A Uniform Approach for Compile-Time and Run-Time Specialization

Charles Consel, Luke Hornof, François Noël, Jacques Noyé, Nicolae Volanschi

Université de Rennes / Irisa
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
{consel, hornof, fnoel, noye, volanski}@irisa.fr

Abstract. As partial evaluation gets more mature, it is now possible to use this program transformation technique to tackle realistic languages and real-size application programs. However, this evolution raises a number of critical issues that need to be addressed before the approach becomes truly practical.

First of all, most existing partial evaluators have been developed based on the assumption that they could process any kind of application program. This attempt to develop *universal* partial evaluators does not address some critical needs of real-size application programs. Furthermore, as partial evaluators treat richer and richer languages, their size and complexity increase drastically. This increasing complexity revealed the need to enhance design principles. Finally, exclusively specializing programs at compile time seriously limits the applicability of partial evaluation since a large class of invariants in real-size programs are not known until run time and therefore cannot be taken into account.

In this paper, we propose design principles and techniques to deal with each of these issues.

By defining an architecture for a partial evaluator and its essential components, we are able to tackle a rich language like C without compromising the design and the structure of the resulting implementation.

By designing a partial evaluator targeted towards a specific application area, namely system software, we have developed a system capable of treating realistic programs.

Because our approach to designing a partial evaluator clearly separates preprocessing and processing aspects, we are able to introduce run-time specialization in our partial evaluation system as a new way of exploiting information produced by the preprocessing phase.

1 Introduction

Partial evaluation is reaching a level of maturity which makes this program transformation technique capable of tackling realistic languages and real-size application programs. However, this evolution raises a number of critical issues which need to be addressed before the approach becomes truly practical.

Universality vs. Adequacy. Until now, most partial evaluators have always been developed based on the assumption that they could process any kind of application programs. They were considered *universal* partial evaluators. As partial evaluation addresses more realistic programs, it also faces new challenges when it tackles existing real-size application programs. This new situation makes it obvious that the usual, general-purpose set of transformations available in traditional partial evaluators falls short of addressing some critical needs of realistic programs.

As a consequence, not only does a partial evaluator need to offer an extensible set of transformations, but the transformations themselves should be developed based on program patterns found in typical applications programs in a given area.

Need for Design Principles. Furthermore, as partial evaluators treat richer and richer languages, their size and complexity increase drastically. Indeed, programs written in realistic languages like C expose a very wide variety of situations where partial evaluation can be applied. As a result, there is now a clear need to propose design principles to structure the added complexity of the resulting partial evaluators.

Compile-Time Specialization is Limiting. When studying components from real software systems, it becomes apparent that exclusively specializing programs at compile time is limiting. In fact, there exist numerous invariants that are not known until run time and can yet be used for extensive specialization. This situation occurs, for example, when a set of procedures implements session-oriented transactions. When a session is opened, many pieces of information are known, but only at run time. They could be used to specialize the procedures which perform the actual transactions. Then, when the session is closed, the invariants would become invalid, therefore the specialized procedures can be eliminated.

Although run-time specialization seems to involve techniques different from compile-time specialization, both forms of specialization are conceptually the same. They should thus be modeled by a unique approach rather than studied separately. Also, when considering realistic languages, the level of effort required to develop a specializer is such that pursuing some uniformity to handle both cases of specialization is critical.

In this paper we present a partial evaluator of C programs, called Tempo[1]. It is based on a general approach capable of handling programs written in a wide variety of languages which spans from the C programming language (in the case of Tempo) to a pure, higher-order dialect of Scheme (in the case of Schism [8, 7]).

[1] The name Tempo has been previously used for a pedagogical programming language to study binding times and parameter passing concepts [22].

A Uniform Approach. This approach is off-line in that it separates the partial evaluation process into two parts: preprocessing and processing [20, 11]. The former part mainly includes a binding-time analysis aimed at determining the static and dynamic computations for a given program and a division (static/dynamic) of its input. Binding-time information is subsequently used by another analysis to assign a *specialization action* (*i.e.,* program transformation) to each construct in the program [9].

The latter part (*i.e.,* processing) performs the specialization for a given action-analyzed program and specialization values. Specialization is then merely guided by the information produced by the preprocessing part. In many regards, this design is very similar to the one used to implement programming languages. For a given program, just like a compiler produces machine instructions, the preprocessing phase produces program transformations. Just like a run-time system executes compiled code, the processing phase (*i.e.,* the specializer) executes the program transformations produced by the preprocessing phase. Just like a compiled code is run many times with respect to different input values, a preprocessed program can be specialized many times with respect to different specialization values. Because specialization has been *compiled*, it is performed very efficiently. Because of the separation between preprocessing and processing, the latter can be implemented in a different language from the former, and thereby facilitate the implementation. More importantly, this design allows one to process action-analyzed programs in many different ways. Indeed, in order to perform compile-time specialization, actions can either be interpreted, or even compiled. The latter form of specialization corresponds to producing a generating extension [19, 2].

Compile-time and Run-time Specialization. More interestingly, actions can be used as a basis to perform run-time specialization. Indeed, actions directly model the shape of residual programs since they express how each construct in a program is to be transformed. In fact, we have developed a strategy to perform run-time specialization based on actions [12]. In essence, an action-annotated program is used to generate automatically source templates at compile time. Then, at run time the compiled templates are selected and filled with run-time values before being executed. This new approach has many advantages: it is general since it is based on a general approach to developing partial evaluators; it is portable because most of the specialization process is performed at the source level; and it is efficient in that specialization is amortized in a few runs of the specialized code.

An important aspect of our approach is that the preprocessing of a program is identical whether it is specialized at compile time or at run time. This is a direct consequence of the kind of information computed in the preprocessing part of the system.

Adequacy. Tempo has been targeted toward a particular area, namely system applications. More precisely, the features of Tempo have been guided by program patterns based on actual studies of numerous system programs and tight collaboration with system researchers. As a result, the accuracy of critical analyses such as alias analysis and binding-time analysis are adequate for typical system programs. Also, the program transformations address the important cases in systems programs.

More globally, this new approach to the design of a partial evaluator has had an important consequence. It clearly showed the need for *module-oriented* partial evaluation. Traditional partial evaluators assume they process a complete program. However, system programs are large enough to reach the limit of what a state of the art analysis, such as an alias analysis, can process [34]. We have therefore developed support to enable one to specialize pieces of a large system.

Summary. This paper makes a series of contributions regarding the design and architecture of a partial evaluator for a real language, applied to real programs. These contributions are as follows.

1. We present an architecture for partial evaluators which is powerful enough to be used for realistic languages and real-size application programs.
2. This architecture has been used to develop a partial evaluator of C programs. Unlike most existing systems, our partial evaluator has been carefully designed to address specific specialization opportunities found in a particular area, namely, system applications. This strategy allows us to better ensure the applicability of partial evaluation.
3. Although dedicated to a particular application area, the architecture is nonetheless open: new program transformations can easily be introduced at the action analysis level.
4. Last but not least, our architecture has proved its generality in that it allows one to perform both compile-time and run-time specialization. As a consequence, this generalized new form of specialization drastically widens the scope of applicability of partial evaluation.

Outline. In Sect. 2 we explain the preprocessing phase, followed by the different processing phases in Sect. 3. Section 4 then discusses the applications we consider. Related work is addressed in Sect. 5, and finally we give concluding remarks in Sect. 6

2 Preprocessing

As shown in Fig. 1, the preprocessing part consists of four main phases. It eventually produces transformation operations that must be performed to specialize the subject program.

2.1 Front-End

This phase transforms a C program into a C abstract syntax tree (AST). We have not written a new parser but have rather reused SUIF components [33]. SUIF is

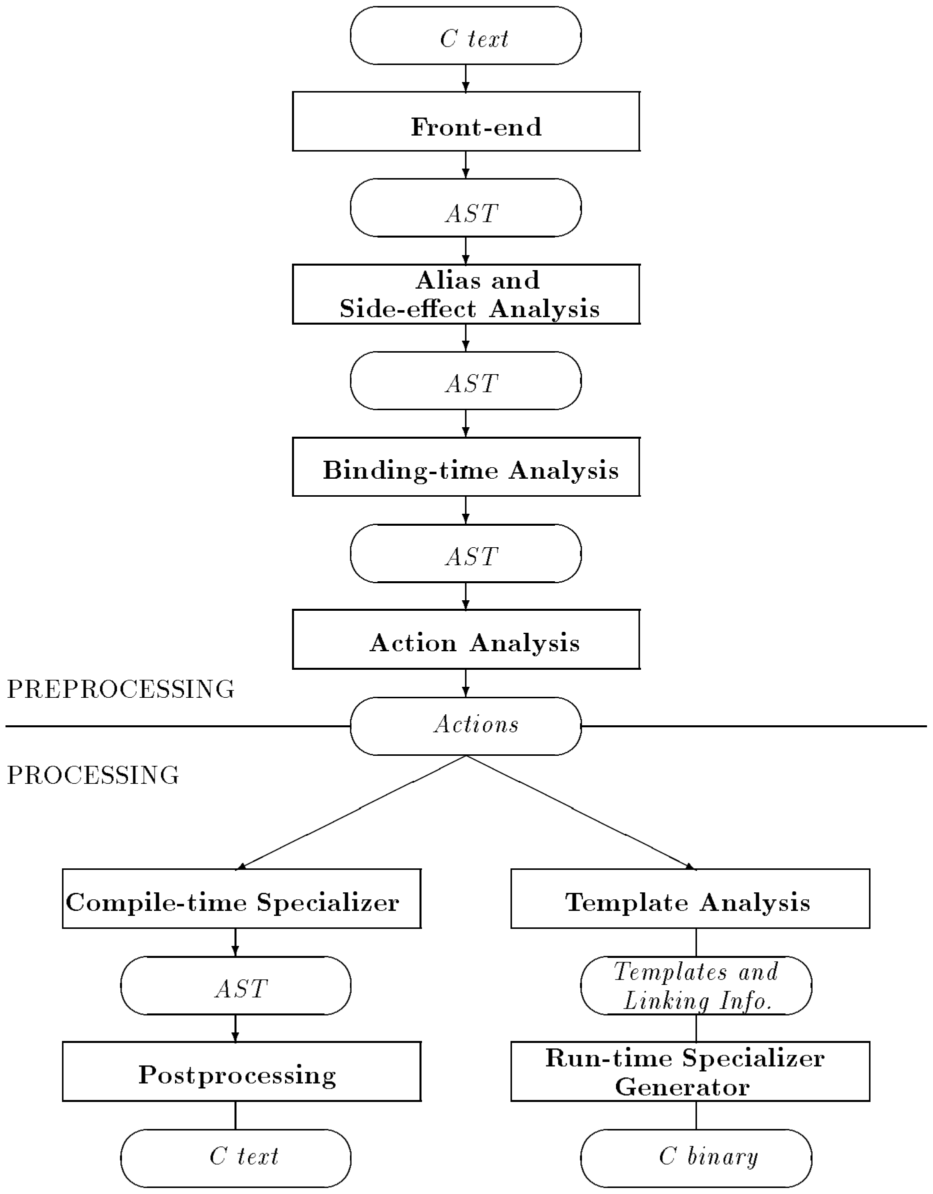

Fig. 1. A General View of Tempo

a testbed system for experimenting with program optimizations in the context of scientific code and parallel machines. Our abstract syntax is not based on the SUIF intermediate format, which is too low-level for our purposes, but on an intermediate representation used by a SUIF program transforming this format back into C. With a couple of minor modifications to SUIF we are then able to turn C programs into simple high-level ASTs. In particular, our abstract syntax includes a single, `do-while`, loop construct, a single conditional construct, no conditional expression, no comma expression, and no nested type declarations.

Two additional important transformations are applied to the ASTs obtained from SUIF. Firstly, in order to allow for compositional analyses, `goto` statements are eliminated as suggested by Erosa and Hendren [18]. Secondly, renaming makes every identifier unique in order to help building up a flattened static store (see Sect. 3). This renaming also facilitates unfolding at postprocessing time.

2.2 Alias and Side-effect Analysis

Because of the pointer facilities offered by the C language, an alias analysis is critical to determine the set of aliases for each variable in a program. This information allows the computation of binding-time properties of variables to take into account side-effects. However, unlike other contexts of use of alias analysis, partial evaluation does not require very accurate information. This situation is essentially due to the kind of static computations that are expected to be performed by partial evaluation. Indeed, based on our experience, static computations rely on invariants whose validity typically follows a very clear pattern. In fact, this is not surprising since invariant behavior must be simple enough to be understandable. Our choice of a context-insensitive analysis is further backed up by the study of Ruf [32] which shows that empirical benefits of a context-sensitive analysis have still to be measured.

Our analysis is very similar to [32] and [15]. It is based on the *points-to* model of aliasing. It is interprocedural, context-insensitive, and flow-sensitive. The analysis takes as arguments a set of procedures, a goal procedure and a description of the initial points-to pairs. The last argument allows the analysis of a set of procedures with respect to some alias context. This feature is critical to enable one to only specialize parts of a large system. Assuming that programs submitted to specialization are correct, the analysis does not separately deal with *possible* and *definite* points-to pairs (see [6, 32]).

The alias analysis is complemented by a side-effect analysis which computes the non-local store affected by each procedure call as well as which procedures are responsible for other side-effects like input/output.

2.3 Binding-Time Analysis

We have developed a binding-time analysis which annotates C programs with their binding times, given a set of procedures, its alias and side-effect information, a goal procedure, and a binding-time description of the context. This context includes the parameters of the goal procedure and a global state.

Like the alias analysis, the design of the binding-time analysis was driven by the typical specialization opportunities that occur in system programs. Some of these situations are well-known in the partial evaluation community and are addressed by standard program transformations (*e.g.,* reduction of primitive applications and procedure residualization).

Other situations require new features to be introduced in the binding-time analysis and the subsequent phases. For example, we have found that certain types of variables, such as integers and pointers, are static in some parts of a procedure and dynamic in others. We handle variables of these types in a flow-sensitive manner, giving one binding-time description per variable per program point.

In addition, our analysis offers elaborate strategies to deal with pointers. More precisely, to exploit thoroughly static information included in partially static objects, it is critical to enable static pointers to refer both to static and/or dynamic objects. Indeed, in the case of pointers to structures, references to its static components should be partially evaluated away, whereas references to its dynamic components should be residualized. This situation implies that a single variable must be able to have two binding-time values, depending on the program point: static if used to dereference static objects, and dynamic if used to dereference dynamic objects.

Interestingly, a similar situation occurs when a pointer may refer to more than one memory cell. If some of these memory cells are static and others dynamic, then the same variable should be described by two different binding-time values. Again, the pointer to the static object should be evaluated while the pointer to the dynamic object needs to be residualized.

Previous analyses are not capable of distinguishing the differences between these cases, and therefore conservatively annotate programs with the most approximate information. For example, other binding-time analyses are flow-insensitive, merging information from all program points together [3, 21, 2]. Further, only one description per variable is available, which may unfortunately residualize some static pointers to static objects [2]. One attempt to resolve this conflict involved splitting structures into static and dynamic parts [2]. This does not work with *module-oriented* partial evaluation, since the interface of a procedure must remain compatible with the interface of the call site.

Another important situation occurs for conditional statements. It has been shown that when conditional statements are processed in a continuation-passing style, the analysis can produce more accurate results [27, 10]. However this strategy might cause code explosion when the test of the conditional statement is dynamic and therefore both branches are residualized.

The analysis of conditional statements via a mixed strategy circumvents the problem. In the case where the test is dynamic, we analyze each branch separately, merge binding-time information at the join point, and then analyze the continuation. On the other hand, if the test is static, then after analyzing each branch separately we analyze the continuation twice, once assuming the true branch will be taken and once assuming the false branch. Since the test is static, the appropriate continuation will be chosen and the useless continuation discarded at specialization time—thus avoiding code explosion.

2.4 Action Analysis

Once the binding-time analysis has been completed, the resulting information is used to determine, for each construct, which specialization *action* (*i.e.,* program transformation) to be performed. This phase comes in addition to the traditional preprocessing phases and has many advantages:

Further compilation of the specialization process. Traditionally, the specializer is directly driven by binding-time information. The complexity of the interpretation of binding-time information depends on the complexity of the binding-time information itself. For realistic languages like the C language, this specialization involves interpreting detailed binding-time information of such objects as data structures. This process can noticeably slow down the specialization phase. In fact, this situation can be improved because binding-time information is available prior to specialization, therefore it can be used to further compile the transformation phase.

More precise definition of the program transformations. Defining specialization actions explicitly forces the designer of the partial evaluator to precisely define the set of program transformations that is needed for a given language. Not only does this provide better documentation to the user, but it also defines in detail the semantics of the specialization phase.

Better separation between preprocessing and specialization. Because action-analyzed programs capture the essence of the specialization process, they can be exploited in many different ways. In fact, in our partial evaluation system, actions can be both interpreted and compiled, and used for compile-time as well as run-time specialization. This range of applications shows the generality of the information expressed by actions.

Specialization actions for C. Four general transformations are potentially defined for each language construct. Action *reduce* (abbreviated *red*) is assigned to an occurrence of a language construct that can be reduced at specialization time. For example, a conditional statement is reduced when its test expression is static. The action *rebuild* (abbreviated *reb*) annotates a construct that needs to be reconstructed but yet includes some static computations. Action *eval* (abbreviated *ev*) is assigned to an occurrence of a language construct that only consists of purely static components. In other words, it can be completely evaluated away at specialization time. At the other end of the spectrum there is action *identity* (abbreviated *id*) which annotates purely dynamic program fragments.

These transformations are fundamental in the sense that they can be used to define other actions. As an example, consider the case of an *explicated* assignment, *i.e.*, an assignment to a variable which is at the same time static and dynamic. The *explication* of such an assignment can be expressed by combining two actions: an action *ev* indicates that the assignment must be completely evaluated so that the value of the variable be available for the static computations; another action *reb* expresses the fact that the assignment also has to be rebuilt so that the variable can be included in a residual program fragment corresponding to dynamic computations.

3 Processing

There can be various back-ends to an action-analyzed program. By back-ends we mean processing phases. They can be divided into two categories: actions can be exploited for either compile-time or run-time specialization.

3.1 Compile-Time Specialization

Traditionally the result of preprocessing is used for compile-time specialization. Just like machine instructions, produced by a compiler, can be interpreted by a simulator or directly executed by a machine, actions can either be interpreted or compiled. Let us describe these two strategies.

Interpreting Actions. An interpreter of actions corresponds to the usual specializer. It consists of dispatching on each action of a program and executing various operations which perform this action. Because information available prior to specialization has been extensively exploited, the specializer is simple, has a clear structure, and is very efficient.

Conceptually, a specializer combines a standard interpreter to perform the static computations, and a non-standard interpreter to reconstruct program fragments corresponding to the dynamic computations. Unlike standard interpretation, programs may sometimes need to be evaluated speculatively. Typically for conditional statements with a dynamic test expression, both branches need to be partially evaluated since the test expression is unknown at specialization time. A mechanism is thus needed to process the branches independently of each other: they must be processed with the same initial store available before considering the branches. This situation requires to make a copy of the store, and reinstall it at a later stage.

One approach to address this problem is to write a specializer which includes a complete interpreter of C programs. A drawback of this approach is the major development effort that it entails due to the syntactic richness of the C language, and also to the wide range of base types and conversion operations between them (which are sometime machine-dependent!).

Another option is to interface a specializer with an existing C interpreter. However, existing C interpreters do not offer a store model which supports speculative evaluation. Implementing this store copying would require the memory of the interpreter to be tagged and would involve a costly memory traversal.

We have developed a third option which allows us to use a standard compiler to process the static computations, thus preserving the semantics of these computations. The first part of this approach consists of flattening the scope of the static variables of a program. Indeed, only static variables can be involved in static computations. To do so, the idea is to rename all the variables of a program so that they can all be global. Of course, this transformation precludes the specialization of recursive procedures when they are partially static. However, system programs do not exploit this language feature. A consequence of this design is that all static data structures can be allocated contiguously and can be easily copied to implement speculative evaluation. Furthermore, invocations of external procedures (*i.e.*, procedures not processed by the specializer) can be done easily since the store layout is compatible.

The other part of our approach consists of encapsulating purely static (*ev*) fragments in C procedures. These C procedures can be directly compiled by some standard C compiler and linked to the specializer. The specializer thus only concentrates on the operations aimed at reconstructing program fragments. In order to process an *ev* fragment, the specializer simply invokes the C procedure which performs the corresponding computations.

The idea of using a standard compiler to perform part or all of the specialization process is not new. Andersen [2] uses a similar approach in his C partial evaluator (C-Mix) by producing generating extensions. However, his store management is more complex in that each data object is indexed by a version number and has an "object function" that can save and restore its value, and compare it to another copy. These operations are aimed at sharing some copies of the same object (*e.g.*, a matrix) between several static stores. However, in our case, module-oriented specialization greatly reduces the need for such optimizations. Indeed, unlike C-Mix which requires a program to be specialized all at once, Tempo can specialize pieces of a program separately. Furthermore, C-Mix includes a symbolic store used, for instance, when procedures are unfolded, or to manipulate pointers to dynamic objects. In contrast, Tempo's specializer only includes the static C store used by *ev* procedures, and unfolding is done as a postprocessing operation for compile-time specialization. This approach greatly simplifies the specializer and does not degrade the quality of the specialized programs.

Notice that the store model described above is general in that it is used both for compile-time and run-time specialization.

Compiling Actions. The natural alternative to interpreting actions is to compile them. The same "run-time" system previously described for specialization is reused. Indeed, *ev* fragments can still be packaged up in procedures, as for action interpretation, and thus be directly treated by the C compiler. The main issue when compiling actions is to compile partially static computations, *i.e.*, to reconstruct residual code. A simple action compiler is an almost trivial task to achieve, as shown by Consel and Danvy [9]. A similar compilation process is also known as generating extension [19, 1, 5, 2]. One difference is that this latter approach is directly based on binding-time information and thus far more complicated than an action compiler.

3.2 Run-Time Specialization

Not only can actions be used to specialize programs at compile time but they can also be utilized to perform specialization at run time. In fact, run-time specialization based on actions is just another way of exploiting this information. Indeed, an action describes how to transform a construct and therefore it also precisely describes the set of possible specialized programs. This set can be formally defined by a tree grammar.

We have developed an abstract interpreter which produces a tree grammar for a given action-analyzed program. As a simple example of what this analysis produces for an action analyzed program, consider the following action tree fragment:

reb(`PLUS`*(id(*`VAR("x")`*), ev(...)))*

Assuming an *ev* fragment of type integer, our analysis then produces the tree grammar rule shown below:

`L` $\longrightarrow$ `PLUS(VAR("x"), int)`

Once produced, the tree grammar is used to generate templates [23], *i.e.*, source code fragments parameterized with "holes" for run-time values. The original program is then transformed to express the various alternatives in the tree grammar and eliminate static computations. For the above tree grammar rule, such a transformation amounts to generating the following template (where the '?' stands for a hole):

```
x + ?
```

Because this transformation is performed at the source level, it is then possible to use a standard compiler to process the templates. As a result, the quality of the compiled templates is as good as the compiler being used. In fact, in our implementation of the run-time specializer, the Gnu C compiler is being used.

One key feature of our approach is that specialization at run-time solely amounts to selecting templates, filling holes in templates with run-time values (the `?` above), and relocating jumps between templates. The simplicity of these operations makes run-time specialization a very efficient process which requires specialized code to be run very few times to amortize the cost of specialization.

A complete description of our approach to run-time specialization is presented in [12].

4 Applications

This section first discusses some generalities regarding the classes of programs which are targeted for specialization. The support for module-oriented specialization is then presented. Finally, typical candidates for specialization in system code are described.

4.1 Classes of Programs Targeted

In recent years, major research projects have been focusing on the design and implementation of operating systems that are both highly-parameterized and efficient. These apparently conflicting goals have led researchers to widen the scope of the techniques which are used in system development. More specifically, programming language techniques have been introduced to perform a critical task, namely, adapting/customizing system components with respect to given parameters. In this context, forms of partial evaluation have become a key technique to develop adaptive operating systems. Examples of such projects include Spin [4], ExoKernel [17], Scout [26], and Synthesis [14, 13].

To validate the applicability of Tempo we have been studying in detail various system components. Some of this work has been done in collaboration with operating system researchers. For example, we have been collaborating with the Synthesis group at Oregon Graduate Institute to apply our approach to file system operations in the Hewlett Packard Unix system. The results of this work are reported in an upcoming SOSP'95 paper [29].

Other areas of current research include inter-process communication (IPC) and remote procedure calls (RPC). In particular, we have been working on fragments of the Chorus operating system [31]. Our goal is to specialize the remote procedure call layer (RPC) with respect to a given RPC stub and server.

These studies suggest the following three observations: there is a need for module-oriented specialization, control flow specialization is important, and there are opportunities to specialize with respect to data flow.

4.2 Module-Oriented Specialization

The need for module-oriented specialization appeared obvious when examining any system code. For example, when studying the communication system of Chorus, it turned out that this component was too large to enable one to reason about potential specializations. We therefore concentrated on a specific layer of the protocol stack: the socket level. When isolating this piece of code, we found a great number of global static variables, some deeply embedded in partially static structures. This situation prompted us to work on a tool capable of determining what parts of the global state was needed for a particular piece of code, and assist the programmer to initialize it when performing specialization.

Furthermore, when specializing an isolated piece of code, the partial evaluation system still needs to reason about the unknown used pieces. More precisely, the pieces external to the one considered have to be understood in terms of their side effects. To this end, we are also developing a declaration language which enables the programmer to specify the effects of external pieces.

Another consequence of studying system code is the use of a demand-driven strategy to develop Tempo. Usually, every language feature is being dealt with in a complete way. That is, all possible cases defined for a language feature are handled. This completeness is needed since no specific application is targeted. However for a language like C, this strategy amounts to being very conservative for a number of its features. When a particular kind of programs is targeted, one can observe that language features are seldomly used in their full generality. This situation arises, for example, in the case of the operation `setjump`, which allows programs to exit non-locally. Although this operation has a very unpredictable effect on the control, it is used in a very structured way in system code. In fact, we have characterized the program patterns corresponding to the way it is used, and for these patterns a treatment is proposed.

4.3 Control Flow Specialization

System programs devote much of their time interpreting data structures. These data structures may contain information of the system state or parameters provided by the user. Like any interpreter, these special-purpose interpreters can be specialized with respect to a given program (here, a regular data structure).

This observation can be illustrated, for example, by the communication system we studied. There are some typical interpreting procedures, like `udp_usrreq(sock, req, ...)`, at the UDP level, which take a user request and dispatch to the corresponding code. There is also much argument checking performed. For example, for each operation on a socket, the procedure `getsock(fdes)` is called and checks whether the file descriptor is not out of bounds. Still, this information, available when the socket is opened, will not change until it is closed. Run-time specialization can be used in such a situation.

In fact, interpretation is really pervasive in an operating system due to the generality of the services it offers. It is common to observe that some operations devote more than a quarter of their conditional computations to interpret user options or parts of the system state. In this respect, this application area is ideal for partial evaluation.

4.4 Data Flow Specialization

Besides the control flow of a program, specialization can also optimize its data flow. In system code, links between data structures are repeatedly interpreted. For example, the same linked list can be traversed many times, and thus, the pointers linking the elements together are dereferenced repeatedly. Specialization can be used in this situation to optimize the code that traverses such data structures with respect to a given instance. It will eliminate the pointer dereferencing operations.

This idea was originally presented by Massalin [30] and named *executable data structures*. He applied this technique to the task scheduler of the Synthesis operating system. It is a routine which is called when a task switch is needed. To perform this task switch, the scheduler saves the registers of the outgoing task (including the program counter), and then — using a global circular task queue — dereferences a pointer to the next task, loads its registers, and jumps to its saved program counter. To obtain an efficient task switch, prologue and epilogue code are dynamically generated for each task, using some hand-written templates in assembly language. This code only saves the used registers of the old task and loads the needed registers of the new one. This optimization is important when dealing with floating-point registers, whose saving and restoring time is expensive.

The same optimization can be performed using partial evaluation. The prologue and epilogue routines can be derived automatically by specializing a generic task scheduler with respect to a given circular task queue. Let us consider, for simplicity, a non-preemptive scheduler expressed in some pseudo-code. This scheduler is called by a task when it wants to give control to the system:

```
task *crt_tsk;

sched() {
  save_regs(crt_tsk);
  crt_tsk=crt_tsk->next;
  load_regs(crt_tsk);
  jmp_to(crt_tsk->PC);
}

task_1(){
  ...
  sched();
  ...
}
```

It is the second command in procedure `sched` which is important. It dereferences the pointer to the next task to be activated. Data flow specialization is targeted towards eliminating such operations.

Once specialized with respect to a given circular list of tasks, the resulting program is:

```
sched_1(){
  save(FP1); save(FP4);
  load(FP2); load(FP3);
  jmp(0x....);
}

task_1(){
  ...
  sched_1();
  ...
}
```

While this example is overly simplified, it still illustrates how specialization can be applied to complex data structures that are repeatedly interpreted by system code.

Finally, note that some situations require both control flow and data flow specializations.

5 Related Work

There are already a number of existing imperative partial evaluators [21, 25, 28, 3, 2]. They are either on-line or off-line partial evaluators, and cover a wide range of imperative programming languages, such as C, Pascal, and Fortran. All of these systems are universal in the sense that they offer general solutions. Also, they are all global, requiring the whole program to be processed in order to perform any optimizations. This severely limits real-size applications. Finally, these systems are limited to compile-time optimizations. Run-time information is ignored.

Tempo addresses these shortcomings. By considering specific realistic applications, we make design decisions which allow us to produce a highly effective tool for certain applications. Being module-oriented, Tempo can be applied to smaller parts of large systems, opening up many new opportunities to apply partial evaluation. As well, including a run-time specializer allows new types of optimizations to be performed.

Recently other forms of run-time specialization have been explored. Engler and Proebsting's approach involves having the programmer manually construct templates which are then compiled into binary code at run time [16]. Leone and Lee's method involves postponing certain compilation operations until run time in order to better optimize programs [24]. These approaches are error-prone due to the need for user intervention, lose efficiency due to the lack of global perspective, and have not been formalized.

We have addressed each of these issues when developing the run-time specializer for Tempo. The process is automatic and formally defined and proven, guaranteeing a relative degree of safety. Since it is based on the Gnu C compiler (GCC), we can port our run-time specializer to any architecture which supports GCC. Finally, efficient code can be created since the templates are all available and compiled before run time, which allows advanced optimizations to be performed.

Current work is also being done on adaptive operating systems [4, 26, 17]. These existing approaches tend to invent new and different technologies in order to provide this adaptiveness.

In contrast, we propose reusing an existing technology, namely partial evaluation, to meet the demands of adaptive operating systems. Our collaboration with Synthetix creates a synergistic effect where both groups benefit from the cross-fertilization [29]. The operating systems group identifies where specialization can be applied, and uses the tools we provide them to perform their adaptive specialization. By applying Tempo to systems programs, our group can continue refining the tools based on the feedback we receive.

6 Conclusion

We have presented an approach to designing partial evaluators for realistic languages and applied it to real-size applications. Our Tempo system is based on a general approach which consists of separating the process into two parts: a preprocessing phase compiles, after a number of static analyses, input programs into actions, and a processing phase which then executes these actions to perform the actual specialization. As we have shown, this has a number of benefits and, in particular, makes it possible to integrate both compile-time and run-time specialization within the same system.

A second key feature which distinguishes Tempo from standard partial evaluators is the fact that this partial evaluator was developed with a particular domain of applications in mind, namely system code. This decision was based on the belief that a universal partial evaluator would be too general to perform the specific optimizations desired. We have accordingly opted for a bottom-up approach which consists of studying the opportunities for specialization of specific applications and then making design decisions based upon these studies.

Our preliminary experiments, on operating system code, are encouraging. They have shown that partial evaluation can indeed be used to migrate current operating systems towards a new range of adaptive operating systems. These early results have also brought to our attention new concepts and techniques, like module-oriented specialization, which we found necessary to perform the realistic program transformations we desired.

Although much remains to be done, we feel that Tempo is a significant step towards making partial evaluation a practical tool for programming in the large.

Acknowledgements

The authors would like to thank Olivier Danvy and Barbara Moura for their comments on drafts of this paper. Thanks are also due to Sandrine Chirokoff, Julia Lawall, Anne-Françoise Le Meur, Jérôme Picault, Vincent Piederriere, and Scott Thibault for their help with the implementation of Tempo.

References

1. L.O. Andersen. – Self-applicable C program specialization. – In C. Consel, editor, *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61. Yale University, 1992. – Research Report 909.
2. L.O. Andersen. – *Program Analysis and Specialization for the C Programming Language.* – PhD thesis, DIKU, University of Copenhagen, May 1994. – DIKU Technical Report 94/19.

3. R. Baier, R. Glück, and R. Zöchling. – Partial evaluation of numerical programs in Fortran. – In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 119–132, 1994.
4. B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Gün Sirer. – SPIN – an extensible microkernel for application-specific operating system services. – Technical Report 94-03-03, University of Washington, Seattle, Washington, February 1994.
5. L. Birkedal and M. Welinder. – Partial evaluation of Standard ML. – Master's thesis, DIKU, University of Copenhagen, 1993. – Research Report 93/22.
6. D.R. Chase, M. Wegman, and F. Kenneth Zadeck. – Analysis of pointers and structures. – In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. – ACM SIGPLAN NOTICES, 25(6), June 90.
7. C. Consel. – Polyvariant binding-time analysis for higher-order, applicative languages. – In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, 1993.
8. C. Consel. – A tour of Schism: A partial evaluation system for higher-order applicative languages. – In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, 1993.
9. C. Consel and O. Danvy. – From interpreting to compiling binding times. – In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, 1990.
10. C. Consel and O. Danvy. – For a better support of static data flow. – In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 496–519. ACM, Springer-Verlag, 1991.
11. C. Consel and O. Danvy. – Tutorial notes on partial evaluation. – In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.
12. C. Consel and F. Noël. – A general approach for run-time specialization and its application to C. – In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, 1996. – To appear.
13. C. Consel, C. Pu, and J. Walpole. – Incremental specialization: The key to high performance, modularity and portability in operating systems. – In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, 1993. – Invited paper.
14. C. Consel, C. Pu, and J. Walpole. – Making production OS kernel adaptive: Incremental specialization in practice. – Technical report, Oregon Graduate Institute, Portland, Oregon, 1994.
15. M. Emami, R. Ghiya, and L.J. Hendren. – Context-sensitive interprocedural points-to analysis in the presence of function pointers. – In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256. ACM Press, June 1994. – ACM SIGPLAN NOTICES, 29(6), June 94.
16. D. R. Engler and T. A. Proebsting. – DCG: An efficient, retargetable dynamic code generation system. – In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM Press, November 1994.
17. D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. – Exokernel: An operating system architecture for application-level resource management. – In *Proceedings of the*

ACM Symposium on Operating Systems Principles, 1995. – To appear.

18. A.M. Erosa and L.J. Hendren. – Taming control flow: A structured approach to eliminating goto statements. – In *Proceedings of the IEEE 1994 International Conference on Computer Languages*, May 1994.
19. A.P. Ershov. – On the essence of translation. – *Computer Software and System Programming*, 3(5):332–346, 1977.
20. N. D. Jones, P. Sestoft, and H. Søndergaard. – Mix: a self-applicable partial evaluator for experiments in compiler generation. – *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
21. N.D. Jones, Carsten Gomard, and Peter Sestoft. – *Partial Evaluation and Automatic Program Generation.* – Prentice Hall International, International Series in Computer Science, June 1993.
22. N.D. Jones and S.S. Muchnick. – *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages (Lecture Notes in Computer Science, vol. 66).* – Springer-Verlag, 1978.
23. D. Keppel, S. Eggers, and R. Henry. – Evaluating runtime compiled value-specific optimizations. – Technical Report 93-11-02, University of Washington, Seattle, Washington, 1993.
24. P. Lee and M. Leone. – Lightweight run-time code generation. – In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, 1994.
25. U. Meyer. – Techniques for partial evaluation of imperative languages. – In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (ACM SIGPLAN NOTICES, 26(9), September 1991)*, pages 94–105, 1991.
26. A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, T.A. Proebsting, and J.H. Hartman. – Scout: A communications-oriented operating system. – Technical Report 94-20, The University of Arizona, Tucson, Arizona, 1994.
27. F. Nielson. – A denotational framework for data flow analysis. – *Acta Informatica*, 18(3):265–287, 1982.
28. V. Nirkhe and W. Pugh. – Partial evaluation and high-level imperative programming languages with applications in hard real-time systems. – In *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992*, pages 269–280. ACM, 1992.
29. 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 *Proceedings of the ACM Symposium on Operating Systems Principles*, 1995. – To appear.
30. C. Pu, H. Massalin, and J. Ioannidis. – The Synthesis kernel. – *Computing Systems*, 1(1):11–32, Winter 1988.
31. M. Rozier, V. Abrassimov, F. Armand, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. – Overview of the Chorus distributed operating system. – In *Workshop on Micro-kernels and Other Kernel Architectures*, pages 39–70. USENIX, May 1992.
32. E. Ruf. – Context-insensitive alias analysis reconsidered. – In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995. – ACM SIGPLAN NOTICES, 30(6), June 1995.
33. R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S.-W. Liao, C.-W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy. – SUIF: An infrastructure for research on parallelizing and optimizing compilers. – *ACM SIGPLAN NOTICES*, 29(12):31–37, December 94.

34. R.P. Wilson and Lam. M.S. – Efficient context-sensitive pointer analysis of C programs. – In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, June 1995. – ACM SIGPLAN NOTICES, 30(6), June 1995.