULT AND CHARLES CONSEL

{marlet,consel}@irisa.fr thibault@gmvhdl.com

Compose group
IRISA / INRIA - Université de Rennes 1
Campus universitaire de Beaulieu, 35042 Rennes cedex, FRANCE

Editor:

Abstract. The notion of flexibility (that is, the ability to adapt to changing requirements or execution contexts) is recognized as a key concern in structuring software, and many architectures have been designed to that effect. However, the corresponding implementations often come with performance and code size overheads. The source of inefficiency can be identified to be in the loose integration of components, because flexibility is often present not only at the design level but also in the implementation.

To solve this flexibility *vs* efficiency dilemma, we advocate the use of partial evaluation, which is an automated technique to produce efficient, specialized instances of generic programs. As supporting case studies, we consider several flexible mechanisms commonly found in software architectures: selective broadcast, pattern matching, interpreters, software layers, and generic libraries. Using Tempo, our specializer for C, we show how partial evaluation can safely optimize implementations of those mechanisms. Because this optimization is automatic, it preserves the original genericity and extensibility of the implementation.

Keywords: software architectures, partial evaluation, program specialization, genericity, extensibility, adaptability, selective broadcast, pattern matching, interpreters, software layers

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 because of its impact on the cost of development, validation and maintenance. Because this choice also affects the extensibility and interoperability of systems, it can have large impacts on the time from conception to product release, providing a competitive advantage or disadvantage.

1.1. The flexibility issue

Complex software systems are characterized by their changing nature: computations may be distributed over a network of heterogeneous machines and components, where tasks can migrate at run time; connections between software components can evolve in time; computations may be intensive or scattered in time; hardware platforms offer vastly different functionalities, performance and resource constraints;

software environments provide applications with changing services; etc. This calls for programs that are able to adapt to changing requirements and execution contexts.

The *flexibility* of an application (or application family) reflects its power of adaptation with respect to variable needs. Three "actors" typically express such needs: the software vendor sells various applications within a product line (or a single, evolving product), varying the set of available platforms (hardware, operating systems, etc.) and features; the user configures or parameterizes a given application depending on specific needs; the environment (CPU, operating system, network, etc.) imposes variable resource constraints to the application as it runs.

The degree of adaptation offered to each of those actors is ideally defined at specification time. Given these requirements, the choice of a software architecture then determines the general framework of a design-time flexibility. The implementation steps that follow take advantage of this flexibility and can further restrict it. The later the adaptation choices are in the development stages, the more flexible the system.

1.2. Flexible software architectures

When architecturing software, common instances of flexibility include extensibility, portability, interoperability, re-usability, modularity, abstraction, genericity, parameterization and configurability, as well as safety, fault tolerance and quality of service. Depending on the granularity used for reasoning, flexibility typically comes in two flavors that usually coexist: it can be provided by the generality of individual components or by the richness of the mechanisms for composing components.

Many approaches aimed at achieving software flexibility have been proposed and put into practice, including pipes and filters [3], layered systems [47], data abstraction and object-oriented organization, event-based communication [37, 62], software buses [52], coordination languages [12], and domain-specific languages [26, 80].

Using flexible software architectures reduces the correlation of components and favors sharing, hence also reduces complexity. Flexibility is therefore a key feature when developing software: it leads to better a cost, time and quality of development, validation, maintenance and evolution.

1.3. The flexibility vs efficiency dilemma

Flexibility is required at adaptation time, whereas efficiency is only required at program-execution time. The problem is that flexibility impairs efficiency when it is not only present at the design level but also in the implementation: in this case, some computations are devoted to adaptation as opposed to what the application is supposed to produce.

For example, in a generic component, some amount of execution time is spent in following decision trees and indirections that correspond to parameterization choices. Because many cases are considered, this is less efficient than using a specific component that only provides the required service for a given execution

context. Likewise, concerning communication mechanisms (what glues components together), some amount of execution time is spent in traversing software connectors rather than spent in the components themselves.

Flexibility and efficiency necessitate a compromise. Our strategy is to express flexibility at the implementation level and to remove it systematically and automatically by program transformation to improve performance.

1.4. This article

Our contributions are the following.

- We characterize the fundamental reasons why implementing the mechanisms of flexible software architectures can lead to inefficiency problems.
- We advocate the use of partial evaluation, a systematic and automatic optimization technique, to build efficient implementations of software architectures without compromising flexibility.
- To support our stand, we consider five representative instances of mechanisms used in software architectures, namely selective broadcast, pattern matching, interpreters, software layers, and generic libraries. For each of them we show how partial evaluation indeed improves efficiency while retaining flexibility.

The rest of the article is organized as follows. Section 2 characterizes the general sources of inefficiency in the implementation of flexible software architectures. Section 3 introduces partial evaluation. Section 4 considers in turn several mechanisms used in flexible software architectures and the application of partial evaluation. Section 5 discusses the general applicability, predictability and automation of partial evaluation. Section 6 compares it to other existing techniques. Section 7 concludes and gives some research directions for further improvements.

2. Sources of inefficiency when implementing software architectures

Flexibility and efficiency depend on the way software components interact: (i) what data they exchange, and (ii) how they communicate. After examining these issues in turn, this section elaborates on the flexibility *vs* efficiency dilemma.

2.1. Data integration

Software systems are made of components that exchange or share data. These components might not use the same data representation (data structure, layout in memory, unit of measure, etc.). Such a situation occurs often, for example when an existing software component is being reused in a different context, when components are programmed using different languages, or when components run on different systems or hardware platforms in a distributed environment.

Data communication between heterogeneous components requires conversions. Two main approaches are being used. One is to systematically convert component-

specific data into a universal format that is used in all inter-component communications. As a consequence, each data communication necessitates two conversions. The universal format may be provided by an intermediate data description languages such as ASN.1 [36] or IDL [66]. Pushing all conversions into the callee is another solution. The called component examines a format 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 of formats. Extensibility is reduced because adding a new data format requires writing many new converters.

Data integration is measured by the amount of computations devoted to producing actual (observable) results compared to computations aimed at managing data communications between components. A tighter data integration makes data representation more uniform across components so that less conversions are needed.

Another instance of the data integration problem is the verification of data well-formedness. When a component does not trust another component, it must check the validity of input data. Type checkers already guarantee some static constraints at compile time. But when the constraints are too complex to be expressed in a type system, or when the implementation language is dynamically typed, the component must resort to explicit, dynamic checking. Besides typing, well-formedness verification includes protection against null pointer dereference and buffer overflow. This can be enforced by the language itself (*e.g.*, systematic array bounds checking in Java). It is clear that verifying safety assertions at run time impairs efficiency. Given a configuration that combines two components, a tighter data integration removes assertion checking when it can be proved that communication will always involve legal data. It is common practice though to turn on assertion-checking only during development. It is turned off in production, for programs with low safety requirements, that have no error recovery policy.

2.2. Control integration

Besides unifying data formats and ensuring data well-formedness and integrity, composing software components also involves strategies to make these components communicate.

Control integration is measured by the amount of computations devoted to producing actual (observable) results compared to the computations needed to invoke services between components (or choosing a service inside a component).

In systems in which the interface of components consists of a collection of routines, communication is based *explicit invocation*, *i.e.* procedure call. Explicit invocation is fast but not very flexible: the exact name of the routines to call must be known at compile time. *Implicit invocation* refers to communications where the called procedure can depend on run-time values. For example, in an object-oriented system, invocation of virtual methods is textually explicit but actually involves an additional object-dispatch indirection. Similarly, broadcasting a message is an implicit invocation of procedures in other components. Implicit invocation is slower than explicit invocation because it introduces an additional dispatch layer. That is why,

e.g., compilers for object-oriented languages try to turn virtual calls into explicit calls. However, implicit mechanisms offer more support for extensibility.

In some sense, generic components can also contain aspects of implicit invocation at a finer grain where parameters of the component are used to select various behaviors. Although general and extensible, parameterization impairs efficiency because execution time is spent in testing options, as opposed to actually providing the required service.

Control integration also has a significant impact on code size. In fact, adaptability calls for the anticipation of many contexts of usage. Given a specific 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. Because of limitations of memory or network bandwidth, the size issue is important for embedded systems and mobile code.

2.3. Improving efficiency while retaining flexibility

For greater efficiency, the integration of data and control that is expressed in the architecture should be made tighter in the implementation: the number of conversions should be reduced; safety and security checks should be removed when unnecessary; implicit control should be turned into explicit control; generic components should be adapted to specific uses.

Also, flexibility might actually be used at different stages of the assembly of a software system. Therefore, gaining efficiency in the implementation may occur at different times: configuration time, compile time, link time, load time, and run time. In practice, the later the adaptation, the more difficult to implement it efficiently.

A central idea in optimizing component integration is *specialization*. (Other strategies are listed in Section 6.) The effects of specialization range from customizing the connection between components to the complete merging of the components' functionalities. Specialization is detailed in the next section.

3. Program specialization and partial evaluation

Program specialization is a program transformation that adapts programs with respect to known information about their inputs. We first give a general overview of specialization. Then, we focus on partial evaluation, a process that automates specialization, and introduce Tempo, a partial evaluator for C.

3.1. Specialization in a nut shell

3.1.1. Principles. Let us consider a program p, taking some argument data d and producing a result r: $exec\, p(d) = r$. If the inputs d may be split into $d = (d_1, d_2)$ where d_1 is a *known* subset of the inputs (*i.e.*, it does not vary) and d_2 is yet *unknown*, we may form a new program p_{d_1}, equivalent to p, where computations depending on d_1 have been exploited: $exec\, p_{d_1}(d_2) = exec\, p(d_1, d_2) = r$. The

```
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  : prterror(*fmt); return;
      }
    fmt++;
  }
}                              /* Legend: Known Unknown */          1a. Original
```

```
mini_printf_fmt(int val[])
{
  putchar('<');
  putint(val[0]);
  putchar(',');
  putint(val[1]);
  putchar('>');
}                                                                  1b. Specialized
```

Figure 1. Specialization of `mini_printf()` with respect to `fmt = "<%d,%d>"`

program p_{d_1} is called a *specialization* of p with respect to the *invariant* d_1. Known (resp. unknown) input is also called *static* (resp. *dynamic*).

More generally, specialization exploits any invariant present in the code, not only input values but also embedded constants. The idea is to factor out computations from the specialized program.

3.1.2. Example. Let us consider the simple example shown in Figure 1. At the top of the figure is the definition of `mini_printf()`, a simplified version of the C `printf()` text formatting function. At the bottom of the figure is a specialized version of `mini_printf()` with respect to the format string `fmt = "<%d,%d>"`. Note that all computations depending on `fmt` (*i.e.*, the interpretation of the 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 only rely on the known partial input (here, `fmt`) or on embedded constants (here, the initial `0` value for variable `i`). They disappear in the specialized program.

3.1.3. 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. If building the specialized program p_{d_1} comes at a certain price though (in particular when it is performed at

run time), it is worth it only if $p_{d_1}(d_2)$ is run enough times to amortize the cost of building p_{d_1}. In addition, cases written to treat other values than d_1 can be removed from p_{d_1}: they are dead code. In this case, program p_{d_1} is *smaller*.

On the other hand, specialization can also involve loop unrolling, which may increase code size. For example, in Figure 1b, the whole loop has been unrolled; if the format string had been longer, many `putchar()` calls would have appeared in the specialized program. The possible drawback of loop unrolling may be prevented while retaining most of the specialization advantages using a technique known as *data specialization* [10, 44].

There already exists tools that provide some primitive support for a kind of 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 optimizations. A more thorough comparison with other existing techniques can be found in section 6.

3.2. Partial evaluation

Partial evaluation is a technique that *automates* the specialization process [14, 39]. Partial evaluation is also *systematic*, as opposed to *ad hoc* specializations that are restricted to specific cases. Using the same notations as in Section 3.1, a *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 also reduce code size.

Roughly speaking, standard partial evaluation can be thought of as a combination of aggressive *inter-procedural* constant propagation (not only applied to scalars but to *all* data types, including pointers, arrays and structures), constant folding, and loop unrolling. It may be followed by procedure inlining and some algebraic simplifications. Those transformations are beyond the scope of optimizing compilers.

Most of the flexibility (whether genericity or extensibility) of the original code is lost in this optimization process. In contrast with manual specialization, partial evaluation is safe. But the key point is that the original code stays unaffected and can be later modified and re-specialized. Because partial 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, and scales up to large programs. Also, because the program analyses that are part of the partial evaluation process determine the portions of the code that can be specialized, partial evaluation is predictable,

Long confined to 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 the basis of the development of adaptable systems in a number of major research projects and in different areas such as networking [48, 76], graphics [44], and operating systems [7, 29, 60].

3.3. Tempo, a partial evaluator for C programs

Our claim concerning the applicability of partial evaluation to software engineering is not specific to a language or a partial evaluator. However, it is backed up by experiments using a specific tool. In the following, we use *Tempo*, a partial evaluator for C programs developed in our group[1] [15, 16]. Tempo has been applied to sizable and realistic programs such the Sun RPC [49]. Tempo and its documentation are publicly available[2]. It is being used by more than twenty people around the world, including non-partial evaluation expert. It is also being used as a front end for specializing Java [63].

There exists another system for automatic C specialization named *C-Mix* [1], whose analyses are less accurate and which does not support run-time code generation (see [50] for comparison details).

Tempo is an *off-line* specializer [14]: partial evaluation is split into two phases. First, a preprocessing phase performs an abstract propagation of all known information throughout the code. This phase is known as the *binding-time analysis* (BTA). It partitions the program into two computational stages (static and dynamic): for each program construct, the BTA determines whether it can be evaluated early (*i.e.*, at specialization-time) or must be evaluated late. The output of this analysis can be visualized as pretty-printed code with color information, in a form which is very similar to the font decoration in Figure 1a. The user can thus assess the benefits of applying partial evaluation. Second, a processing phase performs actual specialization (*i.e.*, code generation), given some partial input values. An additional postprocessing phase performs inlining and some algebraic simplifications.

Tempo can exploit values when they are known, whether at compile time or at run time [20]. Compile-time specialization is a source-to-source program transformation, whereas run-time specialization directly produces binary code. To achieve this, the information gathered in the BTA is used to determine a grammar of all possible specializations (for all possible known input). Templates of code that correspond to the building blocks of the specialized code are thus identified and precompiled. Then, run-time specialization merely amounts to assembling precompiled templates and filling holes in those templates with computed values. Binary template assembly provides fast code generation [51]. Run-time code generation is an important feature because it does not limit us to static configurations, *i.e.*, compile-time architectures.

The following section illustrates applications of Tempo to optimize various software architecture mechanisms.

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 minor transformations like copy propagation (ordinarily performed by optimizing compilers) were done by hand on the specialized source; some irrelevant parts of the code have been omitted, as well as some type and variable definitions. In addition, code has been manually pretty-printed to fit in the figures and some comments have been added.

All partial evaluation examples displayed in this section are compile-time source-to-source program transformations, as opposed to run-time code generation. This is for obvious readability reasons: when specialization values are known at run time, and even vary during program execution, Tempo can generate binary specialized routines on the fly that could only be displayed in assembly format.

4.1. Optimizing selective broadcast

Our first case study deals with *selective broadcast*, also called *reactive integration* [64]. In such an architecture, components are independent agents that interact with each other by sending broadcast events. Components in the system that are interested in particular messages register *callback* procedures to be called each time such messages are broadcast. This mechanism is also called *implicit invocation* because broadcasting events "implicitly" invokes procedures in other components. Blackboard techniques may also be based on similar indirect access mechanisms [32].

4.1.1. The mechanism. The Field programming environment is a representative example of such an architecture [62]. It is an open system that integrates many programming tools. Let us consider a system containing an editor, a debugger and control flow graph viewer. The example in Figure 2b models—the actual implementation of Field is more complex—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 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 node of the current function in the graph.

In order to properly separate concepts, events are identified here using an integer, and data associated to events is a structure pointer (manipulated as a "dummy" character pointer). This only models the bare broadcast mechanism (see Figure 2a). Section 4.2 considers the real selection and communication mechanism of Field that relies on string messages and pattern matching for tool integration.

4.1.2. 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.

```
register_for_event(int event, void (*func)(char*))
{
  handler[nb_handlers].func = func;              // Record callback function
  handler[nb_handlers].event = event;            // Record event id
  nb_handlers++;                                 // Adjust nb of handlers
}
broadcast(int event, char *arg)
{
  for (i = 0; i < nb_handlers; i++)              // For each registration
    if (handler[i].event == event)               // Look for registered event
      (*handler[i].func)(arg);                   // Run callback accordingly
}
```

2a. Mechanism

```
{
  register_for_event( DEBUG_AT, editor_goto );   // Set callbacks for
  register_for_event( DEBUG_AT, cfg_highlight ); //    event DEBUG_AT
  ...
  debug_info->func_line = line;                  // Group some arguments
  debug_info->func_name = fname;                 //    into a single structure
  broadcast( DEBUG_AT, (char *)debug_info );     // Emit event DEBUG_AT
  broadcast( BUS_ERROR, (char *)NULL );          // Emit event BUS_ERROR
}
```

2b. Example of use

```
{
  ...
  debug_info->func_line = line;                  // Group some arguments
  debug_info->func_name = fname;                 //    into a single struct
  editor_goto( (char*)debug_info );              // Invoke editor_goto
  cfg_highlight( (char*)debug_info );            // Invoke cfg_highlight
}
```

2c. Specialization

Figure 2. Registration and broadcast

Worse, the complexity of a broadcast is linear in the number of registered events because the whole registration table must be scanned in order to find, among all registrations, the callbacks that are registered for the given event. This could somehow be optimized with an array or a hash-table for simple event identifiers, but not for a pattern-matching-based selection mechanism (see Section 4.2), which would require a much more complex automaton encoding.

4.1.3. Application of partial evaluation. The static configuration here consists in event identifier that appear in the code. Figure 2c 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 "no operation". Whereas the complexity of a broadcast in the original program is linear in the number of registered events (*i.e.*, `nb_handlers`), the specialized program achieves broadcast in constant time: all functions registered for the given event are known and hard-coded; at run time, it is no longer necessary to look in the handler table.

The applicability of this optimization requires that the registered and broadcast events be known at specialization time. The example in Figure 2c illustrates compile-time specialization but a similar specialization can be done at run time, using a run-time specializer. User-aided specialization has already been considered for run-time compilation of event dispatch in extensible systems [9] but the approach is less automatic and less systematic.

As a by-product of partial evaluation, 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 the body of `broadcast()` whenever there is no registered receiver for a message to broadcast, then partial evaluation replaces all occurrences of such void broadcasts by calls to the warning function. Then, testing the above policy only amounts to looking for calls to the warning function in the specialized program, which can easily be checked. In particular, this process allows the detection of typos in registrations and broadcasts.

Another instance of implicit invocation (another name for selective broadcast) is virtual method invocation in object-oriented languages. The elimination of virtual calls (*i.e.*, transformation into an explicit call), as obtained after class hierarchy analysis [25], can also be achieved using partial evaluation [41].

4.2. Optimizing pattern matching

Selection of callback procedures to execute 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 [62], a pattern not only identifies 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 there is a match, invocation of the callback routine with arguments decoded into the proper internal format.

4.2.1. The mechanism. 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 the pattern `"DEBUG AT %2s line %1d"` with callback procedure `handle_debug_at`, broadcasting the message `"DEBUG AT ./tree.c line 24"` eventually invokes the function with the call `handle_debug_at(24, "./tree.c")`.

Because the original pattern matching code of Field (implemented at Brown University) is more than a thousand lines long, what we show in Figure 3a is only

```
PMATmake_pattern(str, ct, defaults) { ... }     // Make pattern descriptor

process_message(msg, pattern, handler)          // Call handler if pattern
{                                               //    matched text
  n = PMATmatch(msg, pattern, args);            // Match msg against pattern
  if (n >= 0)                                   // n = -1 means failure
    if (pattern->retargs == 2)                  // If two arguments were read
      (*handler)(args[0], args[1]);             // Callback invocation
}
PMATmatch(txt, pp, args)
{
  if (pp->prefix_len != 0)                      // Direct cmp for const prefix
    if (strncmp(txt,pp->pattern,pp->prefix_len) != 0) return -1;
  rslt = TRUE;                                  // Reset matching success flag
  txt += pp->prefix_len;                        // Skip constant prefix
  for (p = pp->pattern + pp->prefix_len; *p != 0; ++p) { // Scan pattern
    if (*p != '%')                              // Match non-escape character
      if (*txt++ != *p) rslt = FALSE;           // Fail if mismatch
    else {                                      // Else scan argument
      ++p;                                      // Move text pointer
      if (*p == 'A') {                          // Case: read some argument
        i = (*++p) - 1;                         // Index of read argument
        ap = (args == NULL) ? NULL : &args[i];  // Address of read argument
        if (!match_arg(&txt, &pp->arg[i], ap)) // Scan text to read argument
          rslt = FALSE; }                       // Remember matching failure
      else if (...) ...                         // Other cases, e.g., %%
    if (!rslt) break; }                         // Stop on failure
  }                                             // Return failure or
  return (!rslt) ? -1 : pp->retargs;            //    number of read arguments
}
match_arg(sp, pa, argp)                         // Read some argument
{
  s = *sp;                                      // Point to text to read
  if (pa->type == PMAT_TYPE_INT) {              // Read integer
    mode = 0;                                   // Reset success flag
    v = 0;                                      // Reset computed integer
    while (TRUE) {                              // Scan input string
      if (!isdigit(*s)) break;                  // Look for digits
      v = v*base + *s++ - '0';                  // Compute integer
      mode = 1; }                               // Set success flag
    if (mode == 0) return FALSE;                // Fail unless success flag set
  }
  else if (pa->type == PMAT_TYPE_STRING) {      // Read string
    bufp = buf;                                 // Reset string buffer pointer
    len = 0;                                    // Reset string length
    while (*s != 0 && !isspace(*s)) {           // Scan input string
      if (len++ < MAX_ARG_SIZE) *bufp++ = *s;   // Check buffer overflow
      ++s; }                                    //    and copy char to buffer
    *bufp = 0;                                  // Mark end of string
    v = (argp != NULL) ? strdup(buf) : 0;       // Make fresh string copy
  }
  else if (...) ...                             // Other type cases
  if (argp != NULL) *argp = v;                  // Store read value
  return TRUE;                                  // Succeed
}                                                                3a. Mechanism
```

Figure 3. Pattern matcher

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

3b. Example of use

```
{                                          // Scan constant prefix "DEBUG AT "
  if (strncmp(msg, "DEBUG AT %A\001 %A\002", 9)) {
    n = -1;
    goto end;
  }
  rslt = TRUE;
  msg += 9;
  bufp = buf;                              // Scan string (i.e., %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 integer (i.e., %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:
  if (!rslt) n = -1; else n = 2;           // Did the text match the pattern?
end:
  if (n >= 0)                              // Callback invocation
    my_callback(args[0], args[1]);
}
```

3c. Specialization

Figure 3. (continued) Specialized pattern matcher

a small representative excerpt. Figure 3b illustrates a typical call to the pattern matcher, with the pattern argument `"DEBUG AT %1s %2d"`. To actually match a string against a given pattern, a pattern descriptor (a C structure) must first be computed. (This is not shown in Figure 3a for size reasons.) This descriptor, that somehow "precompiles" part of the pattern, 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 with argument position information: all arguments to read are marked as `%A`, regardless of types, and are followed by a (`char`) number, *e.g.*, `"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.

4.2.2. Efficiency problems. As stated by Reiss [62, 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).

4.2.3. Application of partial evaluation. The static parameter here is the event pattern. 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 3a a simplified version of the code. For example, numbers are only read in decimal notation, not in octal nor hexadecimal. Because of a current limitation in Tempo regarding static loops that contain dynamic exits, the code was also slightly patched as a workaround. Figure 3c shows the partial evaluation (including inlining) of the call to the pattern matcher displayed in Figure 3b.

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 string comparison. Partial evaluation of pattern matching has been well studied in the context of functional and logical programming [13, 23, 65]. The performance gain varies according the complexity of patterns. In our case, with pattern `"DEBUG AT %1s %2d"` and text `"DEBUG AT ./tree.c 24"`, using `gcc2.8.1 -O2` on a 200MHz Sun UltraSPARC running Solaris 2.5, the specialized code is 3.0 times faster than the original code.

Pattern matching can be combined with the optimization of selective broadcast (see Section 4.1). Assuming patterns and event strings are known at specialization time, all pattern matching results (success or failure) can be computed by partial evaluation. Broadcasts then directly translate into explicit callback invocations, with no lookup; arguments to those callbacks are just calls to explicit string conversions.

Besides, 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 first 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.

4.3. Tight integration of software layers

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 [47].

4.3.1. The mechanism. 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 local, machine-dependent data to a network-independent format, standardized by the eXternal Data Representation protocol (XDR). The user specifies the interface of the function using an Interface Definition Language. The IDL compiler `rpcgen` then generates automatically "stub" routines 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.), dispatches depending on scalars data size, choice between encoding and decoding (the same routine can perform both), choice of generic coding medium (memory, stream, etc.), reading/writing in input/output buffers with overflow checks. Figure 4a shows the bottom of RPC layers stack. As may be seen, the implementation is highly parameterized. For example, a function like `xdr_long()` can achieve both encoding and decoding, depending on a flag provided in the arguments.

A typical execution context for client encoding is displayed in Figure 4b. The function `xdr_pair()` encodes or decodes a pair of integers. It has been generated automatically, given the definition of the remote procedure interface.

4.3.2. Efficiency problems. Layered systems have several good properties: their design follows incremental abstraction steps, they favor extensibility and reuse, and different implementations of the same layer can be interchanged. However, as noted Shaw and Garlan, "considerations of performance may require closer coupling between logically high-level functions and their low-level implementation" [64, p. 25]. This is precisely what partial evaluation achieves automatically.

```
xdr_pair(xdrs, objp)                          // (User generated from IDL spec)
{
  if (!xdr_int(xdrs, &objp->int1))            // Encode or decode 1st int
    return (FALSE);                           // Stop on failure
  if (!xdr_int(xdrs, &objp->int2))            // Encode or decode 2nd int
    return (FALSE);                           // Stop on failure
  return (TRUE);                              // Succeed
}
xdr_int(xdrs, ip)                             // Read or write some integer
{
  if (sizeof(int) == sizeof(long))            // Depending on integer size
    return xdr_long(xdrs, (long *)ip);        //   choose coding routine
  else
    return xdr_short(xdrs, (short *)ip);
}
xdr_long(xdrs, lp)                            // Read or write a long integer
{
  if (xdrs->x_op == XDR_ENCODE)               // If encoding requested
    return XDR_PUTLONG(xdrs, lp);             //   encode long int into I/O buffer
  if (xdrs->x_op == XDR_DECODE)               // If decoding requested
    return XDR_GETLONG(xdrs, lp);             //   decode long int into I/O buffer
  if (xdrs->x_op == XDR_FREE)                 // If free resource requested
    return TRUE;                              //   no op in this implementation
  return FALSE;                               // Other cases are illegal
}
#define XDR_PUTLONG(xdrs,longp) \             // Use specified coding medium
  (*(xdrs)->x_ops->x_putlong)(xdrs,longp)     //   (memory, stream, ...)

xdrmem_putlong(xdrs, lp)                      // Write long to memory
{
  if ((xdrs->x_handy -= sizeof(long)) < 0)    // Buffer overflow check
    return FALSE;
  *(xdrs->x_private) = htonl(*lp);            // Buffer copy
  xdrs->x_private += sizeof(long);            // Buffer offset increment
  return TRUE;                                // Succeed
}                                             // (htonl: treat little/big endian)
#define htonl(x) x                                              4a. Mechanism
```

```
{
  xargs = xdr_pair;                           // Specify argument-coding routine
  xdrs->x_ops->x_putlong = xdrmem_putlong;    // Choose "memory" coding
  xdrs->x_op = XDR_ENCODE;                    // First, set up encoding
  if (!(*xargs)(xdrs,argsp))                  // Perform encoding
    return cu->cu_error.re_status;            // Abort on error (overflow)
  sendto(...);                                // Emit encoded call on network
}                                                              4b. Example of use
```

```
{
  *(xdrs->x_private) = argsp->int1;           // Encode 1st argument into buffer
  xdrs->x_private += 4;                       // Increment buffer pointer
  *(xdrs->x_private) = argsp->int2;           // Encode 2nd argument into buffer
  xdrs->x_private += 4;                       // Increment buffer pointer
  sendto(...);                                // Emit encoded call on network
}                                                              4c. Specialization
```

Figure 4. Tight integration of micro-layers

Concerning XDR, the integration of data is fixed by the protocol. Control integration seems relatively important: invocations are all explicit, apart from the indirect call (through a function pointer) in `XDR_PUTLONG()`. However, invocations are numerous and exit statuses are propagated (and often checked) through each micro-layer. Moreover, a dispatch function like `xdr_long()` does not actually produce any result; it merely acts as a switch. In addition, the output buffer is checked for overflow for each single integer encoding, rather than once and for all. All this introduces significant overhead.

4.3.3. Application of partial evaluation. The information driving the dispatch in `xdr_long()` and the number of integers written in the output buffer can be completely known from the execution context. Consequently, the exit status of the inner-most layer can be known prior to run time (buffer overflow or not). Propagating this information to each layer makes the tests unnecessary.

Figure 4c shows what partial evaluation achieves automatically on such an architecture [50]. Note that the dispatches, the propagation of exit status, and the safety checking for buffer overflow have all been removed. For specializing upper layers (not shown here), we had to slightly change the original code though, because of a limitation in Tempo regarding the binding-time polyvariance of structures. The specialized encoding routines can be up to 3.7 times faster [49]; including time for network transport, remote procedure calls can be up to 1.5 faster.

4.4. Compiling language interpretation

Scripting languages [53] are intended to glue together a set of powerful components (building blocks) written in traditional programming languages. Scripting languages simplify connections between components and provide rapid application development. Domain-specific languages [26, 80] exploit the same idea.

4.4.1. The mechanism. The Toolbus coordination architecture [6] uses this concept. It consists 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 single script that coordinates all the processes. This script is written in a language specific to the Toolbus architecture, called *T script*. 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.

Figure 5b shows a sample script written in T Script. As described in [6], 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

```
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) {
      SB_cons_term(pe->atomic->term, SB_rec_msg());
    }
    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) {
      SB_cons_term(pe->atomic->term, result);
    }
    else if (pe->atomic->action == SND_DO) {
      SB_prod_term(pe->atomic->term);
      (*eval)(pe->atomic->term);
    }
  }
}
```

5a. The mechanism

Figure 5. T script interpreter

prints their result. The evaluation function for the calculator treats terms of the form `calc(`X`)` by evaluating the expression specified by X and terms of the form `prn(`X`)` by printing X. The script consists of an iteration of the four atomic processes `rec-msg`, `snd-eval`, `rec-value`, and `snd-do`, which respectively wait for a message and match it to the term `calc(Exp)`, build the term `run(Exp)` and pass it to the evaluation function for the tool, place the return value of the previous `snd-eval` into the `Val` variable, and build the term `prn(Val)` and pass it the evaluation function. Iteration continues as long as the `rec-msg` continues to succeed, *i.e.* as long as there are messages that match the term `calc(Exp)`.

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

5b. Example of use

```
void specialized_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 = my_handler(petmp3->atomic->term);

    petmp4 = petmp2->e2->e1;
    SB_cons_term(petmp4->atomic->term, result);

    petmp4 = petmp3->e2;
    SB_prod_term(petmp4->atomic->term);
    my_handler(petmp4->atomic->term);

    ok = try(pe->e1);
  }
}
```

5c. Specialization

Figure 5. (continued) T script compilation via partial evaluation

Figure 5a 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()` and `SB_rec_msg()` respectively send and receive a message.

4.4.2. Efficiency problems. Most often, scripts are interpreted and type-less. These features provide more flexibility to the gluing language. However, they also introduce performance overhead that becomes significant when the building blocks are small. As stated by Bergstra and Klint, "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." [6, p. 82].

The performance overhead of interpretation is due to a poor control integration. For example, in `mini_printf()` (see Figure 1a), the format is interpreted. When calling `mini_printf()`, some execution time is lost in scanning the format string, before eventually invoking the printing routines `putchar()` and `putint()`. Similarly, in the case of Toolbus, interpreting T scripts leads to a significant latency in communications.

4.4.3. Application of partial evaluation. The T script interpreter has a similar structure to that of `mini_printf()` (see Figure 1a). As for `mini_printf()`, partial evaluation successfully eliminates the interpretation of T scripts, producing a program similar to what one would write by hand to implement the script functionality. Given a C-structure representation of the script shown in Figure 5b (`pe` argument) and the evaluation function `my_handler` (`eval` argument), specialization yields a 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 (see Figure 5c). 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 further applied to them in a similar manner as in Section 4.2.

The application of partial evaluation to interpreters has been extensively studied [38]. In fact, constructing compilers from interpreters in one of the standard use of partial evaluation [19]. Similarly, a run-time specializer yields a JIT (just-in-time compiler) for the price of an interpreter. It is thus not surprising that partial evaluation is advocated as a general tool to help building domain-specific languages (DSLs) [71].

Typically cited performance gains range from 10 to 100, depending on the static semantics of the language being interpreted [21]. Our own experience of using Tempo for specializing interpreters gives similar figures. For example, an interpreter for PLAN-P (an active network language) shows execution-time speedups of 100 for compile-time specialization, and 40 for run-time specialization [72]. On a learning bridge case study, the latency of the run-time specialized PLAN-P program is similar the C hand-written one, and the throughput is only 20% slower. This offers safety and programmability of application protocols for a very little performance overhead. Another experiment involved GAL, a domain-specific language for specifying video card drivers [73]. While keeping the benefits of a high-level description (in particular, productivity and maintainability, as the specification is 10 times smaller), the drivers generated by Tempo have the same performance as hand-coded ones.

4.5. Efficient instances of generic libraries

Generic libraries like libg++, NIHCL, COOL, or the Booch C++ Components [8] have had a large success in achieving reuse. However, for performance reasons,

```
Real _in_prod(VEC *a, VEC *b, u_int i0)        // Safe encapsulation
{
  if (a == (VEC *)NULL || b == (VEC *)NULL)    // Well-formedness verification
    error(E_NULL, "_in_prod");
  limit = min(a->dim, b->dim);
  if (i0 > limit)                              // Service validity checking
    error(E_BOUNDS, "_in_prod");
  return __ip__(a->ve+i0, b->ve+i0, (int)(limit-i0));
}
Real __ip__(Real *dp1, Real *dp2, int len)     // Faster unsafe implementation
{
  sum = 0;
  for (i = 0; i < len; i++)
    sum += dp1[i] * dp2[i];
  return sum;
}
```

6a. The mechanism

```
{
  norm = v_get(3);                             // Create two vectors of size 3
  light = v_get(3);
  ...
  norm_dot_light = _in_prod(norm, light, 0);
}
```

6b. Example of use

```
{
  norm = ...
  light = ...
  ...
  norm_dot_light = norm->ve[0] * light->ve[0] +
                   norm->ve[1] * light->ve[1] +
                   norm->ve[2] * light->ve[2];
}
```

6c. Specialization

Figure 6. Optimization of a call to a math library function

they implement a large number of hand-written specific components that represent a unique combination of features (*e.g.* concurrency, data structures, 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 [5]. However, the generator is not general purpose but specific to a given architecture. Moreover, the generated code may still contain aspects of the software architecture in the specification. Furthermore, they only address compile-time code generation. In some cases, computer algebras such as Maple and Mathematica may also automatically generate parts of libraries from given mathematical models (yielding Fortran or C code). Still, this is very restricted and specific to a given model and computer algebra system.

4.5.1. The mechanism. We have taken as an example the Meschach Library [68] developed at the Australian National University, which provides a wide range of matrix computation facilities. It is very general in its design and implementation. For example, many functionalities in Meschach are implemented using two routines. The first one provides a safe, clean interface; it controls the validity of arguments and performs bounds checking. The second one does the actual computation on raw data. Such an example is shown in Figure 6a: the function `_in_prod()` provides a safe encapsulation to the unsafe function `__ip__()` which computes an inner product. Figure 6b gives an example of how the library is used: two three-dimension vectors are allocated and used for an inner-product operation.

4.5.2. 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, the inner-product computation involves 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 must thus be considered as implicit. Another interpretation is to see this as the well-formedness aspect of the data integration problem. In any case, the components need tighter integration.

4.5.3. Application of partial evaluation. As may be seen from Figures 6b and 6c, 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 case (see Section 4.3.3). In addition, the raw computation itself is slightly improved using loop unrolling. When an application heavily relies on a generic library, such optimizations become crucial. A study of compile-time and run-time specialization of numeric functions reports performance gains up to a factor of 12, including speedups of 5 on the Fast Fourier Transform [51].

5. Discussion

In this section, we analyse the case studies presented in Section 4 and characterize the scope and the achievements of partial evaluation.

Case studies analysis. All the case studies in Section 4 have two aspects in common. First, some states are encoded in data rather than in the control flow of the program: callback registration array, event pattern, encode/decode flag and buffer size, T script text, vector size. Second, some of these data are constant for a given architecture configuration. The effect of program specialization is to eliminate computations depending on these configuration data, thus also pruning and reducing control.

Generality. Having configurations encoded in data is a general and common technique. This generality is what makes partial evaluation systematic, as opposed to optimizations that rely on definite software architectures or application domains. A new technology does not have to be developed when a new software architecture needs to be treated; only implementing the architecture and studying its specialization is required. However, architecture- or domain-specific optimizations that are richer than specializing a generic program are beyond the scope of partial evaluation.

Besides, because partial evaluation can also be performed at run time, depending on run-time values, flexibility is not limited to source-code structuring: the architectural configuration that decides the interaction between components may evolve dynamically, as the program runs.

Predictability. Although we may expect partial evaluation to be successful each time a configuration state is encoded in some data, we must check that this state depends only on available constants. For this, the binding-time analysis phase prior to code transformation enables the user to visualize the code regions to be eliminated by specialization (*cf.* Section 3.3).

However, it is generally not possible to determine the actual performance improvement because it is as complex as estimating the execution time of general programs. Moreover, the speedup can vary depending on the value of the known input. Preliminary work by Andersen and Gomard suggests that traditional specialization provides an improvement that is linear in the structure of the known input [2]. (Other program transformations like lazy rewriting and common subexpression elimination can theoretically lead to supra-linear improvements.) In practice, experience shows that coupling profiling to the visualization of code regions to be specialized provides good estimates of the performance improvement.

Degree of automation. One key advantage of partial evaluation is that it is automatic. More precisely, the program specialization process, given some known input, is automatic. In particular, when the configuration data is not a parameter but hard-coded into the text of the program (*e.g.*, registration and broadcast statements), blindly running a partial evaluator on the whole application automatically yields an optimized implementation.

However, there are several practical limitations to full automation: programs generally have external configuration parameters; the specializer does not offer any automatic speedup prediction; uncontrolled loop unrolling may arbitrarily increase the code size (*cf.* Section 3.1.3); existing partial evaluator prototypes cannot reasonably specialize more than a few thousand lines of code. It must also be kept in mind that, altough partial evaluation should typically be used in the latest stages of development (after prototyping and debugging), very large code analyses and specializations can take several hours. Even if partial evaluators become more efficient, blindly specializing a whole application is unpractical.

Thus, in practice, using partial evaluation is not a fully automatic process but requires some user interaction. The user has to identify configuration data (using his own expertise, some profiling data, or program analyses), split the program source to focus on relevant parts (*e.g.*, identify the pattern matcher), and instruct the partial evaluator. Examining code regions to be specialized provides a feedback to refine the identification and impact of configuration data. Then, given configuration values, specialized code has to be re-plugged into the original application and used appropriately.

Ongoing work in our group aims at simplifying the user's interaction with the specialization process, that is, offering high-level means to express specialization intentions [75]: without altering the original source code, the user can specify what to specialize, how to specialize it and when to use specialized code. Then, a compiler processes these declarations, and the original source code is automatically instrumented so that it uses specialized code when appropriate. The prototype of a Java specialization classes compiler has been implemented and is publicly available[3].

Our experience is that existing code (not written by us) sometimes requires minor rewriting before Tempo can specialize it successfully. Indeed, we had to slightly change the original code in the two cases studies relying on existing code (Field pattern matching and Sun RPC). These "binding-time improvements" [39] are needed when static and dynamic computations are too much intertwined and end up making all computations dynamic. Obtaining good binding times when writing code explicitly with specialization in mind is easier.

6. Related work

There are several other techniques for deriving efficient implementation of software architectures, preserving or not a form of flexibility. These techniques can be manual or automatic, specific or systematic, implementation-time or code-generation-time.

Language-level mechanisms. Some optimizing compilers implement a simple form of constant propagation and constant folding. However, propagation is often intra-procedural, and limited to scalar values. The range of optimizations provided by partial evaluation goes much beyond that. Indeed, none of the above case studies could have been treated using any existing compiler. A partial evaluator and an optimizing compiler are somehow complementary.

The oldest and roughest approach to tighten the integration of components to improve performance consists in hand-optimizing the implementation code, using any a priori knowledge about the execution context. Because it is tedious and error-prone, manual specialization is generally local, *i.e.* restricted to a small "window" of code; it does not scale up to large systems. In addition, manual optimization tends to duplicate code and freeze choices early in the development process. It thus conflicts with maintenance and extensibility. Yet, it is still widely used to optimize critical paths when performance is a major concern (*e.g.*, in Chorus IPCs [11]).

Another common practice is to rely on a two-level language: macros (as is C or Scheme), templates (as in C++), etc. The idea is to *program* code expansion. Although automatic, this approach is not systematic: each new optimization (*i.e.*, code expansion) must be explictly programmed. However, finely tuning code expansion can lead to a fast final implementation [4, 46, 74]. One major limitation to this approach is that the optimization usually can only be local, not contextual: two separate rewritings cannot usually cooperate. The reason is that the rewriting language often does not have states (or states more complex than scalars) nor scoping mechanisms. For example, in the selective broadcast case, macros and templates are useless for optimizing registration and broadcast. Besides, there are other, practical drawbacks: macros and templates are usually more complex to develop and maintain that standard programs; they require the user to explicitly stage the computations. Moreover, unless such systems have reached enough maturity, the high-level part of the two-level language is often untyped or cannot be type-checked and compiled separately. The user can only examine expanded code, which makes debugging more difficult, especially when expansion occurs at run-time.

Specification-level mechanisms. One way to overcome these limitations is to automatically generate non-flexible implementation from a flexible design specification.

Common examples are generators of code skeletons, stubs, etc. For example, `rpcgen` (the IDL compiler for the Sun RPC) translates the definition of a procedure interface into actual code. However, as discussed in Section 4.3.3, there are still optimization opportunities captured by partial evaluation which are left unexploited by the IDL compiler. These opportunities are located inside the libraries used by all generated code, where knowledge about the specific execution context is not exploited. Smarter compilers exist [27] but are very specific to the domain or the application.

In Section 4.4, a domain-specific language was defined using an interpreter. Languages can also be defined using a specific semantics formalism. An implementation is then derived from the semantics definition. Such a technology is developed in the SDRR project [43]. It is based on higher-order functional definitions and incorporates a form of specialization. Still, the authors recognize that there is much more that can be done to further improve the performance of the generated Ada code.

Experiments are missing to compare partial evaluation with more general code synthesis techniques. Our intuition is that partial evaluation is mostly complementary to these techniques. In particular, systems that relies on axiomatized libraries [69] do not cover specialization of library routines. Moreover, since specification formalisms, by essence, are not meant to express operational behaviors, it is likely that the generated code contains optimization opportunities. Concerning, in model-integrated system development [40] and aspect-oriented programming [42], the customization of components is not general but domain-specific. Besides, these strategies do not primarily focus on performance but on code structuring.

Run-time flexibility. Run-time flexibility adds other tradeoffs: regardless of the approach, being more platform independant favors porting and extensibility but compromises efficiency. The fact that generated code is in this case binary rather than textual makes debugging a delicate issue. This issue not only affects the user but also the developer and the maintainer of the related tools.

Dedicated run-time specializers have the advantage and drawback of being a domain-specific artefact: they produce efficient code but they are not easily extensible [61]. Such specializers are hard to develop, maintain, extend and port.

More general mechanisms have been proposed, consisting in run-time two-level (or multi-level) languages. Besides a delicate debugging process, they have the same advantages and drawbacks listed above concerning compile-time multi-level languages. Some systems include type and coherence checking of the different language levels [34, 70]; others just trust the user's annotations [28]. Besides Tempo, there exists other general run-time specializers which may or not rely on user annotations [33, 45]. However, there were not used specifically on various software architectures.

7. Conclusion

The literature on software architectures presents implementations which are a compromise of flexibility and efficiency. The reason is that genericity and extensibility introduce overhead when they are not only present at the design level but also in the implementation.

We characterize the fundamental inefficiency problems in flexible architecture implementations as being related to the loose integration of data and control in software components. We advocate the use of partial evaluation, a systematic and automatic program transformation that can turn flexible implementations into efficient ones while retaining flexibility at the structuring level. This claim is validated by five case studies. The discussion on the general applicability of partial evaluation argues that it is a major tool for achieving *program adaptation* [22].

Future work

Partial evaluation is well-suited for control integration. It also addresses the safety and security aspects of data integration. However, it does little concerning the heterogeneity problem of data integration (*cf.* Section 2.1).

A more powerful program specialization technique, known as *deforestation* [78], can be used to treat certain combinations of a sequence of data conversions. However, to our knowledge, it has not yet been applied to imperative programming, except in the simpler case of filter fusion [59]. Semi-automatic approaches to copy elimination between software layers have been considered [77] 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* [17] and *generalized partial computation* [30]. However, they have not yet been put into practice on realistic applications.

It is clear that using a partial evaluator today still requires some expertise. Yet, we cannot expect all programmers to be or to become experts in specialization. Thus, our long-term goal is to hide specialization as much as possible. In particular, given implementations of flexible architectures (including design patterns [31]) and a careful analysis of the application of partial evaluation, we want to provide the user with a programming environment where those predefined software architectures have guaranteed optimizations.

Acknowledgments

The research presented here was partly supported by CNET/France Telecom grant 96-1B-027. We would like to thank Brown University and the Australian National University (Canberra), that have given us access respectively to the sources of Field and Meschach. We would like to thank as well the anonymous referees and the members of the Compose group for their helpful and insightful comments.

Notes

1. Compose group home page: `http://www.irisa.fr/compose`
2. Tempo, a specializer for C: `http://www.irisa.fr/compose/tempo`
3. JSCC, a Java specialization classes compiler: `http://www.irisa.fr/compose/jscc`

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. L.O. Andersen and C.K. Gomard. Speedup analysis in partial evaluation: preliminary results. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–7, San Francisco, CA, USA, June 1992. Yale University, New Haven, CT, USA. Technical Report YALEU/DCS/RR-909.
3. Maurice J. Bach. *The Design of the UNIX Operating System*, chapter 5, pages 111–119. Software Series. Prentice Hall, 1986.
4. D. Batory and B. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, February 1997.
5. 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.
6. J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In Ciancarini and Hankin [12], pages 75–88.
7. 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 [67], pages 267–283.
8. 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).

9. 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 [79], pages 118–126.
10. S. Chirokoff and C. Consel. Combining program and data specialization. In PEPM'99 [56]. to appear.
11. Chorus. Chorus kernel v3 r5 implementation guide. Technical Report CS/TR-94-73.1, Chorus Systemes, 1994.
12. Paolo Ciancarini and Chris Hankin, editors. *Coordination and models, Proceedings of the first international conference, Cesena, Italy*, number 1061 in LNCS. Springer Verlag, 1996.
13. C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.
14. 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.
15. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.
16. 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. [24], pages 54–72.
17. C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993. Extended version of [18].
18. 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).
19. C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the* 10^{th} *International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in Lecture Notes in Computer Science, pages 170–194, Pisa, Italy, September 1998.
20. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL96 [58], pages 145–156.
21. C. Consel and Danvy O. Static and dynamic semantics processing. In *Conference Record of the Eighteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, Orlando, FL, USA, January 1991. ACM Press.
22. Charles Consel. Program adaptation based on program transformation. *ACM Computing Surveys*, 28(4es):164–167, 1996.
23. O. Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37:315–322, March 1991.
24. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, February 1996.
25. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP'95*, Aarhus, Denmark, August 1995. Springer-Verlag.
26. *Conference on Domain Specific Languages*, Santa Barbara, CA, October 1997. Usenix.
27. Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 44–56, Las Vegas, Nevada, June 15–18, 1997.
28. D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL96 [58], pages 131–144.
29. D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [67], pages 251–266.
30. Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *J. Theoretical Computer Science*, 90:60–79, 1991.
31. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
32. D. Garlan, G.E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE journal Computer*, 25(6):30–38, June 1992.

33. B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. Annotation-directed run-time specialization in C. In PEPM'97 [55], pages 163–178.
34. L. Hornof and T. Jim. Certifying compilation and run-time code generation. In PEPM'99 [56], pages 60–74.
35. *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, Santa Fe, New Mexico, December 1998. Springer-Verlag.
36. ISO. Specification of abstract syntax notation one (ASN.1). ISO standard 8824, 1988.
37. Ian Jacobs, Janet Bertot, Francis Montagnac, and Dominique Clement. The SOPHTALK reference manual. Technical Report RT 150, INRIA, February 1993.
38. N.D. Jones. What *not* to do when writing an interpreter for specialisation. In Danvy et al. [24], pages 216–237.
39. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
40. G. Karsai, A. Misra, J. Sztipanovits, A. Ledeczi, and M. Moore. Model-integrated system development: Models, architecture, and process. In *Proceedings of the Computer Software and Applications Conference (COMPSAC)*, pages 176–181, Bethesda, MA, August 1997.
41. Siau Cheng Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In PEPM'91 [54], pages 211–222. ACM SIGPLAN Notices, 26(9).
42. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.
43. R. Kieburtz, F. Bellegarde, J. Bell, J. Hook, J. Lewis, D. Oliva, T. Sheard, L. Walton, and T. Zhou. Calculating software generators from solution specifications. In *Theory and Practice of Software Development (TAPSOFT)*, volume 915 of *LNCS*, pages 546–560. Springer Verlag, 1995.
44. T.B. Knoblock and E. Ruf. Data specialization. In PLDI'96 [57], pages 215–225. Also TR MSR-TR-96-04, Microsoft Research, February 1996.
45. P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [57], pages 137–148.
46. 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.
47. G.R. McClain. *Open Systems Interconnection Handbook*. Intertext Publications, McGraw-Hill, New York, 1991.
48. 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.
49. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.
50. G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In PEPM'97 [55], pages 116–125.
51. F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.
52. OMG. *CORBA: The Common Object Request Broker: Architecture and Specification*. Framingham, 1995.
53. John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 1998.
54. *Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).
55. *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

56. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, TX, USA, January 1999. ACM Press.
57. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5).
58. *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, St. Petersburg Beach, FL, USA, January 1996. ACM Press.
59. Todd A. Proebsting and Scott A. Watterson. Filter fusion. In POPL96 [58], pages 119–130.
60. 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 [67], pages 314–324.
61. C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.
62. Steve P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.
63. U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, Lisbon, Portugal, June 1999. To appear.
64. M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996.
65. D. A. Smith. Partial evaluation of pattern matching in CLP domains. In PEPM'91 [54], pages 62–71. ACM SIGPLAN Notices, 26(9).
66. R. Snodgrass. *The Interface Definition Language: Definition and Use*. Computer Science Press, Rockville, MD, 1989.
67. *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.
68. D. R. Stewart. *MESCHACH: Matrix Computations in C*. University of Canberra, Australia, 1992. Documentation of MESCHACH Version 1.0.
69. M. Stickel, R. Waldinger, M. Lowry, and T. Pressburger. Deductive composition of astronomical software from subroutine libraries. In *Twelfth International Conference on Automated Deduction (CADE)*, volume 814 of *LNCS*, pages 341–355, Nancy, France, June 1994.
70. W. Taha and T. Sheard. Multi-state programming with explicit annotations. In PEPM'97 [55], pages 203–217.
71. S. Thibault and C. Consel. A framework of application generator design. In M. Harandi, editor, *Proceedings of the Symposium on Software Reusability*, pages 131–135, Boston, Massachusetts, USA, May 1997. Software Engineering Notes, 22(3).
72. S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.
73. S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. In DSL'97 [26], pages 11–26.
74. T.L. Veldhuizen. Arrays in Blitz++. In ISCOPE'98 [35].
75. E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, USA, October 1997. ACM Press.
76. E.N. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: the RPC case study. In WCSSS'96 [79], pages 24–28.
77. E.N. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noyé, and C. Pu. A uniform and automatic approach to copy elimination in system extensions via program specialization. Research Report 2903, INRIA, Rennes, France, June 1996.
78. Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
79. *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, Tucson, AZ, USA, February 1996.
80. *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.