Mapping Software Architectures to Efficient Implementations via Partial Evaluation

Renaud Marlet Scott Thibault Charles Consel [*]
IRISA / INRIA - Université de Rennes 1
Campus universitaire de Beaulieu
35042 Rennes cedex, France
{marlet,sthibaul,consel}@irisa.fr http://www.irisa.fr/compose

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, layers, and generic libraries. We show how partial evaluation can systematically be applied in order to optimize those mechanisms.

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 the time from conception to product release, providing the competitive advantage or disadvantage. Typical concerns of software architectures include reuse and safety.

[*]This work has been partly supported by FRANCE TELECOM CTI-CNET 96-1B-027.

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 reusability, 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 [23], data abstraction and object-oriented organization, event-based communication [32], coordination [7] and domain-specific languages [42], and software buses [28].

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. 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 [21, 31]. 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]. 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 a 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 are the following.

- 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. Due to lack of space, the pattern matching and interpreter examples are only briefly described here; an extended version of this paper is also available [22] [1].

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.

[1] `http://www.irisa.fr/compose/papers/papers.html`

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 [17]. As a consequence, 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 [23], data abstraction and object-oriented organization, event-based communication [32], coordination [7] and domain-specific languages [42], and software buses [28].

There exist basically two main ways to achieve communication between software components: states and events. A *state* consists of some information available at any time. Access to a state may be explicit (*e.g.*, reference to a global variable) or implicit (*e.g.* with blackboard techniques [26]). 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.

Typically explicit procedure invocation and direct state referencing is faster than the implicit mechanisms. However, 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 conflicts with 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 on one hand, to keep flexibility in the design but on the other hand, 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 run time. 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]. 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.

The central idea in optimizing component integration is *specialization*. It ranges from the specialization of the connection between components to the complete merging of the components' functionalities. This is precisely the goal of a technique known as *program specialization* (or *program adaptation*). Program specialization is detailed in the next section.

3. 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 can 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, since 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 eliminated. 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. In the top of the figure is a definition of `mini_printf()`, a simplified version of the Unix `printf()` text formatting function. In the bottom of the figure is 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 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 rely only on the known values (here `fmt`). All these computations are eliminated and do not appear in the specialized program.

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*. However, specialization can also involve loop unrolling, which may

```
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(' ');
  putint(val[0]);
}
```

Figure 1. Specialization w.r.t. `fmt = "n: %d"`

increase code size. For example, in Figure 1, the whole loop has been unrolled; if the format string had been longer, many calls to `putchar()` would have appeared in the specialized program. Besides, if building the specialized program p_{d_1} comes at a certain price, it is only worth it 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 [21] and operating systems [31]. 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 advanced *compilers* can achieve intra-procedural propagation and folding of scalar constants, as well as inlining. While this is enough to optimize simple parameterization, 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 [19]. 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.

For a long time 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 at the basis of the development of adaptable system in a number of major research projects and in different areas such as networking [24, 38], graphics [20], and operating systems [14, 30]. What we are interested in is to use partial evaluation to generate context-specific efficient instances of 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 [8] 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 partial evaluator for C named C-Mix [1]. (See [25] for a comparison.)

Partial evaluation is often 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 run time [11]. Whereas compile-time specialization is a source-to-source program transformation, run time specialization relies on binary template assembly for fast code generation. Dynamic selection of specialized routines is presented in [37]. 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, run time 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* [33]. 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 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 [16].

The Field programming environment is a typical, representative example of such an architecture [32]. 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 (top and middle) 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 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). 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 for tool integration.

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 a broadcast is linear in the total 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 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.

Application of partial evaluation. The lower portion 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 a broadcast in the original program is linear in

```
my_execution_context() {  /*** original ***/
  register_for_event( DEBUG_AT, editor_goto );
  register_for_event( DEBUG_AT, cfg_highlight );
  ...
  broadcast( BUS_ERROR, (char *)NULL );
  dbg_info->line = line;
  dbg_info->fname = fname;
  broadcast( DEBUG_AT, (char *)dbg_info );
}
```

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

```
my_execution_context() {  /*** specialized ***/
  ...
  dbg_info->line = line;
  dbg_info->fname = fname;
  editor_goto((char*)dbg_info);
  cfg_highlight((char*)dbg_info);
}
```

Figure 2. Registration and broadcast

the total number of registered events (`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 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 [6] 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 calls to the warning function. Testing the above policy then 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.

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 [32], 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 there is a match, invocation of the callback routine with arguments decoded into the proper internal format.

Efficiency problems. As stated by Reiss [32, 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. Detailed results may be found in the extended version of this paper [22]. In summary, the results are that 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. Partial evaluation of pattern matching has been well studied. Although the performance gain will vary according the pattern, results presented by Andersen, for example, indicate performance gains of a factor of 1.6 [1].

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

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 [23].

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 the network, input/output buffers with overflow checks, selection between big and little endian. The middle section of Figure 3 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 achieve both encoding and decoding, depending on a flag provided in the arguments. A typical execution context for client encoding is displayed in the top section of the figure. The stub function `xdr_pair()` 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, 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" [33, p. 25]. 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, invocations are numerous and exit statuses are propagated (and sometimes checked) through each micro-layers. Moreover, a dispatch function

```
my_execution_context()  /*** original ***/
{
  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); // buff copy
  xdrs->x_private += sizeof(long); // ptr incr
  return TRUE;
}
#define htonl(x) x
```

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

Figure 3. Tight integration of micro-layers

like `xdr_long()` does not actually produce any result; it merely acts as a switch. In addition, an output buffer is checked for overflow for each single integer encoding, rather than once and for all. All this introduces significant overhead.

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

The bottom portion of Figure 3 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 that the specialized code of RPC encoding routines is up to 3.75 times faster [25].

4.4. Compiling language interpretation

The mechanism. *Scripting languages* [29] 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 [7] and domain specific languages [42] exploit the same idea. The Toolbus coordination architecture [4] 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 script that coordinates the processes. The scripts, called T scripts, are written in a language specific to the Toolbus architecture. 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.

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" [4, p. 82]. 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 T script interpreter has a similar structure to that of the `mini_printf()` (figure 1, section 3.1). Like `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. More detailed results are presented in the extended version of this paper [22].

Partial evaluation has already been advocated as a general tool to help building domain specific languages [36]. Typically cited performance gains range from 10 to 100 depending on the static semantics of the language being interpreted [12]. Its application to interpreters has also been extensively studied [18]. 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 [5] 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 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 [3]. However, the techniques and the generator are not general purpose. In some cases, computer algebras may also automatically generate parts of libraries from given mathematical models. However, this is very restricted and specific to a model and a computer algebra system.

We have taken as an example the Meschach Library [35] 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 bounds checking. The second one does the actual computation on raw data. Such an example is shown in the middle section of Figure 4: function `_in_prod()` provides the safe encapsulation to function `__ip__()`. The top section 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, 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

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

Figure 4. Call to a math library function

the actual computation, is significant. Communication must then be considered as implicit. The components need tighter integration.

Application of partial evaluation. As may be seen in the bottom section of Figure 4, 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 slightly improved using loop unrolling. When an application heavily relies on a general library, such optimizations become crucial. A previous study of the partial evaluation of numeric functions, like the inner product here, reports performance gains from a factor of 1.4 to a factor of 12.17 [27].

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. All our examples have one aspect in common: some states are encoded in data rather than the program. We may expect partial evaluation to be successful each time this situation arises — it is actually (one of) the essence of partial evaluation. Besides, 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* [40], can be used to treat certain combinations of 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 [39] 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* [10] and *generalized partial computation* [15]. However, they have not been yet put into practice on realistic applications.

References

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

[3] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable Software Libraries. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 191–199, Dec. 1993.

[4] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In Ciancarini and Hankin [7], pages 75–88.

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

[6] 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 [41], pages 118–126.

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

[8] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [13], pages 54–72.

[9] C. Consel and S. 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).

[10] C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993. Extended version of [9].

[11] 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, Jan. 1996. ACM Press.

[12] C. Consel and D. 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, Jan. 1991. ACM Press.

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

[14] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [34], pages 251–266.

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

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

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

[18] N. Jones. What *not* to do when writing an interpreter for specialisation. In Danvy et al. [13], pages 216–237.

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

[20] T. 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.

[21] B. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, Sept. 1987. EUUG.

[22] R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. Rapport de recherche RR-3217, INRIA, Rennes, France, July 1997.

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

[24] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operating system. Technical Report 94–20, Department of Computer Science, The University of Arizona, 1994.

[25] G. Muller, E. 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.

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

[27] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. Rapport de recherche 1065, IRISA, Rennes, France, Nov. 1996.

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

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

[30] 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 [34], pages 314–324.

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

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

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

[34] *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, Dec. 1995. ACM Operating Systems Reviews, 29(5),ACM Press.

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

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

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

[38] E. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: the RPC case study. In WCSSS'96 [41], pages 24–28.

[39] E. 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.

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

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

[42] *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, Jan. 1997. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.