N. d'ordre : 1810

THÈSE

présentée devant

l'Université de Rennes 1

pour obtenir

le grade de Docteur de l'Université de Rennes 1
Mention : Informatique
École Doctorale : Sciences Pour l'Ingénieur
Institut de Formation Supérieure en Informatique et Communication

par

Luke HORNOF

Titre de la thèse

Static Analyses for the Effective Specialization of Realistic Applications

Soutenue le 27 juin 1997 devant la commission d'examen

MM.	Jean-Pierre Banâtre	Président
	Charles Consel	Directeur de thèse
	Pierre Jouvelot	Rapporteur
	Torben Mogensen	Rapporteur
	Thomas Jensen	Examinateur

Acknowledgments

I am grateful to all the members of the COMPOSE group for helping me achieve this work. I would like to thank my advisor, Charles Consel, who created the initial research environment and provided a central focus for this work, both of which were necessary for it to be accomplished. Endless discussions with Jacques Noyé provided key insights. Conversations with Bárbara Moura, contrasting our different approaches to similar problems, also taught me a lot. Many other people worked on the design and implementation of Tempo, most notably François Noël and Nic Volanski. Expertise with systems applications was largely provided by Gilles Muller.

I would also like to thank the research community in general, which helped me in a variety of ways. Valuable lessons were learned from researchers at the Oregon Graduate Institute, such as Calton Pu, who generously offered his advice on many occasions. Interaction with the professors and students at DIKU, University of Copenhagen, and with Olivier Danvy, at the University of Aarhus, provided many useful discussions. Peter Lee, at Carnegie Mellon University, has been a reference for many years.

Other people assisted with other aspects of my thesis, such as the written document and my thesis defense. Julia Lawall provided valuable feedback on both the content and the style of early drafts of the text. Jean-Pierre Banâtre agreed to be president of my defense jury. A special thanks to Pierre Jouvelot, Torben Mogensen, and Thomas Jensen, who not only provided in-depth comments on later versions of the text, but also agreed to be in my defense jury. This entire document was translated into French single handedly by Gilles Lesventes. Subsequent comments on the French translation were given by Remi Douence, Ronan Gaugne, and Valérie Gouranton.

Lastly, I would like to thank my family for their constant support, both emotionally as well as with practical matters. In particular I would like to thank my father, who helped me understand the importance of education, and to whom I would like to dedicate this thesis.

Contents

List of Figures

Chapter 1

Introduction

This dissertation shows how program specialization can automatically and effectively optimize existing, realistic applications. Previous studies have demonstrated that specializing large applications, such as systems programs, produces impressive speedups, but specialization was done by hand [Mas92, MP89, PMI88]. Such an approach has shown to be too time consuming, tedious, and error prone to be systematically and methodologically applied. The key is to perform these specializations *automatically*. Automatic specialization techniques have been studied for some time, and many theoretical and practical advances have been made [And94, BW93a, BM90, Con93a, DRVH95, JGS93a, Mog88]. Unfortunately, current limitations prevent existing specialization technology from effectively specializing most existing, realistic applications. Rather, applications for which automatic specialization has proven effective are typically relatively small and simple, handwritten to specialize well, or written in domain-specific or less widely used programming languages.

Our approach to solving this problem starts with an extensive study of a class of existing applications, namely systems applications [CPW93, PAB+95, VMC96a, VMC+96b]. Such programs are complicated. They are written in a rich language such as C and their behavior is complex since both modularity and efficiency are important. We explore the limits of existing specialization technology and identify the specific missing features which would allow automatic, effective specialization of such programs. We address these missing features and introduce new static analysis that accurately treat the common program patterns in existing systems applications. Specific features include:

Flow sensitivity. A different analysis description is computed for each update statement.

Context sensitivity. A function is analyzed with respect to the specific analysis context of each call site.

Return sensitivity. A different analysis description is computed for the side-effects and the return value of a function.

Use sensitivity. Different uses of a variable may have different analysis descriptions.

Flow sensitivity has been exploited in other static analyses [MRB95], but we incorporate it for the first time into the analyses of a specializer. Context-sensitive analyses have been developed in the context of partial evaluation for simpler languages, such as functional languages [Con93b], but we develop new aspects in order to treat imperative languages. Additionally, treating existing, realistic systems programs reveals the need for the last two features, return and use sensitivity, both of which are completely new concepts.

To develop an analysis containing all these features, we introduce an approach that fundamentally differs from previous work in several respects, including:

- Two-phase binding-time analysis.
- New program transformations.

Our novel approach separates the different tasks performed by a binding-time analysis into two distinct phases, a binding-time phase followed by a transformation phase. This separation simplifies and clarifies the design of the analysis. Additionally, this approach allows us to express new program transformations, such as evaluating and residualizing the same program construct.

Our analyses support specialization at both compile time and run time. We integrate our analyses into a partial evaluator and apply it to a variety of existing applications, including a commercial operating system component. We obtain significant speedups for both compile-time and run-time specialization.

In the rest of this chapter, we will give an overview of partial evaluation, a program specialization technique which transforms a general program into a special-purpose one. We start with a small example and show the different ways in which specialization can be performed. Then, we summarize and asses existing partial evaluators and consider the applications they treat. This is followed by a discussion of existing systems programs and a summary of the specific analysis features necessary to effectively specialize these programs.

1.1 Partial Evaluation

Partial evaluation is a program transformation that takes a program and part of its input, and generates a new program which exploits these input values. Since the new program integrates these input values, it is a special-case of the initial program, and is therefore considered *specialized*. Reducing the generality of a program allows calculations that depend solely on the provided input values to be performed during specialization. The resulting specialized program only contains the remaining calculations that depend on the unavailable input values.

The goal is that the specialized program should contain fewer computations and therefore be more efficient than the original program.

1.1.1 Online vs. Offline

There are two types of partial evaluators: *online* and *offline*. An online partial evaluator produces a residual program in one single stage [JGS93a, Mey91, WCRS91]. It can be thought of as a non-standard interpreter, in which the state maps a known variable to a value and an unknown variable to a textual representation of the variable. Given this state, the non-standard interpreter then processes the program, computing values for constructs that contain only variables with known values, and combining textual representations for constructs that depend on some unknown value. These combined textual representations ultimately produce the residual program.

In contrast, the offline approach divides specialization into two stages, an analysis stage which determines the transformations followed by a specialization stage, which performs them [JGS93a, Con93d, And94]. The analysis stage is not provided with the known input values, only an *abstract description* of them. Based on this limited information, the analysis calculates a transformation for each construct in the program. This information is then passed to the specialization stage, which is also provided the actual input values, and the transformation corresponding to each construct is performed, producing the residual program.

Because the online approach takes actual input values into account to determine a transformation, it may produce better results than the offline approach. For example, consider a conditional statement which has a known test, a truth branch which assigns a variable a known value, and the false branch which assigns the same variable an unknown value. If the test is true, an online partial evaluator can use this value to deduce that the variable will be assigned the known value in the truth branch and use this information following the conditional statement. An offline approach does not have the actual input values during the analysis stage when transformations are determined, and therefore cannot determine whether the truth or the false branch is taken. An approximation is made which assumes the variable is assigned an unknown value.

On the other hand, there are a number of advantages to the offline approach. The specialization stage can exploit global program information generated during the binding-time analysis, which is essential when dealing with existing, realistic programs. For example, treating an imperative language requires global information such as relating the definition of a variable with its uses. Further, analyses may be required to accurately or correctly treat complex language constructs such as pointers, arrays, and data structures. Another advantage is that, since transformations are determined ahead of time, the offline specialization stage is simpler and faster. Therefore, the offline approach is critical for efficient specialization.

1.1.2 Compile-Time vs. Run-Time

The goal of partial evaluation is to minimize the time it takes to execute a program. The time before a program is executed is called *compile time*, while the time a program is executing is called *run time*. *Compile-time specialization* refers to specializing a program with respect to input values that become available before the program is executed. The residual program is compiled and the resulting object code can then be executed. The time between the moment the known input values become available and the moment the specialized program can be executed is substantial, since it includes both specialization time and compilation time. This is acceptable for compile-time specialization since no extra run-time cost is introduced.

For certain applications, however, the known input values only become available at run-time. Specializing a program with respect to these values during the execution of the program is known as *run-time specialization* [CN96, NHCL96]. Run-time specialization is only beneficial when the time saved by executing the more efficient residual program is greater than the time needed to generate that residual program. Therefore, it is critical to minimize the cost of specializing programs at run time.

1.2 A Concrete Example

In order to clarify these ideas, we take a small example and show how it is specialized. First we give an example of online specialization. Then we show the offline approach, consisting of the binding-time analysis followed by specialization. Finally we show how the offline approach leads to efficient run-time specializers.

1.2.1 Online Specialization

Consider the following dot product function which computes the dot product of two vectors of length `size`.

```
int dotproduct(int size, int u[], int v[])
{
  int i;
  int res;

  res = 0;
  for (i = 0; i < size; i++)
    res = res + u[i] * v[i];
  return res;
}
```

A normal call to this function provides a value for all three inputs, such as `size = 5`, `u[] = {4,8,3,2,9}`, and `v[] = {3,3,4,5,5}`. With these arguments, full evaluation of the function produces the appropriate dot product, `103`. On the other hand, if only some of these inputs are provided, the function cannot be fully evaluated. It can, however, be partially evaluated with respect to the available input values. Assuming the inputs `size` and vector `u[]` are provided with the same values as above, online partial evaluation generates the following residual program:

```
int dotproduct_1(int v[])
{
  int res;

  res = 4 * v[0];
  res = res + 8 * v[1];
  res = res + 3 * v[2];
  res = res + 2 * v[3];
  res = res + 9 * v[4];
  return res;
}
```

The result of partial evaluation is not a dot product, but a specialized program which computes a dot product (for vectors of size 5 where the first vector contains the values `{4,8,3,2,9}`). We expect this specialized version of dot product to be more efficient than the original, general version, yet still produce the same result. All of the computations that depend on `size` and `vector u[]` are performed during specialization and therefore not present in the residual program. For example, the loop is unrolled since this only depends on `size`, and each occurrence of `u[i]` is instantiated with its corresponding value. Indeed, evaluating the specialized function `dot_product_1` with input `v[] = {3,3,4,5,5}` produces

the same result as we saw earlier, *i.e.* `103`.

By producing a more efficient residual program specialized with respect to these input values, the multiple calls to `dot_product_1` are more efficient. In practice, performing such a specialization might be useful if, for example, the original function is called many times with the same first two input values, as is the case with matrix multiplication.

1.2.2 Offline Specialization

Now let us see how the offline approach specializes the same program. Offline specialization consists of two stages, the analysis stage followed by the specialization stage. Once the program is analyzed, there is a number of different ways it can be specialized. First we present traditional specialization, followed by a faster specializer called a *generating extension*. Finally, we show how a generating extension can be adapted to efficiently perform run-time specialization.

Binding-time Analysis

The primary analysis of the offline approach is *binding-time analysis*, which determines a transformation for each construct. Binding-time analysis is abstract interpretation over two abstract values, or *binding times*, *static* and *dynamic*. *Static* is the abstraction of a known values and *dynamic* is the abstraction an unknown value. An abstract description of the known input values is provided to the analysis, with which an initial abstract state is created. The binding-time analysis uses this state to process the program, annotating each construct in the program with a binding time. Constructs that only depend on static values are considered static, while those that depend on some dynamic value are considered dynamic.

Let us reconsider the dot product example to see how a binding-time analysis annotates it. The abstract description of the input parameters specifies that variable `size` and vector `u[]` are static and that vector `v[]` is dynamic. The binding-time annotated program is as follows:

```
int dotproduct(int size, int u[], int v[])
{
  int i;
  int res;

  res = 0;
  for (i = 0; i < size; i++)
    res = res + u[i] * v[i];
  return res;
}
```

Italics indicates the construct is static and **boldface** indicates it is dynamic. Since static constructs, such as the loop index operation $i = 0$ and the array access *u[i]*, only depend on static values, they can be performed at specialization time. The dynamic constructs, such as the **for** loop or the assignments to variable **res**, cannot be evaluated at specialization time and must therefore be present in the residual program. In this example, the first assignment to variable **res** is residualized to accommodate the residualized use of **res** in the second assignment.

Specialization

Starting with the binding-time annotated program above, an offline specializer performs transformations based on the binding-time annotations. Constructs which are completely static, such as *u[i]*, are evaluated. Completely dynamic constructs are residualized, such as **return res**. Dynamic constructs that contain static subcomponents are transformed by evaluating the static subcomponents and residualizing the dynamic subcomponents. For example, the **for** loop is unrolled since its test is static, residualizing its dynamic body. The array access **v**[i] is residualized with its index instantiated with a constant value. This process produces the following residual program:

```
int dotproduct_1(int v[])
{
  int res;

  res = 0;
  res = res + 4 * v[0];
  res = res + 8 * v[1];
  res = res + 3 * v[2];
  res = res + 2 * v[3];
  res = res + 9 * v[4];
  return res;
}
```

The specialized program is similar to the online version seen in Section 1.2.1. Here, however, the first assignment to variable `res` is residualized. As mentioned earlier, the offline approach does not always produce results as precise as the online approach. In this example, variable `res` contains the static value `0` during the first iteration of the loop and dynamic values for subsequent iterations. The binding-time analysis makes an approximation, assuming variable `res` is dynamic during all iterations of the loop. This in turn causes all uses of variable `res` to be residualized, which subsequently forces the initial assignment to `res` to be residualized as well. Although this residual program is less optimal than an online version, the difference is not significant. In fact, compiling both residual programs with an optimizing compiler would generate identical executable programs.

Despite this disadvantage, an offline partial evaluator has an important advantage. Since the transformations are pre-computed, the specialization stage is more efficient. The offline specializer simply interprets a binding-time annotated program and triggers transformations based on the annotations.

Generating extension

An even more efficient specializer can be produced by eliminating the overhead of interpreting the binding-time annotated program. This is achieved by explicitly encoding the control flow of specialization into the specializer, creating a dedicated specializer known as a *generating extension* [JGS93a, GJ95]. Whereas an offline specializer takes an annotated program and static input values and produces the specialized program, a generating extension only takes the static inputs to produce the specialized program.

A generating extension can be automatically produced from a binding-time annotated program. For example, the generating extension produced from the binding-time annotated dot product example is as follows:

```
int gen_dotproduct_SSD(int size, int u[])
{
  int i = 0;

  printf("int dotproduct_1(int v[])\n");
  printf("{\n");
  printf("  int res;\n");
  printf("  \n");
  printf("  res = 0;\n");

  for(i = 0; i < size; i++)
    printf("  res = res + %d * v[%d];\n", u[i], i);

  printf("  return res;\n");
  printf("}\n");
}
```

This program takes the static input values, `size` and `u[]`, and directly produces the specialized `dot_product_1` function. The first group of `printf` statements outputs the function header, its formal arguments, the begin block, and the declaration and definition of variable `res`. The last group of `printf` statements outputs the return statement and the end block. These groups of statements *always* output the same lines of code for each specialization, independent of the static input values. Therefore, the first and last lines of every specialization will necessarily be exactly the same.

The line output by the `printf` statement in the middle, however, changes with respect to the static input values `size` and `u[]`. First, since it is in a loop, the line of code is output `size` times. Additionally, the contents of the line of code output are different each time. The assignment statement generated by this `printf` statement contains the static values computed by `u[i]` and `i`, which are inserted into their proper place as indicated by the two `%d` escape characters in the control string.

Target-Generating Extension

All of the specializers considered so far produce residual programs that are written in the same language as the source program. A generating extension that generates source code, such as the one defined above, is called a *source-generating extension*. A *target-generating extension*, on the contrary, produces residual programs that are written in a target language, which can be a different language than that of the source program [CN96].

A target-generating extension is necessary to achieve efficient run-time specialization. We have already mentioned that for run-time specialization, it is

critical to minimize the time between the moment the known input values become available and the moment the specialized program is ready to be executed. Recall that this time includes specialization time and compilation time. We have already seen that a generating extension minimizes the cost of specialization. Here, we show how using a target-generating extension minimizes the cost of compilation.

The key aspect is choosing executable object code as the target language of the target-generating extension. This completely eliminates the run-time compilation time, as the residual program is in executable form. Another way of viewing this is to consider that the order of specialization and compilation are being reversed. Instead of specializing and then compiling, this approach compiles and then specializes.

Automatically creating a target-generating extension that produces executable residual programs involves, at compile time, producing the basic building blocks from which specialized programs are composed, along with the target-generating extension which uses them. Each basic building block is called *template*, which is a fragment of code parameterized by *holes*. A hole is a placeholder introduced to represent a missing fragment of code. These templates must be written in the target language. We can automatically create templates in the target language by constructing the corresponding templates in the source language and translating them into the target language using a compiler.

For the dot product example, there are three *templates*, one for each of the three "group of `printf` statements" in the compile-time generating extension. The source templates are:

t_1
```
int dotproduct_1(int u[])
{
  int res;

  res = 0;
```
t_2
```
  res = res + h0 * v[h1] ;
```
t_3
```
  return res;
}
```

Each source template is basically the concatenation of contiguous `printf` control strings in the corresponding source-generating extension. The first source template, t_1, represents the initial lines of the residualized program, source template t_2 represents the middle line, and source template t_3 represents the last lines. Source template t_2 contains two holes, representing the static values that cannot be determined until specialization. These holes, represented here by h_0 and

h_1, correspond to the `%d` escape characters in the control strings in the source-generating extension.

We now turn to the target-generating extension itself, which manipulates the executable versions of these templates to produce an executable residual program. The target-generating extension has the same structure as the source generating extension—the difference is that target code is generated instead of source code. Again, hole values are static values computed during specialization. The command `output_template` takes the name of the template and the hole values with which the template should be instantiated, and outputs the instantiated target template. The target-generating extension for dot product is as follows:

```
int target_gen_dotproduct_SSD(int size, int u[])
{
  int i;

  output_template(t1);
  for (i = 0; i < size; i++) {
    h0 = u[i];
    h1 = i;
    output_template(t2, h0, h1);
  }
  output_template(t3);
}
```

In this example, template t_1 is output, followed by a loop which outputs and instantiates template t_2 as many times as `size` indicates, followed by template t_3. The first and last lines, represented by templates t_1 and t_3 are the same for every specialization. Only the middle line, represented by template t_2, changes with respect to the static values.

This approach produces a specialized program that is already executable which means that it can immediately be invoked. Doing so is critical for run-time specialization. On the other hand, since templates are compiled before they are instantiated and composed, they cannot be optimized with respect to the actual static values that only became available during specialization. Therefore, the resulting specialized programs may not be as optimized as the same program specialized before being compiled.

1.3 Partial Evaluation and Applications

Now we have the basic idea of what partial evaluation is and how it works. Our goal is to apply partial evaluation to existing, realistic programs. In this section, we consider existing partial evaluators and explain why they are not

capable of effectively treating such programs. We show the specific shortcomings of these systems and explain how the new analyses we introduce overcome these deficiencies.

Before we elaborate on these issues, let us describe what we mean by a *realistic* program. First of all, the program must do something useful, like solve a real problem or perform some service that people are actually interested in. This excludes "toy" examples. Further, we exclude programs which are only of interest to the partial-evaluation community. Many promising results have already been produced in this area. The goal is to see if these results are applicable in other areas. Therefore, we consider programs from other domains.

Let us also explain why we are considering *existing* applications. If existing systems do not have to be completely rewritten, specialization techniques can be more widely applied and researchers in other fields can benefit from specialization technology. For example, even if specialization opportunities are identified, it is unlikely that programmers will exploit them if it requires re-implementing their system in another language. Secondly, comparing an existing unspecialized program with its specialized version provides a fair comparison to assess the real worth of specialization. If the performance of an existing program is at all important, the program will already be somewhat optimized. Therefore, speedups due to specialization will not come from simply removing some obvious and trivial calculations.

With these in mind, let us now look at the existing partial evaluators.

1.3.1 Existing Partial Evaluators

An early predominant focus of partial evaluation research was self-application and compiler generation [AK82, CK91, GJ89, JS80, JGS93a, Jør92]. These systems include both online and offline approaches, and treat functional, imperative, as well as logic programming languages. In general, these systems are more exploratory or pedagogical in nature and were not intended to treat realistic applications. Rather, applications typically consisted of programs like interpreters, parsers, and pattern matchers.

Other partial evaluators were designed to treat more realistic applications. FUSE [AWS91, BW90, WCRS91] and Blitzkrieg [BS94, Sur95], for example, are online systems which report significant speedups for numerical programs such as the n-body problem, particle interaction, and circuit simulation. Their studies reveal that specialization opportunities do exist for these types of programs. Many of these programs are data-independent, which means the control flow of the program does not depend on the input data which it processes. Therefore, these programs can be specialized with respect to the control flow, which not only eliminates constructs such as loops, but additional optimizations are triggered as well.

But both of these partial evaluators treat a subset of Scheme. Existing numerical programs are not written in Scheme. Rather, the programs were rewritten into Scheme so that they could be specialized. The speedups were determined by comparing the unspecialized Scheme programs with specialized Scheme programs. In some cases, the unspecialized was not intended to run fast. For example, one numerically intensive program tested was written primarily for people to understand. Therefore, the program intentionally did not take advantage of special cases, such as exponents of zero or one [BW90].

Many realistic applications are implemented in object-oriented languages. The Simili partial evaluator was designed to treat a small subset of an object-oriented language, with the goal of developing techniques that could scale up to existing object-oriented languages [MS92]. Applications targeted included window systems and file managers.

Simili is an online partial evaluator. Even though the language treated was a simple subset of an object-oriented language, the complexity of an online approach caused the specializer to be impractical to treat any large example. The online partial evaluator needed to maintain too much information during specialization. These observations reinforce our intuition that offline specialization, in which this extra information is confined to the analysis phase, is better suited for complex languages. Simili obtained reasonable speedups, nonetheless, for Ackerman's function and a small interpreter for a Turing machine.

These examples show that existing online techniques that can handle realistic applications can only treat a simple language, while techniques that treat a more complex language can only handle simple programs. Again, this suggests that offline techniques are better suited for treating realistic applications written in a realistic language. A number of works have chosen to pursue this direction.

Numerical programs are usually written in Fortran. Two offline partial evaluators treat Fortran: SFAC [BF93, BF96] and FSpec [BGZ94, GNZ95, KKZG94]. SFAC has been applied to existing, industrial strength programs, such as programs that are used in nuclear power plants or telecommunications satellites. However, the goal of SFAC was to create a tool which would help users understand a large, complex system. To see what parts of a program depended on a certain input variable, the user indicated that input variable was *dynamic*. Specializing the program with this information would residualize all the parts of the program that depended on this input variable. As these transformations were not intended to improve performance, benchmarks were never performed.

FSpec, on the other hand, was specifically aimed at improving the performance of Fortran programs. The numerical applications treated examples such as Fourier transform, cubic spline interpolation, and function integration, obtaining significant speedups. FSpec, however, only handles a subset of Fortran and the applications treated were relatively small. Our goal is to scale up this approach to treat larger, more complicated examples such as operating systems programs.

C-Mix [And94] is an offline partial evaluator with an approach similar to FSpec. However, it treats the full C programming language, including data structures and pointers. Further, it uses efficient constraint-based analyses and a generating extension to efficiently process large, realistic programs. Speedups were obtained for a lexical analysis, a numerical program, and a graphics program, but these applications were written with partial evaluation in mind so that they would specialize well. One existing application, an ecological modeling program, was also specialized. Minor changes to this program were necessary, however, in order to achieve the reported speedups. As our studies show that the static analyses in C-Mix are not capable of effectively treating systems code, it is unclear as to how much effort was needed to write or modify these applications in order for them to specialize well. Nevertheless, C-Mix appears the best suited of all existing partial evaluators to effectively treat existing, realistic programs such as systems applications.

1.3.2 Systems Applications

Although automatic program specialization has achieved significant speedups for a wide range of applications, none of these applications were realistic, existing programs. We show these types of programs can, in fact, be successfully specialized. Specifically, we apply partial evaluation to existing operating systems programs. These programs are large, complex programs written in C. Additionally, they contain both compile-time and run-time specialization opportunities. Therefore, we choose an offline approach, as it is better suited for treating complex applications and it permits efficient run-time specialization.

Empirical studies indicate that existing offline partial-evaluation technology, such as FSpec or C-Mix, is not sufficiently advanced to effectively specialize systems software. This is due to a lack of accuracy of binding-time analyses when dealing with certain basic features of C programs, such as pointers and data structures, as well as dealing with the complicated program patterns found in systems programs. Specifically, we have found that *flow*, *context*, *return*, and *use sensitivity* are necessary in the binding-time analysis [HN97, HNC96]. Let us try to characterize these programs based on our studies, and then show how these analysis features are needed to effectively treat systems applications.

Systems Applications

Systems applications are large applications which provide a wide variety of services. We have identified two aspects which are common to most systems applications: generality and modularity. Generality permits a system to provide many different services, and modularity is critical for structuring and managing the complexity of a system.

Generality is important because it permits code reuse. In the context of operating systems, library calls are generic services that provide a family of functionalities, permitting a number of related services to be implemented by the same function. Many different systems components can then use the same library functions. Therefore, these functions are highly parameterized, providing generic functionalities which can be instantiated according to the needs of each individual component.

Generality is also the result of enforcing strict interfaces. For example, for both security reasons and design simplicity, it is important to minimize the interface between user processes and the operating system. This is done by minimizing the number of system calls available to the user. In order to provide all of the necessary services with a minimum number of functions, a few, general system calls are provided.

Creating a *modular* system has a number of software engineering advantages. A module provides an independent, well-defined service. The ability to compose modules allows more complex services to be easily created. For example, a monolithic system can be organized as separate subsystems. These subsystems are modules that can then be composed horizontally to build a full system. Modules can also be composed vertically, such as with software layers for implementing protocols.

Separate components of a system need to communicate with the other system components. This communication is typically performed by encapsulating useful system information into some form of system state. The execution of each component is guided by the state. This state is passed from component to component—each component accesses and modifies the state values relevant to that component. Ultimately, the contents of the state guides the overall execution of the system.

Another way in which a module communicates is by providing some sort of feedback to its callee. A common technique for providing this feedback is to return some sort of error status, indicating whether the desired functionality was completed successfully, or whether some error occurred which caused execution to abort prematurely. These error status values are propagated back through multiple functions, and are eventually used by the function that initiated the call to decide how to proceed.

Opportunities for Specialization

Systems programs contain numerous opportunities for specialization. These opportunities stem from the necessary generality and modularity of the system, which introduce known values in several ways.

We have seen that there are many parameterized generic functions, such as library calls and system calls, which are used in a wide variety of different contexts. Each of these contexts typically contains a number of known values. Specializa-

tion permits these known values and all of the computations which depend on them to be eliminated. For example, a library function that copies memory takes a vector of strings, but it is often invoked with a vector of size one [VMC96a].

The composition of modules also creates opportunities for specialization. Once a set of modules are composed, either vertically or horizontally, the specific use of a general module becomes known. This introduces known values that can be exploited by a partial evaluator. By eliminating the overhead introduced by the interface, the composition of the modules can be streamlined to only contain their essential computations. In a Remote Procedure Call, for instance, the Internet Protocol layer can support both UDP and TCP. In cases where only UDP is used, retransmission buffers can be eliminated [VMC96a].

During the typical execution of a systems program, the information communicated between modules also contains many known values. The state, which guides the behavior of the system, typically contains both known values that permits modules to be specialized. A file system state, for example, may contain information specifying that file accesses are exclusive and sequential. Knowing these values allows the end of file check to be optimized as well as a number of other tasks to be eliminated [PAB+95]. The feedback with which modules report static error values is also known during specialization in certain cases.

1.3.3 Offline Analyses Applied to Systems Applications

Now that we have an idea of the types of programs we want to treat, let us look at the analyses used to treat these programs. Existing binding-time analyses, such as those used in FSpec or C-Mix, are not precise enough to exploit all of the specialization opportunities that exist in systems programs. This is because their analyses are not flow, context, return, or use sensitive. For each of these features, we first explain how an analysis without the particular feature loses precision, and then how an analysis with the feature does not. We then give a specific examples of how these features are necessary to treat operating systems code.

Flow sensitivity

A flow-insensitive binding-time analysis has only one global binding-time state for the entire program, known as a binding-time division. If a variable is assigned a dynamic value anywhere in the program, then it is considered dynamic everywhere.

Flow sensitivity, however, computes a different analysis description for each update statement [HN97]. This allows a variable that is assigned a static value to be considered static, even if the variable is previously or subsequently assigned to a dynamic value. The need to carefully structure a program to obtain a good binding-time division, as has previously been the case, is thus eliminated.

The main problem in treating systems programs with a flow-insensitive analysis is that the binding-time division is a global division. A global binding-time division is sufficient for small functions, which may only contain a few variables or define each variable only once. But this does not work for large or complicated programs. Inevitably dynamic values leak into the division and cause a global loss. With a flow-insensitive analysis, the only way to avoid this loss is to carefully write the application keeping binding-time division in mind. A flow sensitive analysis removes this restriction and allows existing, realistic programs to be treated.

Context sensitivity

In a context-insensitive analysis, functions are analyzed with respect to a single binding-time call context, which is an approximation of all call contexts. If one call context has many static values and another call context has only a few static values, both calls will use a function analyzed with respect to the few static values. Analyses that analyze functions with respect to a single call context are also referred to as *monovariant* [Hen91, JGS93b].

On the other hand, context sensitivity permits functions to be analyzed with respect to the specific state of each call, allowing the different static values in each state to be exploited by each call [HN97]. Even if calling contexts contain a different number of static values, each call can exploit the specific static values of its specific context. Analyses that analyze functions with respect to multiple call contexts are also referred to as *polyvariant* [Con93c, JGS93b].

We have seen that systems code contains many parameterized generic functions, called by a variety of system components in a wide range of call contexts. These contexts contain static values that can be exploited by a partial evaluator. But a context-insensitive analysis cannot fully exploit all of the static values. Rather, a value is only considered static if it is static in all call contexts. A context-sensitive analysis, on the other hand, is capable of exploiting all of the static values.

A similar case arises with the composition of modules. Modules are composed by arranging the manner in which modules invoke other modules. If a function only appears in one configuration, it will have one single call context. But if the function appears multiple times, it is likely to have different call contexts. In this case, a context-insensitive analysis again only considers the least precise call context, whereas a context-sensitive analysis considers each context separately.

Return sensitivity

A return-insensitive analysis computes the same binding-time value for the return value and the side-effects of a function. If a function has a static return value but contains dynamic side-effects, the return value is also considered dynamic.

Return sensitivity permits the return value of a function to have a different analysis description than the side-effects of the function [HN97]. If a function has a static return value but contains dynamic side-effects, the return value can still be considered static.

Functions communicate and provide feedback by reporting error status values. During specialization, some of these status values become static. Since these values are often returned by a function that contains side-effecting operations, a return-insensitive analysis also considers the return value dynamic. A return-sensitive analysis allows the return values to be considered static.

Use sensitivity

In order to see why use sensitivity is needed, it is first necessary to understand how the context of a variable use effects its binding time. A static variable in a dynamic context is evaluated during specialization and the resulting value is converted, or *lifted*, into its textual representation. Values for which there exists a corresponding textual representation, such as integers, can be lifted, but values which do not have a textual representation, such as pointers, structures, and arrays, cannot be lifted. Therefore, the use of a non-liftable static variable in a dynamic context must be dynamic.

A use insensitive analysis forces all uses of a variable to have the same binding time. For example, assume a variable is assigned a static value and is then used multiple times. If *any* of the contexts are dynamic, then *all* of the uses become dynamic. This unnecessarily forces the uses that appear in static contexts to be considered dynamic.

Use sensitivity keeps a variable that *must be* residualized from interfering with other uses of the variable which *could be* evaluated [HNC96]. Therefore, even if one variable use becomes dynamic due to its dynamic context, the other variable uses in static contexts remain static.

Use sensitivity is more precise than flow sensitivity. Flow sensitivity associates a different binding-time value to a variable each time it is assigned. For each assignment, however, a variable may be used multiple times. Use sensitivity associates a different binding time to each of these uses.

As we pointed out earlier, communication within and between modules also provides opportunities for specialization. The system state is usually implemented by complex data structures that combine such values as pointers, structures, and arrays. Since a system state is usually partially static, it is used in static computations as well as dynamic computations. In a use-insensitive analyses, if a non-liftable value is used in a dynamic context, all of its uses become dynamic, independent of their contexts. Therefore, the whole state is considered dynamic and no static values are exploited. A use-sensitive analysis successfully allows the static parts of the partially static system state to be accessed.

1.4 Summary

This work demonstrates that partial evaluation is a useful technique for optimizing real-world programs. We choose an offline approach, since it is better suited to exploit these opportunities. The static analyses use global program information to pre-compute the transformations applied during specialization, which is necessary to treat the complexities of existing programs written in a realistic language. Additionally, the offline approach allows fast run-time specialization, since binding-time information allows a target-generating extension to be generated automatically.

We have discussed the specialization opportunities that exist in systems programs, and have shown why existing analyses are not able to fully exploit these opportunities. In Chapter 2 we explain in detail the notions of flow, context, return, and use sensitivity. We also introduce our two-phase approach to binding-time analysis. The first phase of our approach includes the binding-time phase, which determines whether a construct is static or dynamic, and is presented in Chapter 3. The second phase, the transformation phase, determines if a construct will be evaluated or residualized, and is given in Chapter 4. These analyses have been implemented and integrated into Tempo, our partial evaluator for C, as shown in Chapter 5. Chapter 6 shows the results of applying Tempo to a variety of applications, including a commercial operating system component. We give final remarks concerning this work in Chapter 7.3.2.

Chapter 2

Analysis Features

In Chapter 1 we have explored existing specialization techniques and shown the respects in which they were inadequate to effectively treat existing, realistic programs. Our empirical study of systems programs demonstrated the need to incorporate new features such as flow, context, return, and use sensitivity into specialization analyses. A brief description of each of these features has already been given.

In this chapter, we further clarify each feature by giving specific examples in which the behavior of the systems programs is represented. For each example, we first present the initial source program, the results of each phase of our analysis, and the resulting specialized program. In some cases, we also show how specializing with existing, less precise analyses, produces a poorly specialized program. As use sensitivity is a novel feature, in addition to giving examples which show why it is needed, we also provide an explanation as to how use-sensitive annotations are computed.

2.1 Two-Phase Binding-Time Analysis

Let us first give a brief overview of our analyses in order to understand how they treat the specific examples. This is important since our approach splits traditional binding-time analysis into two phases.

Binding-time analysis determines a binding time for each construct. The binding time directly determines the transformation used for the construct during specialization. Static constructs are evaluated while dynamic constructs are residualized. Computing this binding time requires the binding-time analysis to perform two different tasks. First of all, constructs are annotated as static if they only depend on static values. This task can be thought of as determining the *availability* of the values on which a construct depends. In some cases, such as treating a simple subset of a functional language, performing this first task is sufficient. The specializer can evaluate static constructs and residualize dynamic

constructs.

When dealing with more complicated programs or programming languages, performing this first task is not sufficient. This is because simply evaluating static constructs and residualizing dynamic constructs during specialization produces a residual program that is either suboptimal or even incorrect. It turns out that it is desirable, and even required in certain cases, that some constructs which only depend on static information and therefore *could* be computed at specialization time be residualized. This is the second task of a binding-time analysis: to identify static constructs which should be residualized and force them to be dynamic. This means certain constructs will be annotated dynamic even though they only depend on static values.

Our approach to binding-time analysis explicitly separates these two tasks into two different phases. The first phase, which we call the *binding-time phase*, only determines whether or not a construct depends on static values. At this point, all constructs which only depend on static values are annotated as static. The second phase then decides how each construct should be transformed, and annotates the program accordingly. To distinguish this aspect of the binding-time analysis, we call this second step the *transformation phase*; it describes its functionality. Additionally, to distinguish the meaning of these annotations from the binding-time annotations, the transformation phase annotates the program with different annotations. Constructs which are evaluated are annotated *evaluate* and those residualized are annotated *residualize*.

Let us now look at examples which exhibit flow, context, return, and use sensitivity. For each example, we first present the initial source program. Then we give the program annotated by the binding-time phase, where static constructs are in *italics* and dynamic constructs are in **boldface**. Remember, this information only reflects the availability of information at specialization time. Next, we present the transformation-annotated program, where constructs in *italics* are to be evaluated and those in **boldface** are to be residualized. These annotation directly indicate how each construct is transformed at specialization time. Finally, we show the specialized program resulting from performing these transformations.

2.2 Flow Sensitivity

For imperative languages, assignment statements update the store. We call the abstract store manipulated by a static analysis a *state*. The evolution of this state is determined by the update operations performed at each assignment. A flow-sensitive analysis associates a different analysis state with each of these assignments. This allows variables at different program points to have different analysis descriptions. Both the binding-time phase and the transformation phase are flow-sensitive.

In the example in Fig. 2.1, the function `f()` contains a sequence of assign-

Source code

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;
  x = x + d;
  x = x + y;
  ⋮                /* p = &x or &y */
  *p = d;
  x = x + y;
}
```

Binding-time annotated code

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;
  x = x + d;
  x = x + y;
  ⋮
  *p = d;          /* alias:  p → { x, y } */
  x = x + y;
}
```

Transformation annotated code

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;
  x = x + d;
  x = x + y;
  ⋮
  *p = d;          /* alias:  p → {x, y} */
  x = x + y;
}
```

Specialized code (w.r.t. x = 2, y = 3)

```
int d;
void f_1(int x, int y, int *p)
{
  x = 5 + d;
  x = x + 3;
  ⋮
  *p = d;
  x = x + y;
}
```

Figure 2.1: Flow sensitivity

ments, in which variable `x` is read and written multiple times. This function is analyzed with an initial binding-time state specifying that parameters `x`, `y`, and `p` are all static, and that the global variable `d` is dynamic.

The binding-time annotations show how assignments update the binding-time state and how variables access it. The first assignment is static, which causes `x` to remain static in the state. This can be seen by the fact that the next use of `x` is considered static. The second assignment is dynamic, since the right-hand side contains a use of variable `d` which is dynamic. This renders `x` dynamic in the state, which in turn causes the next use of `x` to be dynamic. By associating a state with each program point and updating it according to the binding time of the assignments in this fashion, a flow-sensitive binding-time analysis is capable of considering `x` static at some program points but dynamic at others.

Occurrences of `x` on the left-hand side of an assignment are all considered static, even when `x` is dynamic. This is because the binding-time state contains a binding time for the *contents* of variable `x`, not its *address*. The address of a variable is always known at specialization time and therefore is always considered static, regardless of its binding-time description in the state.

The transformation phase then uses these binding-time annotations to determine the transformation for each construct. Most static constructs become evaluate constructs and dynamic constructs become residualize constructs—but there are a few exceptions. First of all, the static left-hand sides of dynamic assignments are annotated residualize. Again, this is because computing the variable's address at specialization time and lifting the resulting value creates an invalid residual program. Therefore, the transformation phase annotates these variables as residualize, instructing the specializer to residualize the variable identifiers. Secondly, even though all of the parameters have a static binding time, they are all annotated with a residualize transformation. This is due to the fact that these parameters must appear in the residual program since all three variables `x`, `y`, and `p` have uses which are residualized.

This example also illustrates some other aspects of the analyses. In general, pointers and aliasing may create *ambiguous* definitions: assignments for which the analysis cannot statically determine which location will be modified at run time. In the example, we assume that the omitted lines (represented with ⋮) contain code after which pointer `p` may point to either variable `x` or variable `y`. This causes the subsequent definition of `*p` to be an ambiguous assignment (binding-time and transformation annotated aliases appear in comments next to the dereferenced pointer statement). Since the assignment is dynamic, both locations become dynamic.

The subsequent specialization phase is guided by the transformation annotations. Evaluate constructs are evaluated and residualize constructs are residualized. Evaluate statements disappear completely. Evaluate expressions are evaluated, and the resulting value is lifted into the residual code. Residualize

expressions and statements are residualized.

2.3 Context Sensitivity

Context sensitivity enables a function to be analyzed with respect to different analysis descriptions, or *contexts*, producing an annotated instance of the function for each context. A context contains the information relevant to analyze the function, consisting of the binding times of formal parameters as well as the binding times of the non-local variables (*e.g.*, a global variable) used by the function. Since annotated instances are treated separately, the analysis is able to exploit the static values of each specific context.

The second example shows a function `f()` which contains a sequence of calls to function `g()`, as given in Fig. 2.2. Function `f()` is analyzed with an initial binding-time description specifying that global variable `x` is static, while global variables `y` and `d` are dynamic. The binding-time context of the first call consists of a static actual parameter, a static non-local variable `x`, and a dynamic non-local variable `y` (the binding-time and transformation contexts appear in comments next to function calls). The first instance of the function is then analyzed with respect to this context. Notice that `x` is dynamic after the first instance of `g()` is analyzed, since `x` is assigned to a dynamic value in the body of `g()`. This creates a different binding-time context for the second call to `g()`, in which `x` is dynamic. Therefore, a second instance of the function is created and annotated with respect to this new context. The third call to `g()` has the same binding-time context as the second call, so a new instance is not created.

The program, including both binding-time annotated instances of function `g()`, is then annotated with transformations. The transformation phase is also context sensitive, which means that an additional instance of each function is created for each additional transformation context generated during the analysis. We do not see this situation in this example, since all transformation contexts are the same.

Finally, the program is specialized. In the residual program, each instance of function `g()` produces a different residual function definition. Since the specializer is also context sensitive (also known as a *polyvariant specializer*) it is possible that different specialization contexts produce different residual functions. This does not happen in this example, since the specialization contexts of the second call and third call to `g()` are the same (z = 5). Therefore, both calls share the same residual function definition.

2.4 Return Sensitivity

Return sensitivity allows a function to return a static value even though the

Source code

```
int x, y, d;
void f()
{
  x = 1;
  y = d;
  g(5);
  g(5);
  g(5);
}
```

```
g(int z)
{
  x = (x + z) + y;
}
```

Binding-time annotated code

```
int x, y, d;
void f()
{
  x = 1;
  y = d;
  g(5); /* ctx: z, x, y */
  g(5); /* ctx: z, x, y */
  g(5); /* ctx: z, x, y */
}
```

```
void g(int z) /* ctx: z, x, y */
{
    x = (x + z) + y;
}
void g(int z) /* ctx: z, x, y */
{
  x = (x + z) + y;
}
```

Transformation annotated code

```
int x, y, d;
void f()
{
  x = 1;
  y = d;
  g(5); /* ctx: z, x, y */
  g(5); /* ctx: z, x, y */
  g(5); /* ctx: z, x, y */
}
```

```
void g(int z) /* ctx: z, x, y */
{
    x = (x + z) + y;
}
void g(int z) /* ctx: z, x, y */
{
    x = (x + z) + y;
}
```

Specialized code

```
int y, d;
void f_1()
{
  y = d;
  g_1();
  g_2();
  g_2();
}
```

```
void g_1()
{
  x = 6 + y;
}
void g_2()
{
x = (x + 5) + y;
}
```

Figure 2.2: Context sensitivity

Source code

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Binding-time annotated code

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d ;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Transformation annotated code

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Specialized code

```
int x, y, d;
void f_1()
{
  x = (g_1() * 2) + d;
}
```

```
int g_1()
{
  y = 1 + d;
  return 4;
}
```

Figure 2.3: Return insensitivity

Source code

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Binding-time annotated code

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d ;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Transformation annotated code

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Specialized code

```
int x, y, d;
void f_1()
{
  g_1();
  x = 8 + d;
}
```

```
void g_1()
{
  y = 1 + d;
}
```

Figure 2.4: Return sensitivity

function contains dynamic side-effects and is therefore residualized. In the third example, shown in Fig. 2.3, function `f()` contains a call to function `g()`, the return value of which is used in further computations. Since return sensitivity is a new concept, we first show how a return-insensitive binding-time analysis would annotate this same example, as shown in Fig. 2.3. Function `f()` is analyzed with an initial binding-time state specifying that global variables `x`, `y`, and `d` are all dynamic. Recall that a return-insensitive analysis only calculates one binding time and one transformation for the entire function. Therefore, the dynamic parts of `g()` which are residualized prohibit the static return value from being evaluated. The return statement is residualized and the multiplication operation is not evaluated.

A return sensitive analysis computes two different binding times for each function, one for the side-effects and the other for the return value, as shown in Fig. 2.4. To visualize these two binding times at the function definition, we indicate that the function contains dynamic side-effects by annotating the identifier **g** as dynamic and a static return value by annotating its return type *int* as static. To visualize the two binding times at the function call site, the identifier ***g*** is annotated as both static and dynamic (indicated by ***italics_and_boldface***). Return sensitivity allows the static value returned by `g()` to be used at its call site, which in turn enables the multiplication operation which depend on this static value to be considered static as well.

The transformation annotations are similar. At the function's definition, the identifier **g** is annotated residualize and its return type *int* as evaluate. At the call site, the identifier ***g*** is annotated as both evaluate and residualize.

The specializer exploits the return value returned by `g()` to evaluate the multiplication operation, and residualizes the call in order to residualize its side-effects. The specialized definition of `g()` is a void function—no longer returning a value.

2.5 Use-Sensitivity

Use sensitivity addresses the complex manner in which systems programs use pointers and data structures. This situation occurs in many other realistic programs. The goal is to prevent uses of a variable which *must be* residualized from interfering with other uses of the variable which *could be* evaluated.

This is not a problem in the binding-time phase, since all uses of a variable have the same binding time at the same program point. This does pose a problem, however, for the transformation phase, since a variable may be annotated with different transformations even at the same program point. This is because a variable transformation depends its context.

This is a new concept not yet explored: it is a subtle yet important point. Therefore, in addition to giving examples which demonstrate the need for use

sensitivity, we also show how use-sensitive annotations are computed. This should also clarify the differences between the binding-time phase and transformation phase of our analysis, as well as motivate the new transformations we introduce.

2.5.1 Examples

Let us consider two different examples that illustrate the need for use sensitivity, one that deals with pointers and one with structures. These examples are based on existing systems programs we have studied. For each example, we will show the annotations produced by a use-insensitive analysis followed by the same example annotated by a use-sensitive analysis.

In Fig. 2.5, a pointer variable `p` is assigned the address of an array, and is subsequently used twice at the same statement. If function `f()` is analyzed with a binding-time context that specifies all global variables are static except variable `d`, the binding-time phase annotates all constructs static, except those that depend on `d`. Specifically, the assignment to pointer `p` as well as both of its uses are annotated as static.

However, a use-insensitive transformation phase annotates the assignment as well as both uses to be residualized. During specialization, a static construct that occurs in a dynamic context is evaluated and the resulting value is lifted into its corresponding textual representation which is then residualized. Some values cannot be lifted, however. For example, lifting a pointer value would residualize an address into the residual program, which is an illegal transformation. Pointer, structure, and arrays values are all *non-liftable*. Therefore, the transformation phase forces static non-liftable constructs in dynamic contexts to be residualized, instead of evaluating the construct and lifting the resulting value.

In the example in Fig. 2.5, we see that the second use of pointer `p` is static and occurs in a dynamic context. Therefore, it is annotated with a residualize transformation. Since a use-insensitive analysis requires that all uses of a variable have a unique binding time, all of the uses of pointer `p` are forced to become dynamic. Residualizing the first use of pointer `p` causes the entire expression in which it appears to be residualized. In addition, the definition of pointer `p` must be residualized.

The program is subsequently specialized with respect to these annotations. Due to the use-insensitive transformation phase, residualizing the use of `p` in the dynamic context prevents the use of `p` in the static context from being evaluated. This situation also keeps the static addition expressions from being evaluated.

Use sensitivity avoids this problem. All uses of a variable are no longer required to have the same transformation at the same program point, and therefore residualized uses do not interfere with the evaluated uses. A use-sensitive transformation phase therefore annotates the program fragment as shown in Fig. 2.6. The use of `p` in the static context is evaluated during specialization, which in turn allows the subsequent additions to be evaluated.

Source code

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Binding-time annotated code

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Transformation annotated code

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234 ;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Specialized code (w.r.t. s = 3)

```
int a[10], *p, d, x;
void f_1()
{
  a[4] = 1234;
  p = a;

  x = *((p + 3) + 1) + *((p + d) + 2);
}
```

Figure 2.5: Use insensitivity (pointer program)

Source code

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Binding-time annotated code

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Transformation annotated code

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234 ;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Specialized code (w.r.t. s = 3)

```
int a[10], *p, d, x;
void f_1()
{
  a[4] = 1234;
  p = a;

  x = 1234 + *((p + d) + 2);
}
```

Figure 2.6: Use sensitivity (pointer program)

Source code

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Binding-time annotated code

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Transformation annotated code

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Specialized code (w.r.t. `s1.s` = 10)

```
struct {int s; int d;} s1, s2;
int x;
void f_1()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Figure 2.7: Use insensitivity (structure program)

Source code

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Binding-time annotated code

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Transformation annotated code

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Specialized code (w.r.t. `s1.s` = 10)

```
struct {int s; int d;} s1, s2;
int x;
void f_1()
{
  s1 = s2;

  x = 13 + (s1.d + 4);
}
```

Figure 2.8: Use sensitivity (structure program)

Another example which motivates the need for use sensitivity is shown in Fig. 2.7, containing a typical example of how data structures are used in systems programs. The data structure `s1` is assigned a value, and each of its fields are used in some computations.

This example is binding-time analyzed with respect to a binding-time state which specifies that variable `x` is dynamic and that data structures `s1` and `s2` are partially static. Specifically, field `s` of the data structures is static while field `d` is dynamic. The binding-time phase annotates the definition of `s1` as dynamic, since one of the fields contains dynamic data. There are two subsequent uses of `s1`. The access to the static field `s` is considered static while the access to field `d` is dynamic. Again, we find that the data structure `s1`, which is considered static since it occurs as a left hand side expression, is used in both a static and a dynamic context.

The transformation phase must force the static use of `s1` in the dynamic context to be residualized, since structure values are non-liftable. A use-insensitive analysis would therefore force all uses of `s1` to be residualized. The resulting transformation annotations and the resulting specialized program are given in Fig. 2.7.

A use-sensitive analysis avoids losing this information. The static use of `s1` in the dynamic context is still residualized (since it is non-liftable), but the static use of `s1` in the static context is evaluated. The transformation annotations and corresponding specialization are given in Fig. 2.8.

2.5.2 Computing Annotations

In order to clarify the idea of use sensitivity, it is important to understand how binding-times and transformations of variables and their definitions are computed.

First of all, the binding time of a variable is determined by the assignment which defines the variable, since the assignment determines the values on which the variable depends. The variable is static if all of the values on which its depends are static. In this case, all of the variable's uses are considered static, since they can be computed at specialization time. If the variable is assigned a dynamic value, all of the uses defined by that assignment are considered dynamic.

To determine whether or not the values on which a variable depends are static, we use the function *values*(), shown in Fig. 2.9, to express the values on which a variable depends and the function *values-bt*() determines if all of these values are static. Fig. 2.10 contains the binding-time domain $\{S, D\}$, representing static and dynamic respectively, the lattice which orders these elements, and the least upper bound operator $\sqcup$ used to to compute *values-bt*(). If a variable does not depend on any value (*e.g.,* a variable which occurs as a left-hand side expression), *values-bt*() returns static.

Secondly, the binding-times of the contexts in which a variable occurs are

$$
\begin{aligned}
\mathit{values}(id) &= \textit{values on which id depends} \\
\mathit{context}^e(id) &= \textit{values on which the context of id depends (at program point e)}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{values\text{-}bt}(id) &= \bigsqcup_{bt \,\in\, (\{bt(c) \;\mid\; c \in \mathit{values}(id)\} \,\cup\, \{S\})} bt \\
\mathit{context\text{-}bt}^e(id) &= \bigsqcup_{bt \,\in\, \{bt(c) \;\mid\; c \in \mathit{context}^e(id)\}} bt
\end{aligned}
$$

Figure 2.9: Value and context binding times.

computed using the function $\mathit{context}^e()$ to represent the values in a context, where the superscript e indicates the program point of the specific variable use, and the function $\mathit{context\text{-}bt}^e()$ to express the binding time of the context. The context is considered static if all the values in the context are static.

Recall that the binding-time annotations are the same for the pointer program examples (Fig. 2.5 and 2.6). Pointer `p` is defined by a static assignment and used in a static context and a dynamic context. This information is computed as follows:

$$
\begin{aligned}
\mathit{values}(p) &= \{a\} \\
\mathit{context}^1(p) &= \{s, 1\} \\
\mathit{context}^2(p) &= \{d, 2\}
\end{aligned}
$$

$$
\begin{array}{rlll}
\mathit{values\text{-}bt}(p) & = bt(a) & = S & = S \\
\mathit{context\text{-}bt}^1(p) & = bt(s) \sqcup bt(1) & = S \sqcup S & = S \\
\mathit{context\text{-}bt}^2(p) & = bt(d) \sqcup bt(2) & = D \sqcup S & = D
\end{array}
$$

Given this binding time information, the transformation phase determines how each construct will be transformed. The four possible transformations are shown in Fig. 2.10. This new domain expresses all of the possible transformations which may be needed. The transformations $\{E\}$ and $\{R\}$ are used to indicate that the construct will be evaluated or residualized, respectively. In certain cases, a construct will be both evaluated *and* residualized, which is represented by the transformation $\{E, R\}$. The transformation $\{\}$ corresponds to a construct which is not used, such as dead code, and therefore does not need to be evaluated nor residualized.

	binding times	transformations
domain	$Bt = \{S, D\}$	$Tr = \{\{\}, \{E\}, \{R\}, \{E, R\}\}$
lattice	D | S	$\{E, R\}$ / \ $\{E\}$ $\{R\}$ \ / $\{\}$
least upper bound	$\sqcup$	$\cup$
$\overline{\ }: Bt \to Tr$	$\overline{D} = \{R\}$ $\overline{S} = \{E\}$	

Figure 2.10: Binding times and transformations.

In many cases, as seen in the examples presented earlier, all dynamic constructs are residualized and most static constructs are evaluated. In these cases, the function $\overline{\ }$ (defined in Fig. 2.10) is simply applied to the binding time of a construct to determine its corresponding transformation.

However, calculating the transformation of a construct is not always this easy. As previously mentioned, complications arise when a static value occurs in a dynamic context. If the value is liftable, then it can be evaluated and the result lifted. If, however, the value is non-liftable, it must not be evaluated since the result cannot be lifted into the residual program. Therefore, even though the construct is static, it must be residualized. Determining this transformation takes into account not only the binding time of the variable but also the binding time of its context.

Fig. 2.11 defines function *use-tr*$^e(id)$ which calculates the transformation for each use of a variable and function *def-tr*(id) for its definition. For a variable whose value can be lifted, these transformations only depend on the binding time given by *values-bt()*. A static variable will be evaluated while a dynamic variable will be residualized. A static variable in a dynamic context will be evaluated and the resulting value will be lifted. For non-liftable values, however, the transformation is computed by taking the least upper bound of the binding times computed by *values-bt()* and the least upper bound of all binding times computed by *context-bt()*. This calculation is done in order to residualize the

		liftable value	non-liftable value
use-insensitive	$\textit{use-tr}^{e}(id)$	$\overline{\textit{values-bt}(id)}$	$\overline{\textit{values-bt}(id) \sqcup \bigsqcup_{e1 \in \textit{uses}(id)} \textit{context-bt}^{e1}(id)}$
	$\textit{def-tr}(id)$	$\textit{use-tr}^{e \in \textit{uses}(id)}(id)$	$\textit{use-tr}^{e \in \textit{uses}(id)}(id)$
use-sensitive	$\textit{use-tr}^{e}(id)$	$\overline{\textit{values-bt}(id)}$	$\overline{\textit{values-bt}(id) \sqcup \textit{context-bt}^{e}(id)}$
	$\textit{def-tr}(id)$	$\textit{use-tr}^{e \in \textit{uses}(id)}(id)$	$\bigcup_{e1 \in \textit{uses}(id)} \textit{use-tr}^{e1}(id)$

Figure 2.11: Computing use and definition transformations

static variables in dynamic contexts. A static variable in a dynamic context will be residualized.

However, the use-insensitive analysis computes one transformation for all uses. All of the context binding-times are merged together, so one dynamic context forces all uses to be considered dynamic. As well, this same transformation is then used for the variable's definition.

Looking at the transformation annotations of the first example (Fig. 2.5), we see that the definition of pointer `p` and all of its uses are annotated residualize. These annotations are computed as follows:

$$\begin{aligned}
\textit{use-tr}^{1}(p) &= \overline{\textit{values-bt}(p) \sqcup \bigsqcup_{e1 \in \{1,2\}} \textit{context-bt}^{1}(p)} = \overline{S \sqcup (D \sqcup S)} = \overline{D} = \{R\} \\
\textit{use-tr}^{2}(p) &= \overline{\textit{values-bt}(p) \sqcup \bigsqcup_{e1 \in \{1,2\}} \textit{context-bt}^{1}(p)} = \overline{S \sqcup (D \sqcup S)} = \overline{D} = \{R\} \\
\textit{def-tr}(p) &= \textit{use-tr}^{e \in \{1,\,2\}}(p) = \{R\}
\end{aligned}$$

Pointer `p` depends on a single static value, and occurs in a static context and a dynamic context. The transformation for its first use, computed by $\textit{use-tr}^{1}(p)$, is the least upper bound of these binding times. Since this least upper bound is

dynamic, the corresponding transformation is $\{R\}$. The transformation for its second use is computed similarly by *use-tr*$^2(p)$. The definition of `p` is annotated with the transformation computed by *def-tr*(p). Since both uses necessarily have the same transformation, $\{R\}$, either use transformation can be used.

The goal of a use-sensitive analysis is to allow different uses to have different transformations, which is achieved by the use-sensitive *use-tr*(id) and *def-tr*(id) defined in Fig. 2.11. The use transformations are combined to compute the definition transformation, using the new transformation lattice and least upper bound operation defined in Fig. 2.10. Since transformations are computed separately for each use, we distinguish between different uses by superscripting *use-tr()* with the program point (e) corresponding to the use. For a variable whose value can be lifted, the use and the definition transformations are computed as in a use-insensitive analysis. For non-liftable values, however, the transformation of each use only takes into account the specific context in which the variable is used. This is where use sensitivity is obtained. Since different uses of the same variable may now have different transformations, the transformation for the corresponding definition must take this into account. This is the purpose of the $\{E, R\}$ transformation.

Looking back at the first example (Fig. 2.6), we see the use-sensitive transformation phase indeed annotates the static use of pointer `p` as evaluate, and the static use of `p` in the dynamic context as residualize. Additionally, its definition is annotated as evaluate and residualize, (indicated by ***italics_and_boldface***). This is calculated as follows:

$$
\begin{array}{rlllll}
\textit{use-tr}^1(p) & = \overline{\textit{values-bt}(p) \sqcup \textit{context-bt}^1(p)} & = \overline{S \sqcup D} & = \overline{D} & = \{R\} \\
\textit{use-tr}^2(p) & = \overline{\textit{values-bt}(p) \sqcup \textit{context-bt}^2(p)} & = \overline{S \sqcup S} & = \overline{S} & = \{E\} \\
\textit{def-tr}(p) & = \bigcup_{e1 \in \{1,2\}} \textit{use-tr}^1(p) & = \overline{D} \cup \overline{S} & = \{R\} \cup \{E\} & = \{E, R\}
\end{array}
$$

As before, pointer `p` depends on a single static value, and occurs in a static context and a dynamic context. However, only the binding time of its first context is used to compute the transformation for its first use, *use-tr*$^1(p)$. The first context is dynamic, which leads to the transformation $\{R\}$. Likewise, only the binding time of its second context is used to compute the transformation for its second use, *use-tr*$^2(p)$. Since this context is static, the transformation is $\{E\}$. The least upper bound of these two transformations is then computed by *def-tr*(p), yielding the transformation $\{E, R\}$ for its definition.

Specializing the program with respect to these use-sensitive transformations produces better results. At specialization time, the definition of `p` is both evaluated and residualized. Evaluating this definition allows the use of `p` in the static context to be evaluated. The fact that the residual use of `p` occurs in the residual program means that the definition of `p` must also be residualized.

In the second example (Fig. 2.7), we compute the following transformations for structure `s1`:

$$\begin{aligned} \mathit{values}(s1) &= \{\} \\ \mathit{context}^1(s1) &= \{s, 3\} \\ \mathit{context}^2(s1) &= \{d, 4\} \end{aligned}$$

$$\begin{aligned} \mathit{values\text{-}bt}(s1) &= S \\ \mathit{context\text{-}bt}^1(s1) &= bt(s) \sqcup bt(3) = S \sqcup S = S \\ \mathit{context\text{-}bt}^2(s1) &= bt(d) \sqcup bt(4) = D \sqcup S = D \end{aligned}$$

The use-insensitive transformation phase produces the annotations seen in Fig. 2.7 as follows:

$$\begin{aligned} \mathit{use\text{-}tr}^1(s1) &= \overline{\mathit{values\text{-}bt}(s1) \sqcup \bigsqcup_{e1 \in \{1,2\}} \mathit{context\text{-}bt}^1(s1)} = \overline{S \sqcup (S \sqcup D)} = \overline{D} = \{R\} \\ \mathit{use\text{-}tr}^2(s1) &= \overline{\mathit{values\text{-}bt}(s1) \sqcup \bigsqcup_{e1 \in \{1,2\}} \mathit{context\text{-}bt}^1(s1)} = \overline{S \sqcup (S \sqcup D)} = \overline{D} = \{R\} \\ \mathit{def\text{-}tr}(s1) &= \mathit{use\text{-}tr}^{e \in \{1,\,2\}}(s1) = \mathit{use\text{-}tr}^1(s1) = \{R\} \end{aligned}$$

The use-sensitive transformation phase, on the other hand, computes the following information and produces the annotations seen in Fig. 2.8.

$$\begin{aligned} \mathit{use\text{-}tr}^1(s1) &= \overline{\mathit{values\text{-}bt}(s1) \sqcup \mathit{context\text{-}bt}^1(s1)} = \overline{S \sqcup S} = \overline{S} = \{E\} \\ \mathit{use\text{-}tr}^2(s1) &= \overline{\mathit{values\text{-}bt}(s1) \sqcup \mathit{context\text{-}bt}^2(s1)} = \overline{S \sqcup D} = \overline{D} = \{R\} \\ \mathit{def\text{-}tr}(s1) &= \bigcup_{e1 \in \{1,2\}} \mathit{use\text{-}tr}^1(s1) = \overline{S} \cup \overline{D} = \{E\} \cup \{R\} = \{E, R\} \end{aligned}$$

As with the pointer example, we see here that the use-insensitive analysis forces all of the uses of `s1` to have the same transformation, namely $\{R\}$, since one of the contexts is dynamic. The use-sensitive analysis allows the different uses to have different transformations. In this example, one use is evaluated while the other is residualized. Supporting these different use transformations, likewise, requires that the definition of `s1` to be evaluated and residualized at specialization time.

2.6 Summary

We have seen a number of examples which demonstrate how analyses which are flow, context, return, and use sensitive achieve more effective specialization. Our

approach involves two separate analyses. The binding-time phase determines whether a construct only depends on static information during specialization. This is followed by the transformation phase, which determines the transformation for each construct. We have shown how to compute binding times and transformations. Computing binding times is similar to previous analyses, since it uses the same abstract domain and lattice, although our binding time analysis only computes availability information. Computing transformations is more complicated, involving the introduction of a new abstract domain and lattice. This leads to a new transformation which both evaluates and residualizes a construct.

Chapter 3

Binding-Time Phase

In this chapter we present the binding-time phase using a data-flow analysis framework (see, for instance, [ASU86, MR90]). First, we present preliminary information needed to understand the analysis, such as the language treated and how memory locations are modeled. We assume that, prior to the binding-time phase, a number of preprocessing analyses have been executed. These analyses include an alias analysis, a definition analysis, and an inter-procedural use-def analysis. We describe these analyses, and then present the binding-time phase of our binding-time analysis.

Language We consider the subset of C described in Fig. 3.1. Constructs such as function calls, multiple returns, structures, and pointers are included in this subset in order to show the important aspects of our analyses. Many other constructs, such as conditional expressions, the comma operator, and gotos, are translated into this subset. For example, `for` and `while` loops are converted into `do-while` loops. Other simplifications are made to clarify and simplify the presentation of the analysis. A function call, for instance, is assumed to return a value, which is always directly assigned to an identifier. This strategy simplifies the analysis without restricting its applicability. We assume that all programs are transformed prior to the analysis, if needed, so that they conform to this constraint. Constructs which are not rewritten, such as break/continue and dynamic memory allocation, are treated directly by the analysis although they are not presented here.

Locations There are three types of locations: variable locations, return locations, and structure component locations. A location for a variable, if this variable is not a structure, is represented by a plain identifier. The return value of a function is denoted $RETURN(f)$, where f is the program point of the function in which the return statement occurs. A location for a structure type *type* and component *field* is denoted *type.field*. We shall assume that, given a structure type, the function *location()* returns the structure component locations associated

Domains:

$$\begin{array}{rcl} const & \in & \mathcal{C}onstant \\ id & \in & \mathcal{I}dentifier \\ bop & \in & \mathcal{B}inary\mathcal{O}perator \end{array}$$

Abstract syntax:

program	::=	type-def* decl* fun-def+	program
type-def	::=	`struct` *id* { decl+ }	type definition
decl	::=	type-spec *id*	declaration
type-spec	::=	`void` | `char` | `int` | ...	base types
	|	`*` type-spec	pointer type
	|	`struct` *id*	structure type
fun-def	::=	type-spec *id* `(` decl* `)` stmt	function definition
stmt	::=	lexp `=` exp	assignment
	|	`if (` exp `)` stmt `else` stmt	conditional
	|	`do` stmt `while (` exp `)`	loop
	|	{ stmt* }	block
	|	*id* `=` *id* `(` exp* `)`	function call
	|	`return` exp	return
lexp	::=	*id*	variable
	|	lexp`.` *id*	structure select
	|	`*` exp	dereference
exp	::=	*const*	constant
	|	*id*	variable
	|	lexp`.` *id*	structure select
	|	`&` lexp	address
	|	`*` exp	dereference
	|	exp *bop* exp	binary expression

Figure 3.1: Syntax of C subset

to that type. Notice that associating locations with structure components on a structure type and field basis allows different components of a given structure to have different binding times but forces the same components of different structures of a given type to have the same binding time. This conservative handling has so far been sufficient to treat the existing applications we have encountered, and can be extended to achieve more precision if needed.

Alias Analysis Our alias analysis is very similar to existing ones [EGH94, Ruf95]. It is based on the *points-to* model of aliasing. The alias analysis gives, for each dereference expression $*^e exp$ at program point e (superscripting is used to denote program points in the syntax of a program), the set $aliases(e)$ of corresponding aliases, *i.e.*, the set of locations the expression may represent.

Definition Analysis The definition analysis computes, for each statement at program point s, the set of locations *defs-stmt*(s), which *may* be defined by the statement. A location is defined if it occurs on the left hand side of an assignment, given by *defs-lexp*(s). If the left hand side expression is an identifier, then the identifier location is defined. A dereferenced pointer expression defines all the possible locations the pointer may point to. Defining a structure component only defines the single structure component specified, while specifying an entire structure defines all components of the structure.

For a given assignment, a set containing multiple locations is computed when the location defined at run-time cannot be determined statically. These *ambiguous* definitions are due either to aliasing or, with our representation of structures, structure component assignment. If, on the other hand, a static analysis can deduce the location which will be defined at run-time by a statement, then the definition is considered *unambiguous*. In this case, the function *unambiguous-defs*() returns the unambiguously defined location, which can be used by the subsequent analyses to produce more accurate results. For example, in the binding-time phase, an unambiguous definition permits a dynamic variable to become static. In the transformation phase, an unambiguous definition is treated as a killing definition, indicating that the location is definitely defined.

Since understanding functions *defs-stmt*(s) and *unambiguous-defs*() is important to understand how binding times are computed, we present both in Fig. 3.2.

Inter-procedural Use-Def Analysis A summary of the non-local locations used and defined is computed for each function, similar to interprocedural summary information [Bar78]. We consider that *used-non-locals*() and *def-non-locals*() represent the used non-local and defined non-local locations of a function, respectively. Note that this phase must follow (or be combined with) alias analysis. When, as a right-hand side (left-hand side) expression, a

function definitions:

id^f `(` decls `)` stmt^s:

$\quad$ *defs-fun*(f) = *defs-stmt*(s)

statements:

lexp^{e_1} `=`^{s} exp^{e_2}:

$\quad$ *defs-stmt*(s) = *defs-lexp*(e_1)

$\quad$ *unambiguous-defs*$(s) = (|\ \textit{defs}(s)\ |= 1) \rightarrow \textit{defs-stmt}(s); \{\}$

`if`^{s} `(` exp^e `)` $\text{stmt}_1^{s_1}$ `else` $\text{stmt}_2^{s_2}$:

$\quad$ *defs-stmt*(s) = *defs-stmt*$(s_1) \cup$ *defs-stmt*(s_1)

`do`^{s} stmt^{s_0} `while (` exp^e `)`:

$\quad$ *defs-stmt*(s) = *defs-stmt*(s_0)

`{` $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ `}`^{s}:

$\quad$ *defs-stmt*$(s) = \cup_{1 \le i \le n}$ *defs-stmt*(s_i)

id_1 `=`^{s} id_2 `(` $\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$ `)`:

$\quad$ *defs-stmt*(s) = *defs-fun*$(f_{id_2}) \cup \{id_1\}$

$\quad$ *unambiguous-defs*$(s) = \{id_1\}$

`return`^{s} exp^e:

$\quad$ *defs-stmt*$(s) = \{RETURN(f)\}$

left-hand side expressions:

is-a-struct(e):

$\quad$ *defs-lexp*(e) = *locations*(*type*(e))

id^e:

$\quad$ *defs-lexp*$(e) = \{id\}$

`*`^{e} exp^{e_0}:

$\quad$ *defs-lexp*$(e) = aliases(e)$

lexp^{e_0} `.`$^e id$:

$\quad$ *defs-lexp*$(e) = \{\textit{type}(e_0).id\}$

Figure 3.2: Definition analysis

pointer potentially points to several locations, all these locations must be considered used (defined). Also, the notion of use and definition actually relates to the analysis rather than to actual executions. In particular, in the case of an ambiguous definition of a pointer dereference, all the locations (potentially) pointed to by the pointer must also be considered used as their binding times are used to compute the binding times of the (potentially) defined locations.

Binding-Time Phase

Now let us consider the analysis which computes binding times. We first give a description of the abstract store model used by the analysis. Then we provide the data-flow equations which compute a binding-time description for each statement in the program. Finally, we explain how this information is used to annotate a program.

States We shall refer to the sets of values propagated by the binding-time phase as states. Binding-time states are elements of $BtState = Location \to Bt$, where Bt is the previously defined binding-time lattice domain. We shall denote $\setminus$ the binary operator from $BtState \times Locations \to BtState$ setting to bottom a set of locations.

The join operator $\underline{\sqcup}$ on binding-time states is defined as a pointwise application of the least upper bound operator $\sqcup$ on the $BtState$ function space range.

We shall use a graph representation of states. It is accessed via a lookup function which takes a graph (a set of pairs location/binding time) and a location, and returns the corresponding binding time. All the locations do not need to occur in the graph. A location which does not occur in the graph is considered to be undefined, in which case the lookup function simply returns the bottom element.

It is easy to show that $(BtState, \underline{\sqcup})$ is a join semi-lattice of finite length and that the set of transfer functions generated by closure under joins and compositions is a monotone operation space [NN91]. This means that the set of equations corresponding to any given program can be solved using one of the standard iterative algorithms [ASU86].

3.1 Data-Flow Equations

For each statement, the binding-time phase computes two different states. The binding-time state that describes the variables in the given statement is called the *statement state*.

An additional *return state* is used to collect and propagate statement states interprocedurally. The return state is initialized at the beginning of a function.

function definition:

id^f `(` decls`)` $body^s$:

$in(s) = in(f)$

ret-in(s) $= \{\}$

ret-out$(f) = \{(loc, lookup(\text{ret-out}(s), loc)) \mid loc \in \text{def-non-locals}(f)\}$

statements:

lexp^{e_1} `=`^{s} exp^{e_2}:

$out(s) = t_s(in(s))$ *ret-out*$(s) =$ *ret-in*(s)

`if`^{s} `(` exp^e `)` $\text{stmt}_1^{s_1}$ `else` $\text{stmt}_2^{s_2}$:

$in(s_1) = in(s)$ *ret-in*$(s_1) =$ *ret-in*(s)

$in(s_2) = in(s)$ *ret-in*$(s_2) =$ *ret-in*(s)

$out(s) =$ *ret-out*$(s) =$

$t_{e,s_1}(out(s_1)) \underline{\sqcup} t_{e,s_2}(out(s_2))$ $t_{e,s_1}(\text{ret-out}(s_1)) \underline{\sqcup} t_{e,s_2}(\text{ret-out}(s_2))$

`do`^{s} stmt^{s_0} `while` `(` exp^e `)`:

$in(s_0) = in(s) \underline{\sqcup} t_{e,s_0}(out(s_0))$ *ret-in*$(s_0) =$ *ret-in*$(s) \underline{\sqcup} t_{e,s_0}(\text{ret-out}(s_0))$

$out(s) = out(s_0)$ *ret-out*$(s) =$ *ret-out*(s_0)

`{` $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ `}`^{s}:

$in(s_1) = in(s)$ *ret-in*$(s_1) =$ *ret-in*(s)

$in(s_{i+1}) = out(s_i), 1 \le i < n$ *ret-in*$(s_{i+1}) =$ *ret-out*$(s_i), 1 \le i < n$

$out(s) = out(s_n)$ *ret-out*$(s) =$ *ret-out*(s_n)

id_1 `=`^{s} id_2 `(` $\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$ `)`:

$ctx = \{(\text{formal-loc}(i, id_2), \text{exp-bt}(e_i, in(s))) \mid 1 \le i \le n\} \cup$
$\{(loc, lookup(in(s), loc)) \mid loc \in \text{used-non-locals}(id_2)\}$

$in(f_{id_2,ctx}) = ctx$

$out(s) = (in(s) \setminus (\text{def-non-locals}(id_2) \cup \{id_1\}))$
$\underline{\sqcup}$ *ret-out*$(f_{id_2,ctx})$
$\underline{\sqcup} \{(id_1, lookup(\text{ret-out}(f_{id_2,ctx}), RETURN(f_{id_2,ctx})))\}$

ret-out$(s) =$ *ret-in*(s)

`return`^{s} exp^e:

$out(s) = \{\}$

ret-out$(s) =$ *ret-in*$(s) \underline{\sqcup} in(s) \underline{\sqcup} \{(RETURN(f), \text{exp-bt}(e, in(s)))\}$

Figure 3.3: Binding-time phase — data-flow equations

$$t_s(state) = \{(loc, \textit{stmt-bt}(s)) \mid loc \in \textit{defs-stmt}(s)\} \sqcup (state \setminus \textit{unambiguous-defs}(s))$$

$$t_{e,s}(state) = \{(loc, \textit{exp-bt}(e, state)) \mid loc \in \textit{defs-stmt}(s)\} \sqcup state$$

Figure 3.4: Binding-time phase — transfer functions

Each time a return statement is encountered, the statement state at the return statement is joined with the previously computed return state. At the end of a function definition, the return state returned by the function is the combination of all the statement states at each return. The binding time of the value returned by the function is propagated to the calling sites via the return location $RETURN(f)$. Each return statement sets the binding time of this location with the binding time of the expression returned.

The data-flow equations relating the statement state $in(s)$ and return state *ret-in(s)* at the entry of a statement at program point s with the statement state $out(s)$ and return state *ret-out(s)* at the output of the statement are given in Fig. 3.3.

Function definition

A function definition accepts an initial statement state $in(f)$, which is used as the input statement state $in(s)$ for the body of the function. The initial statement state for a program contains S for input parameters declared as static and D for those declared as dynamic. The return state of the body *ret-in(s)* is initialized at the beginning of every function to indicate that no return statements have been encountered. After the body is analyzed, the return state of the body, *ret-out(s)*, is used to define the return state of the function, *ret-out(f)*. Specifically, this return state contains the defined non-local variables and their corresponding binding times. Notice that, since the return state of the function is defined in terms of the return state of the body, the output statement state of the body, *out(s)*, is not used.

Let us now see how statements states are computed within the body of each function. This intra-procedural aspect of the analysis is expressed by relating the statement state $in(s)$ at the entry of a statement with the statement state $out(s)$ at the output of the statement. Then we will show how the inter-procedural aspects of the analysis are expressed by relating *ret-in(s)* with *ret-out(s)*.

Intra-procedural

An assignment at program point s modifies the statement state as described by the transfer function $t_s()$ (shown in Fig. 3.4). It updates each location in the set of possible definitions for the assignment with the assignment binding time, given

by *stmt-use-bt*(s) (see Fig. 3.5). Note that the assignment binding time depends on the input state of the assignment. If the assignment is ambiguous, a safe approximation has to be taken: the new binding time of each defined location is the least upper bound of its previous binding time and of the assignment binding time. If the assignment is unambiguous, the new binding time of the defined variable is the assignment binding time. This is achieved by setting all of the defined locations to the binding time of the assignment and then joining this state with a state where all of the unambiguous definitions are set to bottom. Updating the state for each assignment is how a flow sensitive analysis allows a variable bound to a dynamic value to become static.

To compute the proper safe approximation of the state at a join point, the binding time of each location possibly defined within the conditional or loop statement is the least upper bound of its previous binding time and of the binding time of the conditional or loop test, given by *exp-bt*($e, state$) (see Fig. 3.5). A second transfer function, $t_{e,s}()$ (also in Fig. 3.4), takes control dependencies into account to compute the state at join points. If the specializer does not duplicate continuations, as is the case for Tempo, join points exist at the end of conditional statements and loops. If a test is dynamic, all the locations possibly defined in the scope of the test are considered dynamic. In case of a static test, the join operation has no effect; the transfer function is the identity function.

For instance, let us consider the case of a variable which is assigned a static value in a branch of a conditional statement whose test is dynamic. At specialization time, the value of the variable after the join point will remain unknown. At execution time, if the branch is taken, the variable will be assigned a new value; if not, it will keep the value it had before entering the conditional statement. Such a variable has to be considered dynamic at the end of the branch. In general, taking for each location possibly defined in a branch the least upper bound of its use binding time and of the binding time of the test performs the proper safe approximation.

For sequence of statements, the initial state is passed into the first statement. The output state of each statement is used as the input state for the next statement. Finally, the output state of the last statement is the output state of the sequence.

Inter-procedural

Context sensitivity is obtained by duplicating function definitions and the corresponding data-flow equations according to the different calling contexts encountered in the program for a given function. Different instances of the same function definition are distinguished by subscripting function program point f with its identifier id and the specific calling context ctx, producing a context sensitive function program point $f_{id,ctx}$.

A function call uses the statement state *in(s)* to create the calling context,

ctx. This calling context combines the parts of the state relevant to the call input by computing the binding time of each actual parameter and associating it to each corresponding formal parameter as well as the binding times of the non-local locations that may be *used* by the function, taking into account other nested calls.

The call *formal-loc*(i, id) simply returns the location associated to the i^{th} formal parameter of function *id*. Since the number of locations defined by a given program is finite and the height of the binding-time lattice is also finite, the set of these contexts is finite.

The output state *out(s)* of a call statement is obtained by updating the binding times of the defined non-local locations that may be *defined* by the call, again taking into account any other nested call. Additionally, the binding time of the function's return location, $RETURN(f_{id_2,ctx})$, is propagated interprocedurally by updating location id_1 in the state with its value.

The output statement state of a return statement s is the empty state $\{\}$. It is the return state *ret-out(s)* which is propagated to the exit point of the function. At each return statement, the statement state is joined with the previously computed return state. These return states are propagated along through the *ret-in(s)* and *ret-out(s)* input and output return states until the end of a function definition is reached.

Let us, for instance, consider a conditional statement with return statements in both branches as well as paths that do not return. At the join point, only the non-returning paths should be joined and propagated to the next statement as the output state of the conditional statement. On the other hand, all the returning paths should be joined too, taking into account the possibility of `return` statements under dynamic control.

Fig. 3.3 shows that the remaining equations for the return states *ret-in(s)* and *ret-out(s)* are very similar to the ones relating statement states *in(s)* and *out(s)*. For the assignment statement and function calls, *ret-out(s)* is simply equal to *ret-in(s)*. For conditional statements, loops, and blocks, *ret-in(s)* and *ret-out(s)* are related by exactly the same equations as $in(s)$ and $out(s)$. In particular, the transfer function $t_{e,s}()$ deals with return locations under conditional control.

3.2 Binding-time annotations

Solving these data flow equations produces a statement state for each statement, with which the program is then annotated, as seen in Fig. 3.5. The binding time for most constructs is the least upper bound of their subcomponents. For example, the binding time for an assignment is the least upper bound of the binding times of its left hand side and right hand side expressions, and the binding time of a conditional is the least upper bound of the binding times of the test expressions and both branches.

function definition:

id^f `(` decls `)` stmt^s:

$\mathit{fun\text{-}body\text{-}bt}(f) = \mathit{stmt\text{-}bt}(s)$

$\mathit{fun\text{-}return\text{-}bt}(f) = \bigsqcup_{s1 \in \mathit{returns}(s)} \mathit{stmt\text{-}bt}(s1)$

statements:

lexp^{e_1} `=`^{s} exp^{e_2}:

$\mathit{stmt\text{-}bt}(s) = \mathit{lexp\text{-}bt}(e_1, \mathit{in}(s)) \sqcup \mathit{exp\text{-}bt}(e_2, \mathit{in}(s))$

`if`^{s} `(` exp^e `)` $\text{stmt}_1^{s_1}$ `else` $\text{stmt}_2^{s_2}$:

$\mathit{stmt\text{-}bt}(s) = \mathit{exp\text{-}bt}(e, \mathit{in}(s)) \sqcup \mathit{stmt\text{-}bt}(s_1) \sqcup \mathit{stmt\text{-}bt}(s_2)$

`do`^{s} stmt^{s_0} `while` `(` exp^e `)`:

$\mathit{stmt\text{-}bt}(s) = \mathit{exp\text{-}bt}(e, \mathit{out}(s_0)) \sqcup \mathit{stmt\text{-}bt}(s_0)$

`{` $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ `}`^{s}:

$\mathit{stmt\text{-}bt}(s) = \bigsqcup_{1 \le i \le n} \mathit{stmt\text{-}bt}(s_i)$

id_1`=`^{s_1} $id_2^{s_2}$ `(`$\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$`)`:

$\mathit{stmt\text{-}bt}(s_2) = \mathit{fun\text{-}body\text{-}bt}(f_{id_2})$

$\mathit{stmt\text{-}bt}(s_1) = \mathit{fun\text{-}return\text{-}bt}(f_{id_2})$

`return`^{s} exp^e:

$\mathit{stmt\text{-}bt}(s) = \mathit{exp\text{-}bt}(e, \mathit{in}(s))$

left-hand side expression:

id^e:

$\mathit{lexp\text{-}bt}(e, \mathit{state}) = S$

$\mathit{lexp}^{e_0}.^e\mathit{id}$:

$\mathit{lexp\text{-}bt}(e, \mathit{state}) = \mathit{lexp\text{-}bt}(e_0, \mathit{state})$

`*`^{e} exp^{e_0}:

$\mathit{lexp\text{-}bt}(e, \mathit{state}) = \mathit{exp\text{-}bt}(e_0, \mathit{state})$

right-hand side expression:

$\mathit{loc\text{-}bt}(\mathit{loc}, \mathit{state}) = \mathit{struct\text{-}loc}(\mathit{loc}) \rightarrow \bigsqcup_{l \in \mathit{locations}(\mathit{type}(\mathit{loc}))} \mathit{state}(l); \mathit{state}(\mathit{loc})$

const^e:

$\mathit{exp\text{-}bt}(e, \_) = S$

id^e:

$\mathit{exp\text{-}bt}(e, \mathit{state}) = \mathit{loc\text{-}bt}(\mathit{id}, \mathit{state})$

$\mathit{lexp}^{e_0}.^e\mathit{id}$:

$\mathit{exp\text{-}bt}(e, \mathit{state}) = \mathit{lexp\text{-}bt}(e_0, \mathit{state}) \sqcup \mathit{loc\text{-}bt}(\mathit{type}(e_0).\mathit{id}, \mathit{state})$

`&`^{e} lexp^{e_0}:

$\mathit{exp\text{-}bt}(e, \mathit{state}) = \mathit{lexp\text{-}bt}(e_0, \mathit{state})$

`*`^{e} exp^{e_0}:

$\mathit{exp\text{-}bt}(e, \mathit{state}) = \mathit{exp\text{-}bt}(e_0, \mathit{state}) \sqcup (\bigsqcup_{\mathit{loc} \in \mathit{aliases}(e)} \mathit{state}(\mathit{loc}))$

$\text{exp}_1^{e_1}$ bop^e $\text{exp}_2^{e_2}$:

$\mathit{exp\text{-}bt}(e, \mathit{state}) = \mathit{exp\text{-}bt}(e_1, \mathit{state}) \sqcup \mathit{exp\text{-}bt}(e_2, \mathit{state})$

Figure 3.5: Binding-time phase — binding-time annotations

Since our analysis is return sensitive, two binding times are computed for a function call. These two binding times are computed at the function definition but used to annotate the function's call site. The first, computed by *fun-body-bt()*, determines if the body of the function is completely static. A completely static body is simply evaluated at specialization time. The second binding time, produced by *fun-return-bt()*, determines if all the return statements are static and under static control. In this case, the value returned by the function is static and it can be exploited during specialization, even if the body of the function is dynamic and therefore residualized.

Additionally, some constructs do not have any subcomponents, such as a variable as a left hand side expression or a constant right hand side expression. These constructs are always considered static, independent of the binding-time state, since they can always be computed at specialization time. Recall that for a variable as a left hand side expression, only the address of the variable—not its contents—is needed for it to be evaluated.

3.3 Summary

The resulting binding-time annotated code indicates whether constructs depend on known or unknown values. A construct is annotated static if it only depends on known values, in which case it can be evaluated at specialization time. We have already mentioned, however, that not all constructs which *can be* evaluated *should be* evaluated. For example, static non-liftable values in dynamic contexts should be residualized. Therefore, the binding-time annotated code must be analyzed in the transformation phase to determine the correct transformation for each construct. The resulting transformation annotated code is then used to drive specialization.

Chapter 4

Transformation Phase

Binding-time annotations indicate whether constructs depend on known or unknown values at specialization time. The transformation phase uses binding-time information to calculate a transformation for each construct. Specifically, the binding time of each construct and the context in which it appears is used to determine its transformation. Further, the transformations computed for the uses of a variable are collected and used to determine the transformation for the definition of the variable.

Whereas the binding-time phase propagates information *forwards* from a variable definition to its uses, the transformation phase propagates information *backwards*, from variable uses to their definition. Therefore, the analysis that computes transformations is backwards.

In this Chapter we present this backwards analysis. We first describe the states used to compute this information, and then give the data flow equations describing the analysis. The resulting program is annotated with transformations.

States The transformations computed for variable uses are collected and propagated using a transformation state, $TrState = Location \to Tr$, where Tr is the transformation lattice domain previously defined in Fig 2.10. We shall denote $\setminus$ the binary operator from $TrState \times Locations \to TrState$ setting to bottom a set of locations.

Again, we use a graph representation of states. The lookup function takes a graph (a set of location/transformation pairs) and a location, and returns the corresponding transformation. A location which does not occur in the graph is considered to be undefined, in which case the lookup function simply returns the bottom element.

The join operator $\underline{\cup}$ on transformation states is defined as a pointwise application of the least upper bound operator $\cup$ on the *TrState* function space range. $(TrState, \underline{\cup})$ is a join semi-lattice of finite length and the set of transfer functions generated by closure under joins and compositions is a monotone operation space. The set of equations corresponding to any given program can be solved using one

of the standard iterative algorithms.

4.1 Data Flow Equations

For each construct, a transformation and a state are computed. Computing a transformation not only depends on the binding-time of the construct, but also on its context and whether or not it is liftable, as explained in Fig. 2.11. A static construct is annotated $\{E\}$ if it is used in a static context or if it is liftable. A non-liftable static construct used in a dynamic context, as well as all dynamic constructs, are annotated $\{R\}$.

Transformations for the uses and the definition of a variable must be consistent. That is, a variable with uses that are evaluated must have a definition that is evaluated. Likewise, uses that are residualized must have a corresponding residualized definition. If a static non-liftable variable is used in a dynamic context, the variable will be residualized. Therefore its definition must also be residualized. If there are other uses of the variable which are evaluated, its definition must also be evaluated.

Relating the transformations computed for variable uses to their definitions is done via the transformation state. After the transformation of a variable use is computed, the state is updated to include the variable and its corresponding transformation. Updating the store consists of taking the least upper bound of the transformation computed for the variable and the existing transformation in the store for the variable, according to the transformation lattice shown in Fig. 2.10.

The state is propagated backwards within a procedure, updating the state with each additional variable use and its transformation. When variable definitions are reached, the transformations collected for the variable uses determines the transformations for the definitions. The transformation for a variable's definition is the transformation given by looking up the variable in the state. Additionally, if the definition is unambiguous, the state is updated by setting the defined variable to bottom, indicating that all uses of the variable are defined.

Since variable uses that occur in one function may be defined in another function, the state must also be propagated inter-procedurally. A function is analyzed, starting at the end of the function, with respect to the state at the function's call site. Within the body of the function, the state at each return statement is set to the state at the function's call site. The analysis then propagates states intra-procedurally until the beginning of the function is reached, and the resulting state is then propagated back to the function call site. The initial state for the main function in a program is initialized with information expressing how variables are used following the program. If no variables are used after the end of the program, the inital state is empty.

The data-flow equations for the transformation phase are given in Fig. 4.1

with the basic transfer functions given in Fig. 4.2.

Function definitions

Each function is analyzed with respect to an output state *out*(f), used to define the output state *out(s)* with which the body of the function is analyzed. After the body is analyzed, the input state of the body *in(s)* is used to define the input state of the function, *in(f)*, by extracting the non-local variables used in the function and their corresponding transformations.

States are computed and propagated within the body of each function. This aspect of the analysis is expressed by relating the state *out*(s) at the output of the statement with the state $in(s)$ at the input of a statement. The output state of a function *ret-out*(f) does not vary with the body of a function, and is used, to set the statement state at each return statement.

Intra-procedural

The transfer function $t_s()$ is used to compute the input state of an assignment $in(s)$ from its output state *out*(s) by adding transformations for variables used in the assignment statement and removing transformations for variables defined by the assignment. The function *def-tr*() returns the transformation of an assignment by looking up the transformation of the variable defined by the assignment. Since multiple variables may be defined by the same assignment due to aliasing, the least upper bound of the transformations for all the locations possibly defined by the assignment is computed. The function *def-state*() computes a new transformation state based on the assignment transformation and the variable uses on the left-hand and right-hand sides of the assignment statement. In case of an unambiguous assignment, the transformation corresponding to the defined location is set to {} in the state; there is no transformation to be propagated up.

The function *def-state*() relies on two other functions, *lexp-state*() and *exp-state*(). The function *lexp-state*() takes a program point of a left-hand side expression and the transformation of the context. Its returns a location/transformation pair with which the transformation state will be updated. An identifier which occurs as a left-hand side expression does not add any transformations to the state since only the variable's address, and not its contents, is used. The transformation for a structure field access is computed by calling *lexp-state*() on the structure expression. A pointer dereference is similarly computed by calling *exp-state*() on the pointer expression. The function *exp-state*() performs the same functionality for right-hand side expressions. Constant expressions use no variables and therefore do not add any location/transformation pair to the state.

Calculating the state for a variable identifier uses the function *loc-state*(). This function takes a location and a context transformation, and returns the

function definition:

id^f `(` decls`)` $body^s$:

$in(f) = \{(loc, \mathit{lookup}(in(s), loc)) \mid loc \in \mathit{used\text{-}non\text{-}locals}(f)\}$

$out(s) = out(f)$

statements:

lexp^{e_1} `=`^{s} exp^{e_2}:

$in(s) = t_s(out(s))$

`if`^{s} `(` exp^e `)` $\mathrm{stmt}_1^{s_1}$ `else` $\mathrm{stmt}_2^{s_2}$:

$in(s) = t_e(in(s_1) \underline{\cup} in(s_2))$

$out(s_1) = out(s)$

$out(s_2) = out(s)$

`do`^{s} stmt^{s_0} `while (` exp^e `)`:

$in(s) = in(s_0)$

$out(s_0) = t_e(out(s)) \underline{\cup} t_e(in(s_0))$

`{` $\mathrm{stmt}_1^{s_1} \ldots \mathrm{stmt}_n^{s_n}$ `}`^{s}:

$in(s) = in(s_1)$

$out(s_i) = in(s_{i+1}), 1 \leq i < n$

$out(s_n) = out(s)$

id_1 `=`^{s} id_2 `(` $\mathrm{exp}_1^{e_1} \ldots \mathrm{exp}_n^{e_n}$ `)`:

$in(s) = t_s(in(e_1))$

$out(e_i) = in(e_{i+1}), 1 \leq i < n$

$out(e_n) = (out(s) \setminus \mathit{used\text{-}non\text{-}locals}(id_2)) \underline{\sqcup} in(f_{id_2,ctx})$

$out(f_{id_2,ctx}) = ctx$

$ctx = \{(loc, \mathit{lookup}(out(s), loc)) \mid loc \in \mathit{def\text{-}non\text{-}locals}(f)\}$

`return`^{s} exp^e:

$in(s) = t_e(out(s))$

Figure 4.1: Transformation phase — data-flow equations

$$t_s(state) = (\textit{def-state}(s, \textit{def-tr}(s, state)) \underline{\cup}\ state) \setminus \textit{unambiguous-defs}(s)$$

$$t_e(state) = \textit{exp-state}(e, \textit{exp-tr}(e)) \underline{\cup}\ state$$

Figure 4.2: Transformation phase — transfer functions

appropriate location/transformation pair. If the location is for a structure, it returns the transformation corresponding to the least upper bound of all the structure fields. Otherwise, the location is for a variable, in which case the context transformation is returned.

The transformation for a variable identifier which occurs in an $\{E\}$ context is determined by calling *loc-state*() with the context. If the location is a structure and any fields are dynamic, the transformation $\{R\}$ is returned. Otherwise, it returns $\{E\}$. A variable identifier which occurs in a dynamic context is only considered $\{E\}$ if the variable is static and its value can be lifted. Otherwise, *loc-state*() is called with the context. This is how non-liftable static uses in dynamic contexts are residualized.

The subsequent expressions are treated similarly. For expressions that consist of multiple subcomponents, a transformations state is computed for each subcomponent and the results are merged with the $\underline{\cup}$ operator. For dereference pointer expressions, alias information is given by the function $aliases()$.

The data flow equations for the other statements relate their inputs states $in(s)$ from their output states $out(s)$ similarly. The output state is propagated backwards through the construct, adding transformations for variables when they are used and setting their transformation to bottom when they are unambiguously defined.

For a conditional statement, the output states $out(s1)$ and $out(s2)$ of each branch is defined in terms of the output state $out(s)$ of the conditional statement. A second transfer function, $t_e()$ (also in Fig. 4.2), is used to join the input states $in(s1)$ and $in(s2)$ of the branches to define the input state $in(s)$ of the conditional. The loop statement similarly uses this transfer function to define its input state.

For a sequence of statements, the output state is passed to the last statement. The input state of each statement is used as the output state of the previous statement. Finally, the output state of the first statement is the output state of the sequence.

Inter-procedural

In a context-sensitive analysis, the calling context consists of the information at each call site that affects the analysis of a function. Since the transformation state

statements:

$\mathit{def\text{-}tr}(s, \mathit{state}) = \bigcup_{\mathit{loc} \in \mathit{defs}(s)} \mathit{lookup}(\mathit{state}, \mathit{loc})$

$\mathit{def\text{-}state}(s, \mathit{tr}) = \mathit{lexp\text{-}state}(\mathit{lexp}(s), \mathit{tr}) \underline{\bigcup} \mathit{exp\text{-}state}(\mathit{exp}(s), \mathit{tr})$

expressions:

$\mathit{loc\text{-}state}(\mathit{loc}, \mathit{tr}) = \mathit{is\text{-}a\text{-}struct\text{-}loc}(\mathit{loc}) \rightarrow \underline{\bigcup}_{\mathit{loc'} \in \mathit{locations}(\mathit{type}(\mathit{loc}))} \{(\mathit{loc'}, \mathit{tr})\}; \{(\mathit{loc}, \mathit{tr})\}$

left-hand side expressions:

$\mathit{lexp\text{-}state}(\mathit{id}, \_) = \{\}$

$\mathit{lexp\text{-}state}(\mathit{lexp}^{e_0}.\mathit{id}, \mathit{tr}) = \mathit{lexp\text{-}state}(\mathit{lexp}^{e_0}, \mathit{tr})$

$\mathit{lexp\text{-}state}(*\mathit{exp}^{e_0}, \mathit{tr}) = \mathit{exp\text{-}state}(\mathit{exp}^{e_0}, \mathit{tr})$

right-hand side expressions:

$\mathit{exp\text{-}state}(\mathit{const}^{e}, \_) = \{\}$

$\mathit{exp\text{-}state}(\mathit{id}^{e}, \{\mathrm{E}\}) = \mathit{loc\text{-}state}(\mathit{id}, \{\mathrm{E}\})$

$\mathit{exp\text{-}state}(\mathit{id}^{e}, \mathit{tr})$

$= (\mathit{exp\text{-}bt}(e) = S \wedge \mathit{is\text{-}liftable}(\mathit{type}(e))) \rightarrow$

$\{(\mathit{id}, \{E\})\}$

;

$\mathit{loc\text{-}state}(\mathit{loc}, \mathit{tr})$

$\mathit{exp\text{-}state}(\mathit{lexp}^{e_0}.{}^{e}\mathit{id}, \mathit{tr}) = \mathit{lexp\text{-}state}(\mathit{lexp}^{e_0}, \mathit{tr}) \underline{\bigcup} \mathit{loc\text{-}state}(\mathit{type}(e_0).\mathit{id}, \mathit{exp\text{-}tr}(e))$

$\mathit{exp\text{-}state}(\&\mathit{lexp}^{e_0}, \mathit{tr}) = \mathit{lexp\text{-}state}(\mathit{lexp}^{e_0}, \mathit{tr})$

$\mathit{exp\text{-}state}(*\mathit{exp}^{e_0}, \{\mathrm{E}\}) = \mathit{exp\text{-}state}(\mathit{exp}^{e_0}, \{\mathrm{E}\})$

$\mathit{exp\text{-}state}(*{}^{e}\mathit{exp}^{e0}, \mathit{tr})$

$= (\mathit{exp\text{-}bt}(e_0) = S \wedge \mathit{is\text{-}liftable}(\mathit{type}(e))) \rightarrow$

$\underline{\bigcup}_{\mathit{loc} \in \mathit{aliases}(e)} \{(\mathit{loc}, \{\mathrm{E}\})\}$

;

$\underline{\bigcup}_{\mathit{loc} \in \mathit{aliases}(e)} \{(\mathit{loc}, \mathit{tr})\}$

$\mathit{exp\text{-}state}(\mathit{exp}_1^{e_1} \mathit{bop}^{e} \mathit{exp}_2^{e_2}, \mathit{tr}) = \mathit{exp\text{-}state}(\mathit{exp}_1^{e_1}, \mathit{tr}) \underline{\bigcup} \mathit{exp\text{-}state}(\mathit{exp}_2^{e_2}, \mathit{tr})$

Figure 4.3: Transformation phase — auxiliary functions

propagates transformations from variable uses to variable definitions, the only information relevant for analyzing a function is the transformations corresponding to variables used outside of the function and defined within the function. The calling context ctx is therefore defined as the subset of the state corresponding to the non-local variables defined within a function, and is used as the output state $out(f_{id,ctx})$ of the function.

Similarly, after a function is analyzed, the only part of the state that needs to be propagated interprocedurally is the non-local variables used within the function, since these may be defined outside of the function. Therefore, the non-local variables are removed from output state $out(s)$ at the function call and replaced with their corresponding transformations in the function's input state $in(f_{id,ctx})$.

The resulting state is used to treat bindings between formal parameters and actual parameters. Definitions of variables may not only come from assignments (as seen in the intra-procedural analysis) but also from an actual/formal binding. Because of this, the transformation of a parameter definition, like that of an assignment definition, can be $\{E\}$, $\{R\}$, $\{E, R\}$, or $\{\}$. Again, like the assignment definition, this information summarizes the uses of the formal parameter within the function body. As well, like an assignment, the specializer needs to be capable of evaluating and residualizing the same actual/formal binding.

After a transformation is calculated for each parameter, the input state $in(s)$ for the call statement is defined, using the transfer function $t_s()$ to take the assignment into account.

At each return statement, the state is set to the output state $out(s)$ corresponding to the output state $out(f)$ of the function. This state is not modified within a function and therefore the state at each return statement within a function is always the same. For each return statement, the transfer function $t_e()$ is used to take the expression returned into account.

4.2 Transformation Annotations

4.3 Summary

The transformation phase determines a transformation for each construct. This is achieved by taking a binding-time annotated program performing a backwards analysis that performs two tasks. First, the transformation for each variable use is determined by taking into account the binding time of the variable and the context in which it is used. Second, transformations for variable definitions is computed. The resulting annotated program is then used for specialization. Constructs annotated $\{E\}$ will be evaluated. Constructs annotated $\{R\}$ will be

function definition:

id^f `(` decls `)` stmt^s:

$\mathit{fun\text{-}body\text{-}tr}(f) = \mathit{stmt\text{-}tr}(s)$

$\mathit{fun\text{-}return\text{-}tr}(f) = \bigsqcup_{s1 \in \mathit{returns}(s)} \mathit{stmt\text{-}tr}(s1)$

statements:

lexp^{e_1} `=`^{s} exp^{e_2}:

$\mathit{stmt\text{-}tr}(s) = \mathit{lexp\text{-}tr}(e_1, \mathit{in}(s)) \sqcup \mathit{exp\text{-}tr}(e_2, \mathit{in}(s))$

`if`^{s} `(` exp^e `)` $\text{stmt}_1^{s_1}$ `else` $\text{stmt}_2^{s_2}$:

$\mathit{stmt\text{-}tr}(s) = \mathit{exp\text{-}tr}(e, \mathit{in}(s)) \sqcup \mathit{stmt\text{-}tr}(s_1) \sqcup \mathit{stmt\text{-}tr}(s_2)$

`do`^{s} stmt^{s_0} `while (` exp^e `)`:

$\mathit{stmt\text{-}tr}(s) = \mathit{exp\text{-}tr}(e, \mathit{out}(s_0)) \sqcup \mathit{stmt\text{-}tr}(s_0)$

`{` $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ `}`^{s}:

$\mathit{stmt\text{-}tr}(s) = \bigsqcup_{1 \le i \le n} \mathit{stmt\text{-}tr}(s_i)$

id_1`=`$^{s_1} id_2^{s_2}$`(`$\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$`)`:

$\mathit{stmt\text{-}tr}(s_2) = \mathit{fun\text{-}body\text{-}tr}(f_{id_2})$

$\mathit{stmt\text{-}tr}(s_1) = \mathit{fun\text{-}return\text{-}tr}(f_{id_2})$

`return`^{s} exp^e:

$\mathit{stmt\text{-}tr}(s) = \mathit{exp\text{-}tr}(e, \mathit{in}(s))$

left-hand side expressions:

$\mathit{lexp\text{-}tr}(id, \{E\}) = \{E\}$

$\mathit{lexp\text{-}tr}(id, \_) = \{R\}$

$\mathit{lexp\text{-}tr}(lexp^{e_0}.id, tr) = \mathit{lexp\text{-}tr}(lexp^{e_0}, tr)$

$\mathit{lexp\text{-}tr}(*exp^{e_0}, tr) = \mathit{exp\text{-}tr}(exp^{e_0}, tr)$

right-hand side expressions:

$\mathit{loc\text{-}tr}(loc, tr) = \mathit{is\text{-}a\text{-}struct\text{-}loc}(loc) \rightarrow \bigcup_{\mathit{loc'} \in \mathit{locations}(\mathit{type}(loc))} \; tr \; ; \; tr$

$\mathit{exp\text{-}tr}(\mathit{const}^e, \_) = \{E\}$

$\mathit{exp\text{-}tr}(id^e, \{E\}) = \mathit{loc\text{-}tr}(id, \{E\})$

$\mathit{exp\text{-}tr}(id^e, tr) = (\mathit{exp\text{-}bt}(e) = S \wedge \mathit{is\text{-}liftable}(\mathit{type}(e))) \rightarrow \{E\} \; ; \; \mathit{loc\text{-}tr}(loc, tr)$

$\mathit{exp\text{-}tr}(lexp^{e_0} .^e id, tr) = \mathit{lexp\text{-}tr}(lexp^{e_0}, tr) \; \bigcup \; \mathit{loc\text{-}tr}(\mathit{type}(e_0).id, \mathit{exp\text{-}tr}(e))$

$\mathit{exp\text{-}tr}(\&lexp^{e_0}, tr) = \mathit{lexp\text{-}tr}(lexp^{e_0}, tr)$

$\mathit{exp\text{-}tr}(*exp^{e_0}, S) = \mathit{exp\text{-}tr}(exp^{e_0}, S)$

$\mathit{exp\text{-}tr}(*^e exp^{e0}, tr) = (\mathit{exp\text{-}bt}(e_0) = S \wedge \mathit{is\text{-}liftable}(\mathit{type}(e))) \rightarrow \{E\} \; ; \; \mathit{loc\text{-}tr}(loc, tr)$

$\mathit{exp\text{-}tr}(exp_1^{e_1} bop^e exp_2^{e_2}, tr) = \mathit{exp\text{-}tr}(exp_1^{e_1}, tr) \; \bigcup \; \mathit{exp\text{-}tr}(exp_2^{e_2}, tr)$

Figure 4.4: Transformation phase — annotations

residualized. Assignments annotated $\{E, R\}$ will be both evaluated and residualized. Assignments annotated $\{\}$ correspond to dead code and are simply discarded.

Chapter 5

Tempo

The analyses described in this dissertation have been implemented and integrated into Tempo, our partial evaluator for C. This chapter gives an overview of the entire Tempo system. First we give a summary of the main motivations behind the development of this partial evaluator, followed by a description of the system. We explain how the two-phase binding-time analysis fits in with the other phases of the analysis stage, and show how the results of these analyses drive different specialization stages.

5.1 Motivation

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

Realistic Applications Until now, most partial evaluators have been developed without focusing on realistic applications. As partial evaluation addresses more realistic programs, it also faces new challenges when it tackles existing real-size application programs. This new situation reveals that the usual, general-purpose set of analyses and 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. Programs written in realistic languages like C expose a very wide variety of situations where partial evaluation can be applied. As partial evaluators treat richer and richer languages, their size and complexity

increase drastically. 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 functions implement 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 functions which perform the actual transactions. Then, when the session is closed, the invariants become invalid and the specialized functions 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.

The design of Tempo address each of these issues. Its analyses and transformations are capable of effectively treating realistic programs.

A Uniform Approach. This approach is off-line in that it separates the partial evaluation process into two parts: an analysis stage followed by a specialization stage [JSS89, CD93]. The former part includes a binding-time phase aimed at determining the static and dynamic computations for a given program and a division (static/dynamic) of its input, followed by a transformation phase which determines transformations for each computation. These transformations are then used to determine a *specialization action* for each construct in the program [CD90].

The specialization stage of the partial evaluator performs the transformations corresponding to the action-analyzed program, according to the given specialization values. Specialization is then merely guided by the information produced by the analysis stage. 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 analysis stage produces program transformations. Just like a run-time system executes compiled code, the specialization stage executes the program transformations produced by the analysis stage. Just like a compiled code is run many times with respect to different input values, an action-analyzed 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 analysis and spe-

cialization, 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 compiled. The latter form of specialization corresponds to producing a generating extension [And94, Ers77].

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, a strategy has been developed to perform run-time specialization based on actions [CN96]. In essence, an action-annotated program is used to automatically generate 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 analysis 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 analysis stage of the system.

Systems Applications. Tempo has been targeted toward a particular area, namely systems applications. More precisely, the features of Tempo have been guided by program patterns based on actual studies of numerous systems programs and tight collaboration with systems researchers. As a result, the accuracy of critical analyses such as alias analysis and binding-time analysis are adequate for typical systems programs. Also, the program transformations address the important cases in such 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, systems programs are large enough to reach the limit of what a state of the art analysis, such as an alias analysis, can process [WL95]. We have therefore developed support to enable one to specialize pieces of a large system.

Summary. Tempo offers a series of contributions regarding the design and architecture of a partial evaluator for a real language. We present an architecture for partial evaluators which is powerful enough to be used for realistic languages and real-size application programs. 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, systems applications. This strategy allows us to better ensure the applicability of partial evaluation. Although dedicated to a particular application area, the architecture is nonetheless open: new program transformations can easily be introduced at the action analysis level. Also, 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.

In the next section the analysis stage is presented. The subsequent section describes the different specialization stages.

5.2 Analysis Stage

As shown in Fig. 5.1, the analysis stage consists of four main phases: the front-end, the alias, definition, and use/def analyses, the binding-time analysis, and the action analysis. The binding-time analysis is divided into two parts: the binding-time phase followed by the transformation phase. The output of the analysis stage is an annotated program that is used to specialize the subject program.

5.2.1 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 [WFW+94]. SUIF is 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 which transforms 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 loop construct and a single conditional construct. It does not include the comma expression nor gotos.

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 [EH94]. Secondly, renaming makes every identifier unique in order to help building up a flattened static store (see Sect. 5.3). This renaming also facilitates function inlining at post-processing time.

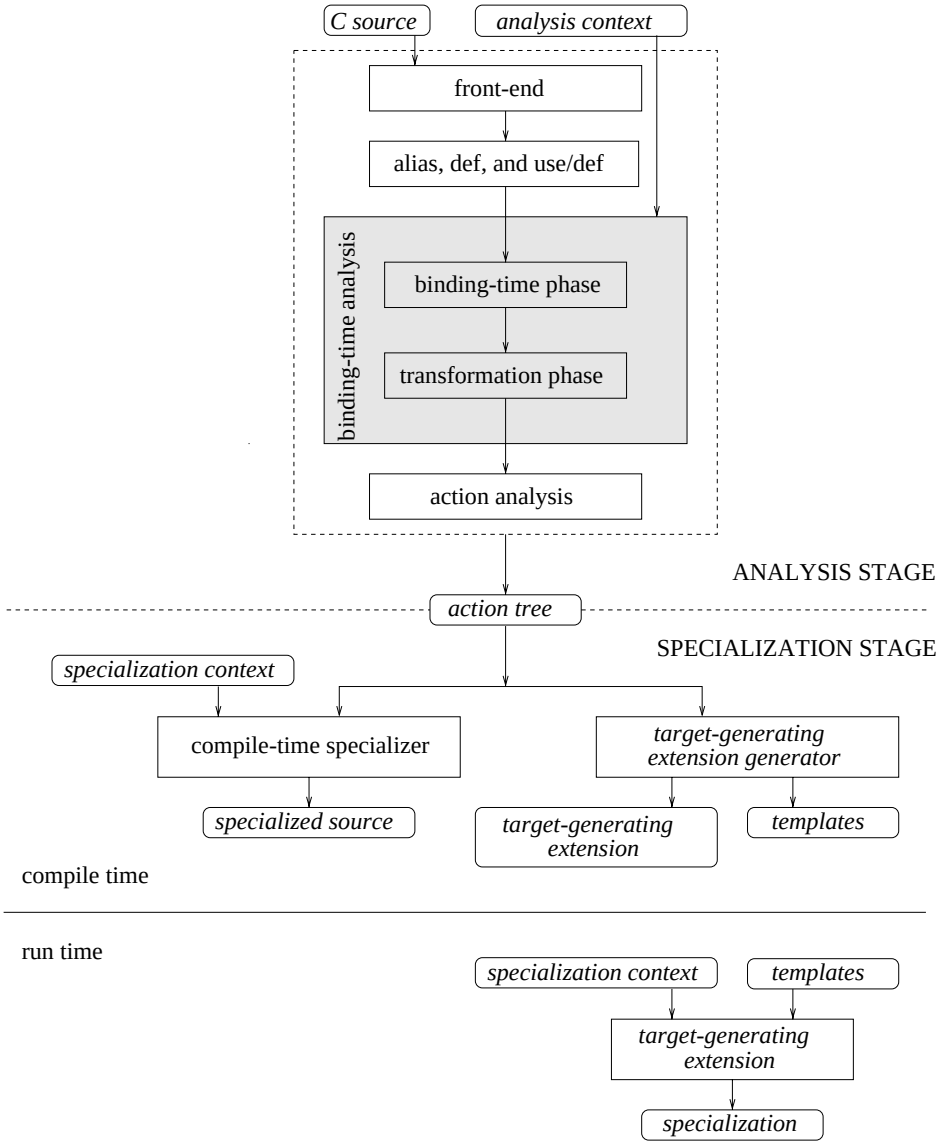

Figure 5.1: A general view of Tempo

5.2.2 Alias, Definition, and Use/Def Analyses

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. Our analysis is very similar to existing ones [EGH94, Ruf95]. It is based on the *points-to* model of aliasing. It is inter-procedural, context-insensitive, and flow-sensitive. 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. Our choice of a context-insensitive analysis is further backed up by the study of Ruf [Ruf95] which shows that empirical benefits of a context-sensitive analysis have still to be measured.

The analysis takes as arguments a set of functions, a goal function, a description of the initial points-to pairs, and a description of the effect of the external functions on the points-to relation. The last two arguments make it possible to perform alias analysis in a module-oriented way. 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 [CWKZ90, Ruf95]).

The alias analysis is complemented by a definition analysis which computes the set of locations which *may* be defined by the statement. Additionally, a use/def analysis computes the non-local store affected by each function call.

5.2.3 Binding-Time Analysis

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

Like the alias analysis, the design of the binding-time analysis was driven by the typical specialization opportunities that occur in systems programs. Our binding-time analysis separates the different tasks performed by a binding-time analysis into two distinct phases. First, the *binding-time phase* determines whether or not a construct depends on static values. At this point, all constructs which only depend on static values are annotated as static. Second, the transformation phase takes into account how the program should be specialized, and annotates the program accordingly. Constructs which are evaluated are annotated $\{E\}$ and those residualized are annotated $\{R\}$. Certain constructs are both evaluated and residualized, in which case they are annotated $\{E, R\}$.

The binding-time analysis phase is flow, context, return, and use sensitive. This precision allows existing, systems programs to be effectively specialized.

5.2.4 Action Analysis

The transformation phase of the binding-time analysis annotates each construct with a transformation. These annotations describe *what* transformations are applied to the constructs at specialization time. The action analysis takes a transformation-annotated program and assigns a specialization *action* to each construct. A specialization action describes *how* a construct is transformed at specialization time.

There are four general specialization actions: *evaluate*, *reduce*, *rebuild*, and *identity*. All constructs annotated with the $\{E\}$ transformation will have an *evaluate* (abbreviated *ev*) action. These constructs only consist of subcomponents annotated $\{E\}$, and are therefore completely evaluated away at specialization time. Constructs with an $\{R\}$ transformation will be annotated with a *reduce*, *rebuild*, or *identity* action, depending on the situation. 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 $\{E\}$. The action *rebuild* (abbreviated *reb*) annotates a construct that needs to be reconstructed but yet includes some $\{E\}$ computations. The action *identity* (abbreviated *id*) which annotates purely $\{R\}$ program fragments. At specialization time, constructs annotated *id* are simply copied verbatim into the residual program.

Constructs annotated with an $\{E, R\}$ transformation will be associated with two specialization actions, one for each part of the transformation. The E part will be translated into an *ev* action. The R part will be *red*, *reb*, or *id*, depending on how the context in which the construct is used.

The action analysis does not change any transformation associated to a construct; it merely describes how the transformation will be performed. Therefore, specializing with respect to a transformation-annotated program and an action-annotated program produces the same residual program. However, specialization actions offer a number of 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. This situation is improved by compiling binding-time information into actions prior to specialization.

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 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.

5.3 Specialization Stage

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

5.3.1 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.

5.3.1.1 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 sometimes 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 in some way 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. Local variables are handled with an explicit stack in order to correctly treat recursive functions. 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 functions (*i.e.*, functions 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 functions. These C functions 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 function 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 [And94] 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 functions are unfolded or when manipulating pointers to dynamic objects. In contrast, Tempo's specializer only includes the static C store used by *ev* functions. Inlining is done as a post-processing operation for compile-time specialization. This approach greatly

simplifies the specializer and does not degrade the quality of the specialized programs.

5.3.1.2 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 functions, 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 [CD90]. A similar compilation process is also known as generating extension [And92, And94, BW93b, Ers77]. One difference is that this latter approach is directly based on binding-time information and thus far more complicated than an action compiler.

5.3.2 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 an action tree 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` → `PLUS(VAR("x"), HOLE(INT))`

Once produced, the tree grammar is used to generate templates [KEH93], *i.e.*, source code fragments parameterized with "holes" for run-time values. For the above tree grammar rule, we obtain the following template:

`x +` *int_hole*

The tree grammar also enables templates to be compiled in their context, *i.e.*, without loosing control flow information, using a standard compiler. 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 executing the static computations, selecting templates, filling holes in templates with run-time values, and relocating jumps between templates. The simplicity of these operations makes run-time specialization a very efficient process which on average requires, according to preliminary experiments, specialized code to be run very few times to amortize the cost of specialization.

A complete description of this approach to run-time specialization is presented elsewhere [CN96, NHCL96].

5.4 Summary

We have presented an approach to designing partial evaluators for realistic languages. Tempo is based on a general approach which consists of separating the process into two parts: analysis stage compiles, after a number of static analyses, input programs into actions, and a specialization stage which then executes these actions to perform the actual specialization. As we have shown, this separation 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 general partial evaluator would not be capable of performing 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.

Chapter 6

Experimental Results

The Tempo partial evaluator is being used to specialize a wide variety of existing, complex, and real-world applications. In this section we summarize the applications that have already been specialized by Tempo. As well, we give a couple of examples taken from these applications that show how key features of Tempo's binding-time analysis, as described in earlier, enable static data to be exploited.

6.1 Applications Specialized by Tempo

We are currently applying Tempo to a wide variety of programs in order to assess its effectiveness. First we present the results of applying Tempo to operating systems programs. Additionally, we obtain significant speedups for several numerical algorithms and image processing routines. Since the analysis phase is used for both compile-time and run-time specialization, we give examples of both. Finally, we explain how Tempo is also being used to generate applications from specifications.

6.1.1 Operating Systems

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 [BSP+95], ExoKernel [EKO95], Scout [MMO+94], and Synthetix [CPW93, CPW94].

Existing work has shown that specializing operating system components with

respect to system state values that occur frequently can produce significant speedups [Mas92, MP89, PMI88, VMC96a]. For example, consider the Unix file system. An operation such as file `read` is written in a generic manner in order to accommodate a wide variety of uses. At each read, the `read` operation interprets the system state to determine the specific, low level operations it should perform. Many parts of the system state become known to the operating system at file open time, such as which file is being read, which process is doing the reading, the file type, the file system block size, if the inode is in memory, etc. The `read` operation can be specialized at open time to eliminate the interpretive overhead which depends on this information.

Another example is the Remote Procedure Call (RPC) protocol [Sun90, RAA+92]. This operating system protocol supplies an interface between a client on a local machine with the server on the remote machine to make a remote procedure look like a local one. The main operations performed by a RPC are coding and decoding data from a machine dependent representation to a network independent representation, and managing the exchange of messages through the network. RPC is written in a generic fashion, including a choice of transport protocol (TCP or UDP), different encoding/decoding implementations, and set of functions to encode/decode information for each specific data type. Before any remote procedure call is made it must be initialized, at which time system state values such as buffer sizes and locations become known, as well as all protocol structures and some encoding/decoding information. Specializing RPC at initialization time with respect to this information eliminates those operations which depend on these values.

The Synthesis operating system combined this kind of specialization with a number of other features to produce significant gains in efficiency, ranging from a factor of 3 to a factor of 56 [Mas92, MP89, PMI88]. We have performed our own studies to determine how much speedup could be obtained from specialization alone. Additionally, we wanted to determine if these speedups could be obtained by specializing an existing operating system, rather than one like Synthesis that was written explicitly to specialize well. Toward this end, the Hewlett-Packard HP-UX `read` file system call was specialized by hand [PAB+95]. Indeed, speedups were obtained, ranging from 1.1 t to 3.6 depending on the size of the data read. Similarly, the Chorus RPC was also specialized by hand [VMC96a]. Specializing the client at initialization time yielded a speedup of 3.7, which resulted in an overall speedup of 1.5 for the entire process.

These investigations demonstrate the value of specialization, but they were all performed by hand. One primary goal of Tempo is to *automatically* achieve these kinds of speedups. As previously mentioned, Tempo was specifically designed to treat systems programs. Tempo has been applied to both the client and server code for the 1984 copyrighted version of Sun RPC [MMV+97, MVM97]. The unspecialized code is about 3200 lines of code.

To assess the effectiveness of automatically specializing such components, we

Array Size	IPX/SunOs			PC/Linux		
	Original	Specialized	Speedup	Original	Specialized	Speedup
20	0.047	0.017	**2.7**	0.071	0.063	**1.1**
100	0.20	0.057	**3.5**	0.11	0.069	**1.5**
250	0.49	0.13	**3.7**	0.17	0.08	**2.1**
500	0.99	0.30	**3.3**	0.29	0.11	**2.6**
1000	1.96	0.62	**3.1**	0.51	0.17	**3**
2000	3.93	1.38	**2.8**	0.97	0.29	**3.3**

Table 6.1: Client marshaling performance in ms

Array Size	IPX/SunOs			PC/Linux		
	Original	Specialized	Speedup	Original	Specialized	Speedup
20	2.32	2.13	**1.08**	0.69	0.66	**1.00**
100	3.32	2.74	**1.20**	0.99	0.87	**1.10**
250	5.02	3.60	**1.39**	1.58	1.25	**1.20**
500	7.86	5.23	**1.50**	2.62	2.01	**1.30**
1000	13.58	8.82	**1.53**	4.26	3.17	**1.34**
2000	25.24	16.35	**1.54**	7.61	5.68	**1.34**

Table 6.2: Round trip performance in ms

compared the performance of original Sun RPC code with the same code specialized by Tempo. The test program, representative of a network application, used remote procedure calls to exchange large amounts of data. This benchmark was run on two different platforms in order to verify that results obtained were not particular to a specific platform. The first platform consisted of two Sun IPX 4/50 workstations running SunOS 4.1.4 with 32 MB of memory, Fore Systems ESA-200 ATM cards and a 100 Mbits/sec ATM link. The second platform consisted of two 166MHz Pentium PC machines running Linux with 96M of memory and a 100Mbits/s Fast-Ethernet network connection. All programs were compiled using `gcc` version 2.7.2 with optimization option `-O2`.

A round-trip remote procedure call consists of client data coding and decoding, message management, and the time to transmit information along the physical wire. Execution times of the unspecialized and specialized client functions were recorded. The speedup of specialized functions are shown in Table 6.1. Additionally, times were also recorded for the entire round-trip of the remote procedure call. This gives the overall speedup, shown in Table 6.2. These times represent the average of 10000 executions.

From Table 6.1, we see that the specialized client stub code is between 2.75 to 3.75 times faster on the IPX/SunOS, and between 1.2 to 3.65 times faster on the PC/Linux. As expected, eliminating the calculations that depend on information available at initialization time significantly improves the marshaling functions. The round-trip times shown in Table 6.2 indicate that the speedup gained from specializing client marshaling has a significant impact on the global speedup. These speedups are not as high, however, since these times reflect the portions of code that were not specialized as well as wire transmission time.

RPC relies heavily on intermediate data structures for modularity and portability, which creates additional opportunities where partial evaluation can be used to improve performance. Instead of eliminating computations which depend on values known at initialization time, Tempo has been used to eliminate these intermediate data structures, a technique referred to as *copy elimination* [VMC+96b]. The approach consists of first instrumenting the program to maintain a *history* of data structure copies, and then automatically transforming the instrumented code. The end result is that, instead of being propagated many times through each RPC layers, data is ultimately only copied once. As this is ongoing research, quantitative measurements on the effectiveness of this approach are not yet available.

6.1.2 Numerical algorithms and image processing routines

Tempo has also been applied to a variety of numerical algorithms and image processing routines. Romberg integration approximates the integral of a function in an interval using trapezoidal estimates. Cubic spline interpolation approximates a function using a cubic polynomial. Chebyshev polynomials approximate continuous functions in a given interval. These three programs can be found in Glück, Nakashige, and Zöchling's work on applying partial evaluation to mathematical algorithms [GNZ95], and are based on algorithms given in Kincaid and Cheney [KC91] and Press *et al.* [PTVF93]. These programs calculate their approximations by iterating their corresponding algorithm until the approximation converges.

The Fourier transformation translates data from a time domain into a frequency domain. The input is a sequence of points and the number of points, and the output is the Fourier coefficient at each point. The fast Fourier transformation (FFT) is the fastest known algorithm for calculating the Fourier transformation. The program we used was written by Dave Edelblute.[1]

Dithering transforms a digital image by reducing the number of colors used in the image. The *ppmdither* program we used is part of the widely available

[1] Available by ftp from `ftp.usc.edu/pub/C-numanal/fft-stuff.tar.gz` in the file `asum.2`.

Application	Invariant	Original Time	CT Specialization Time	**Speedup**
romberg	n=2	7.00	5.00	**1.40**
	n=4	20.00	14.00	**1.43**
	n=6	60.00	40.00	**1.50**
	n=8	196.00	138.00	**1.42**
cubic spline	n=10	11.20	6.20	**1.81**
	n=20	23.30	14.30	**1.63**
	n=30	35.50	20.90	**1.70**
chebyshev	n=5	68.00	12.00	**5.67**
	n=10	246.00	28.00	**8.79**
	n=15	520.00	49.00	**10.61**
	n=20	913.00	75.00	**12.17**
FFT	n=16	43.06	7.97	**5.40**
	n=32	101.81	19.42	**5.24**
	n=64	225.81	45.55	**4.96**
	n=128	483.34	134.61	**3.59**
dither	4,5,9,5	2.06	0.39	**5.28**
in ppm	4,5,9,5	3.03e6	1.72e6	**1.76**

Figure 6.1: Numerical algorithms and image processing routines specialized by Tempo

portable pixmap library.[2]

Compared with the operating systems examples, these programs are all rather small, consisting of under 100 lines of code. These programs were compiled using `gcc` version 2.7.2 with optimization option `-O2`, and run on a Sun SparcStation 20 Model 70 running SunOS 4.1.4 with 96 MB of memory.

Specialization of Romberg integration, cubic spline interpolation, and Chebyshev approximation is performed by considering that the number of iterations, n, is known and then specializing the program with respect to this value. When the number of iterations is known, the control flow of these algorithms becomes known, which allows specialization to perform loop unrolling and constant folding. Additionally, straightline code containing more constants is generated, which can subsequently be compiled into efficient code. Different values of n were tested in each case to observe the impact this makes.

[2]Copyright (C) 1991 by Christos Zoulas. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Available by ftp from *e.g.* `ftp.stanford.edu/class/cs248/netpbm/ppm/ppmdither.c`.

The results, seen in Fig. 6.1, show that specializing Romberg integration and cubic spline interpolation produces minor speedups, ranging from 1.4 to 1.8. In addition to the test and branch instructions eliminated due to loop unrolling, calculations that are eliminated include integer power, floating point multiplication, and floating point division. Speedups for the Chebyshev approximation are more significant; the specialized program runs up to 15 times faster than the original. The main source of this speedup comes from the fact that Chebyshev approximation includes calls to a the cosine library function, which is computationally very expensive. Loop unrolling exposes constant values, which allows these expensive library calls to be performed during specialization.

FFT was specialized by considering that the number of points, n, of the function's input is known. Similarly, this information controls the control flow of the algorithm, and specialization makes this control flow explicit. Loop unrolling and constant propagation provide some speedup. Like the Chebyshev approximation, library functions sine and cosine are eliminated during specialization, which accounts for the significant speedups obtained (see Fig. 6.1).

The dithering program we specialized consists of a setup procedure, which initializes some arrays used in the dithering process, and the dither procedure, which calculates the new value of each pixel in the image. This program takes command-line arguments which specify the size of the dither matrix and the number of shades of red, green, and blue. We used this information to specialize the program (4x4 matrix, 5 shades of red, 9 of green, and 5 of blue). This information permits some constant folding to be performed during specialization. Additionally, substantial speedup comes from a transformation known as *strength reduction*, which transforms integer multiplications into shift and add operations, and divisions by a power of two into shift operations. The speedups achieved from these optimizations are shown in Fig. 6.1.

6.1.3 Application generation

Specialization is also used in various approaches to design *application generators*, programs that automatically translate specifications into applications [BVT+94, BBH+94, TC96]. Tempo plays a key role in the approach presented in [TC96], which presents a framework of application generator design structured into two levels. The first level is based on well established ideas of generic components, which aid structure design and allow code reuse. More specifically, an abstract machine is written, which defines a collection of operations suitable for building domain-specific applications. The second level consists of defining a micro-language which allows these operations to be composed. This micro-language provides an interfaces to the abstract machine, supporting code reuse among similar applications.

Application generation allows applications which are currently being written by hand to be automatically generated. Automatically generated applications

take less time to create, are easier to maintain, and are less error prone. The disadvantage is the additional overhead introduced by the generation process. For example, the abstract machine implementation adds a level of interpretation. Additionally, the approach advocates using an interpreter for the micro-language, which also contains interpretation overhead.

The role of partial evaluation in this approach is to automatically remove these overheads and ultimately generate an application which is competitive in size and speed to a hand-written version. This approach uses Tempo as the partial evaluator, and is currently being applied to automatically generate device drivers for video cards such as SVGA drivers for the XFree86 X11 server. The abstract machine implementation consists of about 1000 lines of code, while the interpreter is roughly 5000 lines. Preliminary test results are encouraging, indicating that most if not all of the interpretive overhead is indeed eliminated by Tempo.

6.2 Template-Based Run-Time Specialization

Run-time specialization allows programs to be optimized by exploiting dynamic information, which has been shown to drastically improve code performance on programs ranging from the Synthesis operating system [PMI88] to graphics programs [Loc87, PLR85]. This improvement is achieved by exploiting run-time invariants to generate code during the execution of the program. Such code can be more optimized than what can be generated based on the information available statically.

Our template-based approach to run-time specialization is efficient and produces high quality code [NHCL96]. We utilize a target-generating extension, as explained in Chapter 1. Since the code generation occurs during the execution of the program, the cost to generate code is minimized. As many computations as possible are performed at compile time. Further, all possible specializations which may be generated are anticipated at compile-time. Doing so allows templates to be identified and precompiled at compile time. As well, a highly optimized specializer is generated, which is specifically streamlined to generate only the specializations needed by assembling these templates. Since the templates are compiled using using existing compiler technology, the code quality of the resulting specialization inherits the code generation techniques of these compilers, such as register allocation and instruction scheduling.

Our approach to run-time specialization is completely automatic. Static analyses do all of the work to determine which transformations to apply and how to apply them. Template identification as well as producing the generating extension are both based on the results of the binding time analysis. It is critical that this analysis is performed at compile-time so that its cost is not occurred at run time. Additionally, the precision of its results directly impacts the quality of the code generated at run time, since it is used to identify the templates as well as

Application	Invariant	Original Time	RT Specialization Generate	Time	**Speedup**	=	CT/RT **Ratio**
romberg	n=2	7.00	61.00	6.00	**1.17**	**61**	**83%**
	n=4	20.00	191.00	16.00	**1.25**	**48**	**88%**
	n=6	60.00	489.00	47.00	**1.28**	**38**	**85%**
	n=8	196.00	1375.00	165.00	**1.19**	**45**	**84%**
cubic spline	n=10	11.20	203.50	8.40	**1.33**	**73**	**74%**
	n=20	23.30	428.00	17.60	**1.32**	**76**	**81%**
	n=30	35.50	653.20	26.90	**1.32**	**76**	**78%**
chebyshev	n=5	68.00	164.00	13.00	**5.23**	**3**	**92%**
	n=10	246.00	561.00	30.00	**8.20**	**3**	**93%**
	n=15	520.00	1199.00	54.00	**9.63**	**3**	**91%**
	n=20	913.00	2110.00	88.00	**10.38**	**3**	**85%**
FFT	n=16	43.06	194.02	11.14	**3.87**	**7**	**72%**
	n=32	101.81	457.83	26.58	**3.83**	**7**	**73%**
	n=64	225.81	1054.98	61.55	**3.67**	**7**	**74%**
	n=128	483.34	2392.48	183.83	**2.63**	**8**	**73%**
dither	4,5,9,5	2.06	24.00	0.50	**4.12**	**17**	**78%**
in ppm	4,5,9,5	3.03e6	24.00	1.68e6	**1.81**	**17**	**103%**

Figure 6.2: Run-time specialization and comparision with compile-time specialization

the generate the generating extension.

We have assessed the effectiveness of this approach by run-time specializing the numerical programs and image processing routines mentioned above. The results are shown in Fig. 6.2 [NHCL96]. For each application specialized, we recorded the amount of time spent at run time to generate the specialized code and the amount of time to execute the specialized code. The speedup is calculated by comparing the execution times of the specialized code with the statically compiled program. The break even point (the column labeled "=") is the number of times the specialized program must be executed to amortize the generation time.

The break even point is the critical factor when assessing run-time specialized code, since it determines how many times a function must be called for run-time specialization to be lucrative. A low break even point is achieved by combining a low generation time and a high speedup.

Looking at Fig. 6.2, we notice that the highest break even points are for Romberg integration and cubic spline interpolation, ranging from 38 to 76. The generation times are not particularly high, but the speedups are relatively low as well. Chebyshev approximation have the best break even points, ranging from 3 to 8, due to the high speedups obtained. Dithering also has a low break even

Original Source Code

```
x = a dynamic expression;
statements in which x is considered dynamic;
```

Transformed Code

```
x = a dynamic expression;
if (x == common_case_value) {
  x = common_case_value;
  statements in which x is considered static;
} else {
  statements in which x is considered dynamic;
}
```

Figure 6.3: Example of generalized partial computation which relies on flow-sensitive analysis

point, 17, due to its low generation time and a fairly good speedup.

6.3 Exploiting Analysis Features

We have seen that applying Tempo to a variety of realistic applications, both at compile time and at run time, produces significant speedups. Let us now give a couple of examples of how the features presented in this paper, flow sensitivity, context sensitivity, return sensitivity, and use-sensitivity, were used to effectively specialize these applications.

Generalized partial computation

The first example illustrates a binding-time improvement that relies on a flow-sensitive analysis. Fig. 6.3 contains a program fragment where variable `x` is assigned a dynamic value, followed by a number of statements that use `x`. Since the assignment renders `x` dynamic, all of its subsequent uses are dynamic as

well. If, however, it is known that there are certain values for `x` that are more common than others, the program can be transformed in such a way to exploit this information. Specifically, a conditional statement is introduced to determine if `x` is in fact equal to some common value. If it is, then by explicitly adding an assignment in the truth branch of the conditional and copying the statements that use `x` into both branches, the statements in the truth branch can be specialized with respect to this common value for `x`. This example of *generalized partial computation* [Fut88, FN88] has proven useful both with the Sun RPC example as well as with application generation. This binding-time improvement is possible because the binding-time analysis is flow sensitive.

Sun RPC

The second example shows how return and use sensitivity are crucial to specialize the excerpt of the Sun RPC client code [Mic89] shown in Fig. 6.4. The RPC code is organized as follows. The initial function `Xdr_bytes()` encodes data in the client buffer by making a call to `Xdr_u_int()` and checking the return value for a success or failure. By following this call interprocedurally, we finally arrive at the function `Xdrmem_putlong` which does the actual encoding. In addition to doing the encoding (performed by the assignment to `*(xdrs->x_private)`), this function also decrements the client buffer size `xdrs->x_handy`, increments the client buffer pointer `xdrs->x_private`, and returns an error value (`0`) if the buffer was empty (`xdrs->x_handy < 0`) and a success value (`1`) if not.

Sun RPC requires a remote procedure call to be initialized before it is called. Many values become known when the call is initialized, such as whether to encode or decode data and the size of the client buffer, which creates opportunities for specialization. In fact, most of the computations in Fig. 6.4 depend on these known values.

If an analysis is not precise enough, however, these known values are lost. A use-*in*sensitive analysis would force almost every construct to be residualized. Fig. 6.4 gives the annotations according to use-insensitive analysis.[3] Since pointer `xdrs` occurs in both static and dynamic contexts, every use is considered dynamic. These dynamic values are propagated throughout the program, causing almost everything to be residualized.

For this example, an analysis must be use sensitive to fully exploit these known values, as seen by the use-sensitive annotations in Fig. 6.5. The assignment to `*(xdrs->x_private)`, which performs the encoding, will be residualized, as well as the pointer to the client buffer. This is because the data to be encoded will only become known at run time. However, all of the other operations, such as those that depend on the client buffer, will be evaluated.

[3]Recall from Chapter 2, constructs in *italics* are to be evaluated and those in **boldface** are to be residualized

```
int Xdr_bytes(...)
{
  ⋮
  if ((Xdr_u_int(xdrs, sizep) != 0) == 0)
    return 0;
  ⋮
}

int Xdr_u_int(struct str1 *xdrs, unsigned int *up)
{
  ⋮
  return Xdr_u_long(xdrs, up);
  ⋮
}

int Xdr_u_long(struct str1 *xdrs, unsigned int *ulp)
{
  ⋮
  if ((int)(xdrs->x_op) == 0)
    return Xdrmem_putlong(xdrs, (int *) ulp);

  if ((int)(xdrs->x_op) == 2)
    return 1;

  return 0;
  ⋮
}

int Xdrmem_putlong(struct str1 *xdrs, int *lp)
{
  xdrs->x_handy = xdrs->x_handy - 4u;
  if (xdrs->x_handy < 0)
    return 0;
  *(xdrs->x_private) = htonl(*lp);
  xdrs->x_private = 4u + xdrs-> x_private;
  return 1;
}
```

Figure 6.4: Use insensitivity for operating systems code

```
int Xdr_bytes(...)
{
  ⋮
  if ((Xdr_u_int(xdrs, sizep) != 0) == 0)
    return 0;
  ⋮
}

int Xdr_u_int(struct str1 *xdrs, unsigned int *up)
{
  ⋮
  return Xdr_u_long(xdrs, up);
  ⋮
}

int Xdr_u_long(struct str1 *xdrs, unsigned int *ulp)
{
  ⋮
  if ((int)(xdrs->x_op) == 0)
    return Xdrmem_putlong(xdrs, (int *)ulp);

  if ((int)(xdrs->x_op) == 2)
    return 1;

  return 0;
  ⋮
}

int Xdrmem_putlong(struct str1 *xdrs, int *lp)
{
  xdrs->x_handy = xdrs->x_handy - 4u;
  if (xdrs->x_handy < 0)
    return 0;
  *(xdrs->x_private) = htonl(*lp);
  xdrs->x_private = 4u + xdrs-> x_private;
  return 1;
}
```

Figure 6.5: Use sensitivity for operating systems code

```
int Xdr_bytes(...)
{
  ⋮
  *(xdrs->x_private) = htonl(*lp);
  xdrs->x_private = 4u + xdrs-> x_private;
  ⋮
}
```

Figure 6.6: Sun RPC example specialized with respect to use sensitive annotation

For example, testing the value (`xdrs->x_handy < 0`) determines if a buffer overflow occurred and can be computed at specialization time. If the analysis is also return sensitive, then the resulting return value can be propagated interprocedurally, causing the initial `if` statement to be reduced. Return sensitivity allows static return values to be propagated interprocedurally, despite the fact that functions contain dynamic side-effects. The residual program produced by specializing a return- and use-sensitive annotated program is given in Fig. 6.6. For this example, all of the intermediate function calls have been inlined during the post-processing phase. The resulting two line program clearly illustrates the potential of applying specializing to operating systems components.

It is not seen in this example, but context-sensitivity was also useful in specializing the RPC application. When function-argument encoding is specialized at initialization time, argument types are known while the contents of the arguments is not known. In another slightly different context, a function pointer is encoded, where both the type and the contents of the pointer are known. Our context-sensitivity analysis created a new context for this second case, where all of the known information was considered static.

6.4 Use-sensitivity vs. Use-insensitivity

We have previously identified operating systems as being good candidates for specialization by hand specializing certain existing operating systems components and achieving significant speedups [PAB+95, VMC96a]. Using current partial evaluation technology to automatically obtain these same speedups is not possible, as existing binding-time analyses are not precise enough to determine the transformations applied by hand. It has always been necessary to write programs from scratch, or to take existing code and adapt it by hand, in order to achieve successful specialization. If partial evaluation is to be successfully applied to existing code, as shown in [MMV+97], a precise binding-time analysis is necessary.

To provide a quantitative assessment of the need for use sensitivity, we com-

pare a use-sensitive analysis with a use-insensitive analysis on several systems programs. We start by giving a brief description of each program considered and present the key invariants used for its specialization.

The first program, `copy_elim`, involves typical message packet manipulation found in network software [VMC+96b]. The packets are handled via pointers to data. Parts of this data is static (typically, some headers), while other parts are dynamic (the message itself). The second program, `minix_read`, is a fragment of the Minix file system implementation [Tan87]. Precisely, we specialized the higher-level routines of the `read()` system call, with respect to a given file and a given size to be read. Here also, the file descriptor is only partially static (*e.g.,* the file mode is static, while the file offset is dynamic), and this structure is handled via a pointer. The third and fourth programs, `client_stub` and `marshaling`, are two code fragments coming from Sun's remote procedure call (RPC) implementation [Sun90]. The program `client_stub` contains the client stub layer, and the program `marshaling` comes from the marshaling layer. We specialized these programs with respect to a given client/server interface, where various descriptors (file descriptors, socket descriptors, protocol descriptors, ...) were partially static.

	total number of lines	expressions			
		total number	% evaluate		
			use-insens.	use-sens.	gain
`copy_elim`	254	352	27%	85%	58%
`minix_read`	314	378	41%	65%	24%
`client_stub`	960	1732	46%	60%	14%
`marshalling`	910	887	38%	78%	40%

Table 6.3: Percentage gain of evaluate statements and expressions of systems programs due to use-sensitivity.

Our experiment consisted of analyzing all four programs twice: first, with a use-insensitive analysis and then with a use-sensitive analysis. After each analysis, statements and expressions annotations were counted. These results are given in Table 6.3 for each program considered, together with its number of lines, statements, and expressions. The number of statements and expressions annotated evaluate are expressed as a percentage. The percentages obtained by the use-insensitive and use-sensitive analysis are used to compute the gain produced by the latter analysis. In the use-sensitive case, statements and expressions found to be both evaluate and residualize (*i.e.*, *E,R*) were not counted as evaluate since they appear in the residual program.

The main observation to make on Table 6.3 is that the use-sensitive analysis detects on all programs between 10% and 58% more evaluate statements and expressions. Since specialization evaluates static constructs and residualizes dy-

namic constructs, the higher percentage of static constructs directly translates into a more optimized residual program. Indeed, the specialization of these programs clearly showed that all their invariants mentioned above were exploited as expected. In some cases, the resulting specialized programs were competitive with respect to their manually specialized version.

Chapter 7

Related Work

In this chapter, we examine research that is related to the static analyses presented in this dissertation. We start by considering existing binding time-analyses and comparing them with our work. As well, we present other analyses that are different but share some aspects with our analyses. We conclude by exploring program adaptation, a field of research in which static analyses may play an important role.

7.1 Binding-time Analysis

There are a number of existing off-line partial evaluators for imperative languages [And92, And94, BGZ94, JGS93a, NP92], as well as for functional languages [Con93c, HM94, JGS93a, RG92]. Certain features of our analyses, such as flow sensitivity, only apply to imperative languages. Other features, such as context and use sensitivity, can also be considered for functional languages. We explore each of these features and how they relate to existing binding-time analyses.

7.1.1 Flow Sensitivity

All existing imperative binding-time analyses that we know of are *flow-insensitive*; that is, one single description of the binding-time state is maintained for an entire program [And92, And94, BGZ94, JGS93a, NP92]. In this case, if a variable is dynamic anywhere in the program, its single description would be dynamic, and therefore the variable would be considered dynamic everywhere in the program. In this paper we have obtained flow sensitivity by writing an analysis that is flow sensitive.

An alternative approach would be to use a flow-insensitive analysis on an intermediate program flow representation that explicitly encodes flow dependencies, such as Single Static Assignment (SSA) [CJR+91]. For example, a binding-time

analysis has been described for a simple imperative language, which obtains flow-sensitivity by using a Program Representation Graph, a representation which contains some of the features of SSA [DRVH95]. The focus of this work is on providing formal semantics and proving safety conditions of binding-time analyses in order to establish a semantic foundation, and therefore implementation or application issues were not considered. It would be interesting to determine if this framework could be adapted to handle real programs, for example, by treating a more realistic language containing pointers, data structures, or functions.

Flow sensitivity does not apply to functional languages since there is no notion of a state or update operations.

7.1.2 Context Sensitivity

Similarly, all existing imperative binding-time analyses are *context insensitive*. Contexts of all the calls to a function are approximated by a single, least precise, context. If a parameter or non-local variable is dynamic at any call site, it will be considered dynamic at every call site. On the other hand, there are a number of existing binding-time analyses for functional languages which are context sensitive, more commonly referred to as *polyvariant* [Con93c, HM94, RG92]. However, a context-sensitive binding-time analysis for an imperative language is more complicated since contexts must include the binding-times of the non-local variables that are read by a function and the state must be updated with respect to non-local variables that are written. This is further complicated by the possibility of definitions being ambiguous due to aliasing.

7.1.3 Return Sensitivity

Return sensitivity, which prevents the side-effect binding time of a function from interfering with its return binding-time, is a new concept which has not previously been explored. We discovered the need for return sensitivity when applying partial evaluation to operating systems code.

Return sensitivity is not applicable for functional languages since pure functions have a return value but do not contain side-effects. However, there are similar situations that do arise for functional languages. For example, an expression may contain some dynamic components but return a static value. Techniques have been developed, however, which pass the continuation of the expression into the body of the expression in order to exploit the static value [Bon92].

7.1.4 Use sensitivity

Work related to use sensitivity has been considered from different perspectives and languages. Program transformations and value representations have been

proposed to achieve some forms of use sensitivity. As well, analyses have been developed to solve similar data flow problems.

Static and Dynamic Representations for the Same Value

There are a number of existing analyses that can maintain both a static representation (concrete value for static uses) and a dynamic representation (textual representation for dynamic uses) for the same value. For example, the partial evaluators FUSE [WCRS91] and Schism [Con93b, Con93d] both carry around two representations of each closure, allowing it to be applied or residualized depending on the context. Danvy *et al.* show how program transformations prior to partial evaluation can achieve similar results [Dan95, DMP95].

These solutions suffice when each value has some dynamic representation, *i.e.*, an appropriate piece of text that can replace the value if it needs to be residualized. This is comparable to seeing all values as being liftable. However, for imperative languages like C, pointers, arrays, and structure values cannot be lifted. In certain cases, pointer values can have dynamic representations based on the name of a variable, (*e.g.,* using `&x` for the address of `x`), but this is not always possible. For example, these representations are not valid interprocedurally (if local variable names are passed out of scope) or with dynamic memory allocation (where there is no variable name). Further, in C, no such dynamic representation exists for structures. In these cases, use sensitivity is required to achieve accurate results.

On the other hand, use sensitivity can also be applied to liftable values, which in certain cases can produce better specialization. For example, Schism incorporates a form of use sensitivity with respect to data structures. Even though data structures in Schism can always be lifted, which means dynamic uses do not pollute static uses, lifting many copies of the same data structure introduces more code and memory usage in the residual program. When a large, static data structure occurs in many dynamic contexts, it is not desirable to lift it and thus residualize it in many different places. By detecting when this situation happens, Schism residualizes one unique instance of the data structure itself along with multiple references to it, thus avoiding data duplication.

It should also be mentioned that any other analysis that is based on a similar two-level semantics (i.e. evaluate and residualize) will also encounter similar problems. For example, much work has been done in the area of constant propagation, which involves a phase where constant values are propagated, followed by a phases where computations which depend solely on constant values are folded [MS93]. In these works, only liftable values are considered. If non-liftable values were propagated and used in folding expressions, a similar technique would need to be developed in order to resolve the problems discussed in this paper.

Structure Splitting

C-Mix is a partial evaluator for C that handles pointers, arrays, and data structures [And92, And94, JGS93a]. However, since its binding-time analysis is use-insensitive, dynamic uses of non-liftable values interfere with static uses. We have seen that, for certain programs, a use-insensitive binding-time analysis can incur a significant loss of precision.

As mentioned earlier, one way to circumvent the these losses is to rewrite the code by hand, carefully separating static uses from dynamic uses. C-Mix attempts to automate such a separation by *structure splitting*. This technique splits a data structure into separate components by creating a new variable for each structure element. This process can be repeated recursively (on nested structures) until all structures are eliminated. If the fields of the initial structure are liftable values, then all of the corresponding new variables are liftable. And, since values that are liftable do not incur a loss in flow-insensitive analyses, the problem is resolved.

Although this approach is currently intra-procedural, Andersen proposes an inter-procedural extension to structure splitting which would introduce a new function parameter for each field of a structure [And94]. This approach, however, does not appear to scale up to realistic applications. As already mentioned, systems programs typically maintain a system state which consists of numerous nested data structures. For example, in the `marshaling` application considered in Sect. 6, the system state is represented by `struct cu_data`, a data structure containing a total of 29 fields. To pass this information interprocedurally, the structure is always passed via a pointer, which avoids copying each field at each function call. Andersen's proposed inter-procedural extension defeats this technique, since each new parameter introduced reintroduces the copying.

Also, when dealing with large systems programs, it is typically the case that a small piece of the system is extracted and specialized. After specialization, the new, optimized piece must be reinserted into its larger context. For this reason, it is necessary to preserve the interface between these two parts. Andersen's proposed inter-procedural extension does not preserve this interface.

7.2 Related Analyses

Our two-phase binding-time analysis, consisting of a binding-time phase followed by a transformation phase, has similarities with other analyses. We compare our analyses with those used for program slicing, arity raising, and specializing functional representations of imperative programs.

7.2.1 Slicing

Program slicing computes the parts of a program that potentially affect values at a given program point [Tip94]. Program slicing can be seen as another form of

specialization, similar to the SFAC partial evaluator described in Section 1.3.1. The user provides a slicing criterion, from which a residual program is produced. Applications of slicing include software maintenance and debugging.

Forward slicing techniques, which propagate information from variable definitions to variables uses, have been used to define binding-time analyses for imperative programs [DRVH95]. This forward analysis is very similar to our binding-time phase. However, non-liftable values are not addressed by this work; no transformation phase is provided to treat uses in different contexts.

Additionally, there are backward slicing techniques which are similar to the transformation phase of our binding-time analysis, propagating information from variable uses to variable definitions [RT96]. This can be viewed as a form of *neededness* information. Just as slicing computes which commands are needed in a slice, our analysis computes the transformations for each construct. The main difference is that, instead of using a two-point domain (needed, not needed), our analysis is performed with a four-point domain (evaluate, residualize, evaluate and residualize, and {}) since certain constructs may to be both evaluated and residualized.

7.2.2 Arity Raising

Arity raising has been shown to be useful when specializing functional programs [Rom88, Rom90]. The motivation for this work is to eliminate unnecessary data constructors and accessors as well as to reduce function call overhead. For example, consider a cons cell that is passed to a function that uses only one of its components. Arity raising transforms the program by passing the two values to the function instead of the cons cell. This eliminates the initial construction and subsequent cell access. Further, instead of passing both values, only the value used by the function needs to be passed.

Arity raising, like our two stage binding-time analysis, is achieved by combining a forward analysis with a backward analysis. In both cases, the forward phase determines the feasibility of a certain transformation. Our binding-time analysis determines if a construct can be evaluated at specialization time while arity raising determines if a data structure being passed interprocedurally can indeed be split into its subcomponents. As well, both of the backward phases perform a form of neededness analysis. The binding-time analysis collects information concerning a variable's uses in order to determine the corresponding annotation with which to annotate the variable's definition. Arity raising determines which subcomponents are used by a function and therefore need to be passed. Notice how both require a backwards analysis to propagate information from variable uses to variable definitions.

7.2.3 Functional Representation of Imperative Programs

A different approach for obtaining effective specialization of imperative programs has been proposed [Mou97, MCL96]. Instead of directly treating an imperative program, the original source program is transformed into a functional representation. An existing partial evaluator for a functional language is then used to specialize the program, after which the residual program is transformed back into the original imperative language. The main advantage of this approach is that reusing an existing, mature partial evaluator avoids the need to design and implement a new partial evaluator. Initial results show that this approach may achieve a high degree of specialization; flow, context, return, and use sensitivity have been demonstrated for small examples. More experimentation would be needed to determine if this approach could be scaled up to handle the size and complexity of existing, realistic programs.

7.3 Static Analyses for Program Adaptation

Program adaptation, the ability for a program to adapt to the context in which it is used, has been proposed as a promising application of program transformation [Con96]. Program transformations such as partial evaluation are a particularly well-suited to adapt a general program to its environment. It seems necessary to use some type of static analyses, such as a binding-time analysis, for adaptation to be feasible and efficient. The important aspect of binding-time analysis is that it identifies the dependencies between different parts of a program. This information can then be used to prepare transformations that are eventually triggered once the actual values in the program's environment change. Two concrete applications of adaptation are run-time code generation and adaptive operating systems.

7.3.1 Run-time Code Generation

Run-time code generation has recently received a lot of attention. One key reason for this renewed interest is because recent advances in programming language research now offer more portable and less error prone techniques to generate code at run time.

Some approaches, however, still require the programmer to manually specify specializations [EHK96, EP94]. The need for user intervention is time consuming, somewhat complicated, and potentially error-prone. Additionally, without any analysis to produce global program information, the resulting specialized program may be less efficient.

Other approaches have attempted to automate the specialization process by providing static analyses to determine at compile time the transformations that

will be performed at run time. For example, Fabius [LL94, LL96] and the Dynamic Compilation system [APC+96] include some type of binding-time analysis in their approaches.

Fabius treats a simple language: a first-order subset of Standard ML. Therefore, it is likely that its binding-time analysis would not need the advanced features presented in this dissertation, such as flow, context, return, and use sensitivity. On the other hand, treating a simple language also prevents Fabius from treating existing or realistic applications. Dynamic Compilation, on the other hand, is aimed at realistic programs written in C. Interestingly enough, it is also targeted at treating operating systems components. Therefore, it is likely that its static analyses would need to contain the features presented in this dissertation to fully exploit specialization opportunities.

7.3.2 Adaptive operating systems

Current work is being done on adaptive operating systems [BSP+95, EKO95, MMO+94]. These approaches have developed new 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 the Synthetix group creates a synergistic effect where both groups benefit from the cross-fertilization [PAB+95]. The operating systems group identifies where specialization can be applied, and uses our tools to perform their adaptive specialization. By applying Tempo to systems programs, our group can continue refining the tools based on the feedback we receive.

Conclusion

Program specialization can automatically and effectively optimize existing, realistic applications. Previous studies have demonstrated that specializing large applications, such as systems programs, produces impressive speedups, but specialization was done by hand. We design new static analyses that automatically and accurately specialize these programs.

Our analyses contain specific features in order to fully exploit the specialization opportunities that exist in systems programs. Flow sensitivity allows a different analysis description to be computed for each update statement. Context sensitivity permits a function to be analyzed with respect to the specific analysis context of each call site. Return sensitivity enables a function which contains dynamic side-effects to return a static value. Use sensitivity allows different uses of a variable to have different analysis descriptions. We show that the lack of even one of these features drastically decreases the degree of specialization when dealing with systems programs.

Our approach includes a number of novel aspects. Unlike existing binding-time analyses, ours consists of two phases. The binding-time phase propagates information *forwards* from a variable definition to its uses, in order to determine whether constructs are static or dynamic. The transformation phase propagates information *backwards*, from variable uses to their definitions, to determine a transformation for each construct in the program. Additionally, we introduce new program transformations, such as evaluating *and* residualizing a construct.

Our analyses are integrated into Tempo, our partial evaluator for C, and serve as the basis for both compile-time and run-time specialization. Tempo has been applied to several systems applications running on different hardware platforms. For example, specializing the Sun Remote Procedure Call yielded speedups of up to 3.75. Tempo has also proven effective at specializing other types of programs, including numerical algorithms and image processing routines. Specializing these programs achieved speedups of up to 12.

Tempo's template-based approach to run-time specialization has been shown to be highly effective. Static analyses are used to generate templates and a target-generating extension at compile time, allowing efficient and effective specialization at run time. Our results show that run-time specialized programs run nearly as fast as the same program specialized and compiled at compile time.

Specialization requires as few as 3 runs to be amortized. Additionally, this approach is automatic, portable, and treats a realistic language, which increases the number of potential applications to which it can be applied.

Future Work: Tempo

Our results are so far encouraging. Let us now review the evolution of Tempo's analyses and how they might be extended in the future.

At the beginning of our thesis work, we knew that we would need some type of binding time analysis, but we did not know what kind of analysis. By studying the types of programs we wanted to specialize, we determined the features we believed were needed, such as flow and context sensitivity. Likewise, we identified the features we thought were not necessary, such as treating recursive functions.

Once preliminary versions of our analyses were designed, they were implemented and tested on application programs in order to assess their effectiveness. Although we had an idea of what to expect, the large size and complexity of the application programs made it impossible to precisely predict the results. Initial results revealed, for example, that our original design decisions were not sufficient in order to fully exploit important specialization opportunities. Therefore new features, such as use sensitivity and the dual evaluate/residualize transformation, were added to the improve the accuracy of the analyses. The new features were integrated into the implementation and the new system was once again applied to the application programs.

Having a running prototype that allowed us to specialize programs helped us to identify new specialization opportunities. For example, error status values were not initially considered very important. After specialization successfully eliminated most other constructs from a set of function calls, however, we realized that in certain cases these residualized return values became the bottleneck. This observation, for example, led to the addition of return sensitivity to our analyses.

As we continue to treat more programs, we are finding some of the features we decided to not include are, in fact, desirable. For example, we initially opted not to treat recursive functions, because systems programs do not often use them and it complicates the analyses and transformations. But since we started our work, we have in fact found a number of interesting applications, such as an operating systems packet filter or a Java byte-code interpreter, which do this type of recursion. Therefore, we have recently added this feature.

Another decision we initially decided to make is an approximation for data structures. We treat *partially static* data structures, allowing different fields of a data structure to have different analysis descriptions, but we only handle data structures *monovariantly*. In other words, there is one description for each data structure type, which applies to all data structures of that type. Initially, this approximation was sufficient, again, because systems programs typically use data structures in this uniform manner. However, subsequent applications which we

are considering, such as object-oriented languages, require that data structures are handled *polyvariantly*, allowing different data structures to have different descriptions. We are currently working on adding this feature to our analyses sometime soon.

As seen by these examples, the static analyses presented in this dissertation were not predetermined in advance. Rather, they evolved over time. Certain notions, such as use sensitivity and return sensitivity, did not even exist when the analyses were first designed. Other potential features may some day be included, as well, if the need arises. These possibilities include duplicating continuations following dynamic conditionals, or memoizing specializations on a per function basis, as opposed to memoizing at each program point.

An advantage of developing a partial evaluator in collaboration with researchers in another domain is that both parties receive opportunities that they would not have had otherwise. As programming language researchers, we are furnished with a wealth of operating system expertise which gives us a target for which we can continue to refine and improve our work. In return, we provide a tool which allows operating systems to be optimized in ways not previously possible. The exchange between the two parties creates continues to create more opportunities for both sides: the tool get improved, more applications can be optimized, which in turn identifies new opportunities to further improve the tool.

Since our system is continually changing, however, it is not a mature, stable system. Working with a changing prototype has drawbacks. For example, it would be desirable to prove some aspects of correctness about the analyses. Namely, we would want to prove that any construct annotated as static only depends on static input values. Further, we would like to prove that all constructs that are evaluated are static. There may also be additional properties we would like to prove, such as the correctness of the dual evaluate/residualize transformation we introduce. Such proofs would help to understand the analyses, since doing so would force all aspects of the analyses to be more rigorously expressed. Additionally, these proofs would guarantee that the fundamental properties of the analyses are correct.

As an evolving prototype, proving correctness would have to be done each time the analyses are modified. We expected to tackle these proofs once the features are relatively stable, but we are always finding new ways in which to improve the system. One possibility would be try to select a small subset of the language which seems to be stable, and provide proofs for this subset. Such proofs would be especially important if the system was more widely distributed and used by many other researchers.

Future Work: Applications

Tempo is currently being applied to other operating system programs, such as signal delivery and memory allocation. Results of these experiments will give

further insight as to the applicability of our approach. Further, we are exploring new areas, such as object oriented programs, as this area also seems to have many specialization opportunities similar to operating system programs. Below we outline the current state of these studies.

Transient "connections" between modules in a systems provide opportunities for specialization. In Chapter 1 we have explained how information communicated between modules of a system may contain many static values, giving the example of a file system. Another example of communication is when a group of processes repeatedly communicate to achieve some common goal. Experiments are currently being done where a connection is shared between two processes. One process repeatedly sends UNIX signals to the other process. The target and destination processes are their relevant properties are considered static. Specializing this case with Tempo produces code which sends the signal three times faster. This speedup was not completely achieved automatically, but rather with some human assistance. Further studies will show if this intervention can be automated and integrated into Tempo.

Dynamic memory allocation another function which is parameterized and generic so that the same function can be used in a wide variety of different situations. As it turns out, in systems programs it is often used in a regular fashion. Tempo is also being applied to specialize the allocator function in order to exploit the invariants available in the common cases.

Harissa is an efficient environment for the execution of Java programs. Harissa's compiler translates Java bytecode to C. Once translated into C, the Tempo could used to partially evaluate the program. In this case, Tempo's compile-time specializer would generate residual programs written in C code. This could be compiled and executed, or perhaps translated back into Java or Java byte code. Tempo's run-time specializer directly produces executable object code.

Instead of combining Harissa and Tempo, a partial evaluator could be developed to directly treat Java or Java byte code. In this case, it is possible that the principles and techniques used for Tempo might be useful. For example, the analyses in an offline approach may include flow, context, return, or use sensitivity.

Declarative specialization is an approach to program specialization in the context of the object-oriented paradigm. The goal of this approach is to help a programmer control specific aspects of specialization. For example, the programmer can express whether the specialization done manually or automatically, or at compile time or run time. Other directives are used to detect when the usage context changes, which means a specialized version can no longer be used, and to indicate specialized versions should be kept and how long.

Bibliography

[AK82] S.M. Abramov and N.V. Kondratjev. A compiler based on partial evaluation. In *Problems of Applied Mathematics and Software Systems*, pages 66–69. Moscow State University, Moscow, USSR, 1982. (In Russian).

[And92] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, San Francisco, CA, USA, June 1992. Yale University, New Haven, CT, USA. Technical Report YALEU/DCS/RR-909.

[And94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[APC+96] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI96 [PLD96], pages 149–159.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AWS91] W.Y. Au, D. Weise, and S. Seligman. Generating compiled simulations using partial evaluation. In *28th Design Automation Conference*, pages 205–210. New York: IEEE, June 1991.

[Bar78] J. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[BBH+94] J. Bell, F. Bellegarde, J. Hook, R. B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D. P. Oliva, T. Sheard, L. Tong, L. Walton, and T. Zhou. Software design for reliability and reuse: A proof-of-concept demonstration. In *Proceeding of TRI-Ada*, pages 396–404, 1994.

[BF93] S. Blazy and P. Facon. Partial evaluation for the understanding of fortran programs. In *Software Engineering and Knowledge Engineering, San Francisco, California, June 1993*, pages 517–525, 1993.

[BF96] S. Blazy and P. Facon. An automatic interprocedural analysis for the understanding of scientific application programs. volume 1110 of *Lecture Notes in Computer Science*, pages 1–16. Berlin: Springer-Verlag, 1996.

[BGZ94] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In PEPM94 [PEP94], pages 119–132.

[BM90] A. Bondorf and T. Mogensen. Logimix: A self-applicable partial evaluator for Prolog. DIKU, University of Copenhagen, Denmark, May 1990.

[Bon92] A. Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference in Lisp and Functional Programming, San Francisco, California (Lisp Pointers, vol. V, no. 1, 1992)*, pages 1–10. New York: ACM, 1992.

[BS94] A.A. Berlin and R.J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In PEPM94 [PEP94], pages 133–141.

[BSP+95] B.N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP95 [SOS95], pages 267–283.

[BVT+94] D. Batory, S. Vivek, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, September 1994.

[BW90] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[BW93a] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, 1993. DIKU Research Report 93/22.

[BW93b] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, Computer Science Department, University of Copenhagen, 1993. Research Report 93/22.

[CD90] 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.

[CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

[CJR+91] R. Cytron, Ferrante J., B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems*, pages 451–490, 1991.

[CK91] C. Consel and S.C. Khoo. Semantics-directed generation of a Prolog compiler. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP '91, Passau, Germany, August 1991 (Lecture Notes in Computer Science, vol. 528)*, pages 135–146. Berlin: Springer-Verlag, 1991.

[CN96] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL96 [POP96], pages 145–156.

[Con93a] C. Consel. Polyvariant binding-time analysis for applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 66–77. New York: ACM, 1993.

[Con93b] C. Consel. Polyvariant binding-time analysis for applicative languages. In PEPM93 [PEP93], pages 145–154.

[Con93c] C. Consel. Polyvariant binding-time analysis for applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 66–77. New York: ACM, 1993.

[Con93d] C. Consel. A tour of Schism. In PEPM93 [PEP93], pages 66–77.

[Con96] Charles Consel. Program adaptation based on program transformation. *In ACM Workshop on Strategic Directions in Computing Research. ACM Computing Surveys*, 28A(4), December 1996.

[CPW93] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In PEPM93 [PEP93], pages 44–46. Invited paper.

[CPW94] C. Consel, C. Pu, and J. Walpole. Making production OS kernel adaptive: Incremental specialization in practice. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 1994.

[CWKZ90] 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, White Plains, NY, USA, June 1990. ACM SIGPLAN Notices, 25(6).

[Dan95] O. Danvy. Type-directed partial evaluation. Technical Report PB-494, Computer Science Department, Aarhus University, July 1995.

[DMP95] O. Danvy, K. Malmkjær, and J. Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–228, September 1995.

[DRVH95] M. Das, T. Reps, and P. Van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 100–110, La Jolla, CA, USA, 1995. ACM Press.

[EGH94] 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 SIGPLAN Notices, 29(6), June 1994.

[EH94] 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.

[EHK96] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL96 [POP96], pages 131–144.

[EKO95] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [SOS95], pages 251–266.

[EP94] 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), pages 263–273. ACM Press, November 1994.

[Ers77] A.P. Ershov. On the essence of translation. *Computer Software and System Programming*, 3(5):332–346, 1977.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. Amsterdam: North-Holland, 1988.

[Fut88] Y. Futamura. Program evaluation and generalized partial computation. In *International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 1–8, 1988.

[GJ89] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, Amsterdam: North-Holland, 1989.

[GJ95] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Languages, Implementations, Logics, and Programs (PLILP'95). Lecture Notes in Computer Science, vol. ???* Berlin: Springer-Verlag, 1995. To appear.

[GNZ95] R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Berlin: Springer-Verlag, 1991.

[HM94] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 287–301. Berlin: Springer-Verlag, 1994.

[HN97] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In PEPM97 [PEP97], pages 63–73.

[HNC96] L. Hornof, J. Noyé, and C. Consel. Accurate partial evaluation of realistic programs via use sensitivity. Research Report 1064, IRISA, Rennes, France, June 1996.

[JGS93a] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, June 1993.

[JGS93b] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[Jør92] J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992. Student Project 92-1-4.

[JS80] N.D. Jones and D.A. Schmidt. Compiler generation from denotational semantics. In N.D. Jones, editor, *Semantics-Directed Compiler Generation, Aarhus, Denmark (Lecture Notes in Computer Science, vol. 94)*, pages 70–93. Berlin: Springer-Verlag, 1980.

[JSS89] 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.

[KC91] D. Kincaid and W. Cheney. *Numerical Analysis: Mathematics of Scientific Computing.* Brooks/Cole, 1991.

[KEH93] D. Keppel, S. Eggers, and R. Henry. Evaluating runtime compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[KKZG94] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. In U. Meyer and G. Snelting, editors, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.

[LL94] M. Leone and P. Lee. Lightweight run-time code generation. In PEPM94 [PEP94].

[LL96] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI96 [PLD96], pages 137–148.

[Loc87] B.N. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, September 1987. EUUG.

[Mas92] H. Massalin. *Synthesis: An Effictient Implementation of Fundamental Operating System Services*. PhD thesis, Department of Computer Science, Columbia University, 1992.

[MCL96] B. Moura, C. Consel, and J. Lawall. Bridging the gap between functional and imperative languages. Rapport de recherche, INRIA, Rennes, France, June 1996.

[Mey91] U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105, New Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).

[Mic89] Sun Microsystem. NFS: Network file system protocol specification. RFC 1094, Sun Microsystem, March 1989. ftp://ds.internic.net/rfc/1094.txt.

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

[MMV+97] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. Publication interne PI-1094, IRISA, Rennes, France, March 1997.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. Amsterdam: North-Holland, 1988.

[Mou97] B. Moura. *Bridging the Gap between Functional and Imperative Languages*. PhD thesis, University of Rennes I, April 1997.

[MP89] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, December 1990.

[MRB95] T.J. Marlowe, B.G. Ryder, and M. Burke. Defining flow sensitivity for data flow problems. Technical Report LCSR-TR-249, Computer Science Department, Rutgers University, July 1995.

[MS92] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992. Available from `ftp.diku.dk` as file `pub/diku/semantics/papers/D-152.ps.Z`.

[MS93] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letter on Programming Languages and Systems*, 2(1–4):213–232, March–December 1993.

[MVM97] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In PEPM97 [PEP97], pages 116–125.

[NHCL96] 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, November 1996.

[NN91] H.R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing. John Wiley & Sons, 1991.

[NP92] V. Nirkhe and W. Pugh. Partial evaluation and high-level imperative programming languages with applications in hard real-time systems. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 269–280, Albuquerque, New Mexico, USA, January 1992. ACM Press.

[PAB+95] 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 [SOS95], pages 314–324.

[PEP93] *Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM Press.

[PEP94] *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.

[PEP97] *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[PLD95] *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.* ACM SIGPLAN Notices, 30(6), June 1995.

[PLD96] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.* ACM SIGPLAN Notices, 31(5), May 1996.

[PLR85] R. Pike, B. N. Locanthi, and J.F. Reiser. Hardware/software tradeoffs for bitmap graphics on the blit. *Software - Practice and Experience*, 15(2):131–151, 1985.

[PMI88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[POP96] *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[PTVF93] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN The Art of Scientific Computing.* Cambridge University Press, Cambridge, 2nd edition, 1993.

[RAA+92] V. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX - Workshop Proceedings - Micro-kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, USA, April 1992.

[RG92] B. Rytz and M. Gengler. A polyvariant binding time analysis. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 21–28. New Haven, CT: Yale University, 1992.

[Rom88] S.A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. Amsterdam: North-Holland, 1988.

[Rom90] S.A. Romanenko. Arity raiser and its use in program specialization. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990 (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Berlin: Springer-Verlag, 1990.

[RT96] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 409–429, February 1996.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In PLDI95 [PLD95], pages 13–22.

[SOS95] *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[Sun90] Sun Microsystems. *Network Programming Guide*, March 1990.

[Sur95] Rajeev Surati. Practical partial evaluation. Master's thesis, Cambridge, MA: MIT Press, 1995.

[Tan87] A.S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.

[TC96] S. Thibault and C. Consel. A framework of application generator design. Rapport de recherche RR-3005, INRIA, Rennes, France, December 1996. To appear in ACM SIGSOFT Symposium on Software Reusability (SSR'97).

[Tip94] F. Tip. A survey of program slicing techniques. Report CS-R9438, Computer Science, Centrum voor Wiskunde en Informatica, 1994.

[VMC96a] E.N. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: the RPC case study. In *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, pages 24–28, Tucson, AZ, USA, February 1996.

[VMC+96b] E.N. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noyé, and C. Pu. A uniform automatic approach to copy elimination in system extensions via program specialization. Research Report 2903, INRIA, Rennes, France, June 1996.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191, Cambridge, MA, USA, August 1991. Springer-Verlag.

[WFW+94] 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.

[WL95] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis of C programs. In PLDI95 [PLD95], pages 1–12.

Summary

This dissertation shows how program specialization can automatically and effectively optimize existing, realistic applications. We consider a class of existing applications, namely systems programs. Previous studies have demonstrated that specializing these programs produces impressive speedups, but specialization was done by hand. We present new static analyses that automatically and accurately treat the common program patterns found in these programs. Specific features include:

Flow sensitivity. A different analysis description is computed for each program point.

Context sensitivity. A function is analyzed with respect to the specific analysis context of each call site.

Return sensitivity. A different analysis description is computed for the side-effects and the return value of a function.

Use sensitivity. Different uses of a variable may have different analysis descriptions at the same program point.

Two-phase binding-time analysis. A binding-time phase followed by a transformation phase.

New program transformations. Multiple transformations may be associated with a single program construct.

Our analyses support specialization at both compile time and run time. We integrate our analyses into a partial evaluator and apply it to a variety of existing applications, including a commercial operating system component. We obtain significant speedups for both compile-time and run-time specialization.