The Dissertation Committee for Samuel Zev Guyer
certifies that this is the approved version of the following dissertation:

Incorporating Domain-Specific Information into the Compilation Process

Committee:

Calvin Lin, Supervisor

Robert A. van de Geijn

James C. Browne

Don Batory

Kathryn S. McKinley

Craig M. Chase

Incorporating Domain-Specific Information into the Compilation Process

by

Samuel Zev Guyer, M.S., B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2003

In memory of my uncle Tim.

Acknowledgments

I am indebted to many people for helping me become a better scientist and for making these past years enjoyable and rewarding. First, I would like to thank my advisor, Calvin Lin, for consistently supporting me and for investing both the time and effort needed for this research. In addition to providing valuable guidance and insight, he gave me the freedom to develop my own ideas and to pursue my own goals. Seven years ago we embarked on an ambitious plan, and Calvin remained committed to it despite my outrageous insistence on building an entire compiler from scratch.

I would like to thank my committee members, each of whom contributed to my research in their own way: J.C. Browne, for convincing me that it's okay to spend two years building infrastructure; Don Batory, who treated me like his own student, even though I was just hanging around distracting his real students; Craig Chase, who has given me a new appreciation for the work being done in the ECE department; Kathryn McKinley, who can deliver a gold nugget of advice in a 30-second meeting; and last, but not least, the incomparable Robert van de Geijn, who showed me that computer science research can be a full-contact sport.

I would also like to thank several of my fellow graduate students, who served both as friends and as colleagues. Two students in particular, Emery Berger and Yannis Smaragdakis, helped me to understand what it means to be a computer scientist, how to think deeply about problems, and not to apologize for being an intellectual. I am also proud to have been a part of the institution known as "the lunch bunch": in its first incarnation with

Yannis and Jeff Thomas, and later with Emery, Daniel Jimenez, Rich Cardone, Ram Mettu, Brendon Cahoon, Xianglong Huang, and the inimitable Phoebe Weidmann—it's not lunch until Phoebe is in vapor-lock.

I owe more to my family than I can possibly express. I suspect that my parents knew I would become a scientist even before I did. They always supported and nurtured my interests, even if it meant driving several hours in order to dig for rocks in an abandoned quarry. I continue to be amazed and inspired by their achievements and by the achievements of my siblings.

Finally, my wife Andrea supported me in so many ways throughout my graduate career and endured many of the trials and tribulations along with me. She is much more than my spouse—she is my confidant, my partner, and my best friend. And she makes me laugh.

SAMUEL ZEV GUYER

The University of Texas at Austin
May 2003

Incorporating Domain-Specific Information into the Compilation Process

Publication No. ____________________

Samuel Zev Guyer, Ph.D.
The University of Texas at Austin, 2003

Supervisor: Calvin Lin

Despite many advances in compiler research, traditional compilers continue to suffer from one significant limitation: they only recognize the low-level primitive constructs of their languages. In contrast, programmers increasingly benefit from higher level software components, which implement a variety of specialized domains—everything from basic file access to 3D graphics and parallel programming. The result is a marked difference between the level of abstraction in software development and the level of abstraction in compilation.

In this thesis we present the Broadway compiler, which closes this gap. Broadway represents a new kind of compiler, called a *library-level compiler*, that supports domain-specific compilation by extending the benefits of compiler support to software libraries. The key to our approach is a separate annotation language that conveys domain-specific information about libraries to our compiler, allowing it to treat library routines more like built-in language operations. Using this information, the compiler can perform *library-level* program analysis and apply *library-level* optimizations.

We explore both the opportunities and challenges presented by library-level compilation. We show that library-level optimizations can increase the performance of several parallel programs written using a highly-tuned parallel linear algebra library. These high-level optimizations are beyond the capabilities of a traditional compiler and even rival the performance of programs hand-coded by an expert. We also show that our compiler is an effective tool for detecting a range of library-level errors, including several significant security vulnerabilities. Finally, we present a new *client-driven* pointer analysis algorithm, which provides precise and scalable program analysis to meet the demanding requirements of library-level compilation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last few decades, compiler research has produced a body of techniques that allow compilers to serve as much more than mere translators. As these techniques have grown more sophisticated, the role of compilers has expanded to include improving program performance, checking programs for errors, and supporting software engineering. We now rely on compilers as indispensable tools in the development of quality software. Despite many advances, however, one significant limitation prevents traditional compilers from reaching their full potential: they only recognize the low-level primitive constructs of their languages. In contrast, programmers increasingly benefit from higher level software components, which are encapsulated in reusable modules—everything from basic file access to 3D graphics and parallel programming. The result is a marked difference between the level of abstraction in software development and the level of abstraction in compilation. In this thesis, we present a system aimed at closing this gap. We leverage the power of existing compiler techniques, and we provide a way to incorporate domain-specific information about non-primitive constructs into the compilation process. By combining high-level information with aggressive algorithms, we push compilers well beyond their current capabilities.

1.1 Problem: Libraries are languages without compilers

In this work we develop a technique for providing high-level compilation within current programming practices. This choice reflects our goals of improving the quality of existing software and of minimizing our impact on the software development process. We satisfy these goals by focusing on programs written in a traditional general-purpose programming language: ANSI C. Our ideas apply equally well to programs written in Java or C++ and might even benefit from the increased modularity in object-oriented languages. In these languages, high-level programming abstractions are most often represented by software libraries.

A library typically implements some well-understood domain of computation, and the library interface defines the types and operations of this domain. Programmers access this functionality from within C using the function call mechanism, which allows high-level library operations to coexist with built-in language constructs. In this sense, libraries serve as small, domain-specific language extensions that are embedded in a traditional programming language. When viewed as operations of a language, library routines share many features with the built-in operations, such as rules of usage and performance tradeoffs. Unlike their built-in counterparts, however, library routines do not receive any compiler support.

The two code fragments in Figure 1.1 illustrate the inferior treatment afforded to library routines. The code on the left shows an `if` statement whose true branch is missing its close curly brace (shown in comments). The C compiler reports an error here, and the programmer can immediately fix the problem. The code on the right shows some 3D graphics code written using the OpenGL library. The call to `glBegin()`, which initiates the definition of a shape, is missing the corresponding call to `glEnd()`. In this case, the C compiler reports no error because it is unaware of any rules governing the use of these functions. The bug will be discovered at run-time and could require considerable testing and debugging to diagnose.

We find the same inequity in optimization, where primitive operations enjoy a vari-

```c
if (cond) {
  ...
/* Missing: } */
```

```c
glBegin(GL_POLYGON);
  ...
/* Missing: glEnd(GL_POLYGON); */
```

Figure 1.1: The C compiler checks primitive delimiters, but not library function delimiters.

ety of performance-enhancing transformations, while library calls are primarily a hindrance to optimization. Consider, for example, the difference between the built-in math operators and the extended math operators provided by the standard math library. Figure 1.2 shows an example `for` loop, on the left, along with two optimized versions of the loop. The middle version includes two optimizations we can expect from a conventional C compiler: the computation `d1 = 1.0/z;` is loop invariant and the integer division `j = i/4;` can be converted to a more efficient bit shift. Notice, however, that the computation `d2 = cos(z);` is also loop invariant, and that the expression `pow(y, 3)` can be replaced by a more efficient multiplication, `y * y * y`. Unfortunately, few, if any, conventional compilers can take advantage of these optimization opportunities.

```c
for (i=1; i<N; i++) {
  d1 = 1.0/z;
  d2 = cos(z);
  j = i/4;
  d3 = pow(y, 3);
}
```

```c
d1 = 1.0/z;
for (i=1; i<N; i++) {
  d2 = cos(z);
  j = i >> 2;
  d3 = pow(y, 3);
}
```

```c
d1 = 1.0/z;
d2 = cos(z);
for (i=1; i<N; i++) {
  j = i >> 2;
  d3 = y * y * y;
}
```

Figure 1.2: Compilers can optimize built-in operators, but miss opportunities involving library calls.

Furthermore, these missed opportunities are noticeably different from traditional optimizations: the math library optimizations described above have a far greater impact on performance than their conventional counterparts. With these additional optimizations, the loop on the right performs significantly better than either of the other two. The reason is that the conventional optimizations improve the loop by just a few instructions, while the math library optimizations improve it by hundreds or thousands of instructions.

1.2 Solution: A library-level compiler

In this thesis, we present a new kind of compiler, called a *library-level compiler*, that extends the benefits of compiler support to software libraries. The goal of library-level compilation is to treat library routines more like built-in language operations by making the compiler aware of their domain-specific semantics. Using this information, the compiler can perform *library-level* program analysis and apply *library-level* optimizations. In many cases, domain-specific compilation is also *high-level* compilation because library routines often implement more complex or more abstract operations than their built-in counterparts. This is not always the case, however, because libraries can also represent low-level domains, such as device drivers.

The examples presented earlier provide a glimpse of the potential benefits of library-level compilation. By recognizing and exploiting the domain-specific semantics of libraries, we raise the level of abstraction of compilation. But the examples also hint at some of the challenges. First, unlike the built-in language features, there is no fixed set of libraries to support. Therefore, a library-level compiler needs to accommodate the different needs of different libraries. Second, library routines are often more complex than their built-in counterparts. For example, in Figure 1.1 the call to `glEnd()` only matches the call to `glBegin()` when the input argument is the same. Other typical complications include multiple input and output arguments, implicit arguments and internal side-effects, access to I/O devices, and the use of pointers and pointer-based data structures. Finally, library routines are not bound by any explicit grammatical rules, so related library calls are often spread throughout the source code. For example, the programmer is not obligated to put the call to `glEnd()` in the same function as the call to `glBegin()`. This lack of scoping constraints suggests that library-level compilation is a whole-program activity.

1.3 The Broadway Compiler

The Broadway compiler is our implementation of a library-level compiler. Its design is guided by the observation that many compiler algorithms work just as well on library operators as they do on built-in operators. The compiler does not need entirely new mechanisms for library-level compilation; it primarily needs information about the library routines so that it can integrate them into the existing mechanisms. Therefore, our compiler system consists of two main components: an annotation language for capturing information about library routines, and a configurable compiler that reads the annotations and uses them to perform library-level program analysis and optimization.

The annotation language provides a simple, declarative syntax for describing properties of the library interface. Each library has its own set of annotations, which accompany the usual header files and library implementation. Besides basic interface information, the real value of the annotations is the library-specific information they contain: expert knowledge on how to use the library safely and effectively, which is so often lost or relegated to a user manual. However, the library expert who provides this information is unlikely to also be a compiler expert. The challenge in designing the language is to balance these two opposing goals: the desire to have a sophisticated language that supports powerful compilation tasks, and the need to keep the language accessible to non-compiler experts. Our approach relieves some of the tension between these goals by leveraging existing compiler mechanisms. The annotation language only conveys information that configures the mechanisms, which allows us to hide many of the compiler implementation details.

The Broadway compiler is a C source-to-source translator that accepts as input unmodified C source code along with annotation files for the libraries used in the code. It implements many familiar compiler mechanisms, including program analysis and optimization passes, which we have modified to incorporate information from the annotations. The core of the compiler is a configurable, whole-program dataflow analysis framework, including a precise pointer analyzer. A novel feature of our analyzer is that we can control its precision at a very fine level, without requiring any changes in the annotations. We use this feature to explore how the precision of analysis impacts the accuracy and efficacy of library-level compilation.

1.4 Benefits

Library-level compilation, and our approach in particular, provides a number of benefits. First, library-specific information does more than just increase the number of errors detected or optimizations applied: these new capabilities are qualitatively different from conventional error detection and optimization. Library-level errors encompass a wider range of subtle and severe bugs, including information leaks and security vulnerabilities. Individual library-level optimizations often have a far greater impact on performance than conventional optimizations.

Second, the annotations capture information that is otherwise hard to obtain and painful to apply. Library expertise is traditionally supplied only in an informal form, such as documentation, or is discovered through trial and error. Therefore, checking for errors and tuning performance are typically difficult and error-prone manual tasks. As a result, the quality of software depends heavily on the level of the expertise attained by library users, and on their level of motivation in applying this knowledge. Even in the best circumstances, the manually applied optimizations often render the source code incomprehensible and unmaintainable.

Annotating the library goes a long way towards eliminating these problems. The annotations serve as a repository of expert information that programmers can easily communicate. The formal notation enables automation, which ensures that error checking and optimization are thorough, complete, and painless. As a result, all users of the library obtain the maximum benefit with minimal effort. And while developing the annotations can be difficult, this cost is borne by the library developer and is amortized over all the uses of the library.

Third, our approach promotes a more sensible division of the responsibility for software quality. The annotations are provided by a library expert, and their primary function is to capture and convey knowledge about the library. The library expert needs very little compiler knowledge because we provide it in the Broadway implementation. In addition, by confining the library expertise to a separate language, we avoid putting ad hoc features in the compiler. Together, the annotations and the compiler allow the programmer to concentrate on clean application design.

Finally, an interesting side-effect of library-level compilation is that software modularity becomes an asset to the compiler, rather than a liability. Previous research has shown that modularity can interfere with compilation and hurt program performance [1]. Module boundaries are often barriers to conventional compilation because they hide critical information. In our approach, the module boundary provides an opportunity not just to supply missing information, but to add new information that raises the level of abstraction. In many cases, this high-level information could not otherwise be derived, even in the absence of modularity.

1.5 Evaluation

The main thesis of this research is that we can improve the error-checking and optimization capabilities of compilers by adding library-specific information into the compilation process. In order to evaluate this claim, we identify a number of specific scientific questions, which follow directly from the goals and benefits described above.

- We evaluate the annotation language by annotating several different libraries with different error-checking and optimization requirements. Can the language express a variety of library-level compilation tasks, even across disparate libraries? Is the notation sufficiently powerful without overburdening the library expert? Are there error-checking or optimization tasks that we would like to express, but cannot?

- We evaluate library-level optimization by applying it to performance-sensitive applications and measuring the performance improvement. Does library-level optimization yield significant improvements over conventional compilation? Can we approach the performance of "hand-coded" applications? Can the annotations capture a range of optimizations, both within a single library and across libraries? What properties of the library make it more amenable to library-level optimization?

- We evaluate library-level error detection by looking for a number of high-level errors and security holes in programs that use the Standard C Library. What kinds of error detection problems can we express using the annotation language? How effective is the compiler at detecting these errors? What is the false positive rate?

- We also focus on evaluating the program analysis framework itself. How does the precision of the framework affect its accuracy? How does the precision affect its cost, in time and space? Are different analysis problems inherently easier to solve than others?

1.6 Contributions

The primary contribution of this thesis is a systematic approach to encoding domain-specific expertise in a form that a compiler can exploit. In current programming practice, domain expertise is represented only in human-readable forms, such as user manuals, if it is even written down at all. As a result, acquiring this knowledge and applying it in software development are tedious and error-prone manual tasks, and they place a considerable burden on the programmer. Our solution codifies domain knowledge so that the compiler can apply it automatically, which relieves the programmer of this burden and produces consistent and reliable results.

This thesis makes the following specific contributions:

- We develop an annotation language for expressing information about library interfaces and conveying it to the compiler [43]. This language is the key to our approach because it serves as a repository for library-specific expertise. The language design focuses on striking a balance between power of expression and ease of use.

- We implement a fully functional library-level compiler called the Broadway compiler, which reads annotation files and performs library-level error detection and optimization on C programs. The design of our compiler represents foundational research on the opportunities and challenges of this new compilation technique.

- We show that library-level optimizations yield performance improvements that cannot be obtained by conventional optimization alone. In some cases, our approach can equal the performance of hand-coded applications [44].

- We detect a significant class of high-level programming errors and security holes in real C programs [42]. For one such security hole, the format string vulnerability, our compiler finds all the known errors with no false positives.

- We introduce a new *client-driven* pointer analysis algorithm [45] that addresses the demanding analysis requirements of library-level compilation. Our algorithm provides precise interprocedural pointer analysis information at a reasonable cost by automatically adapting its precision to satisfy the specific needs of each compilation task.

1.7 Outline

The rest of this dissertation is organized as follows. Chapter 2 presents a more detailed technical motivation, including the requirements and challenges of library-level compilation, and describes our system design. Chapter 3 describes the implementation of the compiler, which contains a number of novel features, and provides background on the compiler analysis and optimization algorithms. Chapter 4 presents the annotation language, including a complete description of its syntax and semantics. Chapter 5 presents our library-level optimization experiments and results. Chapter 6 presents our error detection experiments and results. Chapter 7 describes our client-driven pointer analysis algorithm and explores the effects of analysis precision on the cost and accuracy of high-level program analysis. Chapter 8 reviews previous work in all areas related to this research. Finally, Chapter 9 wraps up with a summary and conclusions.

Chapter 2

System Design

In this chapter we describe the overall design of the Broadway compiler. We start with a detailed technical motivation for library-level compilation, expanding on both its opportunities and its challenges. The most significant feature of our design is its focus on adding domain-specific information into existing compiler mechanisms, rather than trying to support arbitrary new compiler passes. The goal of this chapter, however, is to present the rationale for our design decisions. Many other designs are possible, and we show some of the tradeoffs and alternatives along the way.

2.1 Motivation

Domain-specific compilation has the potential to greatly improve the quality of software by recognizing and exploiting domain-specific program properties. However, there are many possible ways to take advantage of these opportunities. Our approach expresses domain-specific compilation problems in terms of existing compiler mechanisms, which have grown to include increasingly powerful program analysis and transformation capabilities. We build on compiler techniques that have proved themselves effective on traditional programming languages and we reformulate them to work on non-primitive programming constructs. In this section we focus on three fundamental tasks that traditional compilers perform: error checking, traditional optimization passes, and operator refinement and specialization. We use these tasks to guide the design of our compiler and we show how they extend in a natural way to domain-specific compilation

2.1.1 Error checking

Traditional compilers have always provided some form of error checking. Modern languages avoid many basic errors by enforcing syntactic structure, such as lexical scoping and matching delimiters. Most languages also include a type system, which the compiler uses to ensure operator and operand compatibility.

Domain-specific error checking can provide a substantial improvement over the ordinary compiler checks. First, it greatly expands the compiler's repertoire of errors, including many errors that are not recognized as errors in the semantics of a general-purpose programming language. Second, it allows the compiler to provide meaningful domain-specific error messages at compile time, rather than allowing programs to fail at run-time. Finally, domain-specific errors often represent more significant and subtle program defects, such as security vulnerabilities.

2.1.2 Traditional optimizations

Programmers have come to rely on compilers for performance improvement. Current state-of-the-art compilers often apply a formidable battery of optimizations, such as constant propagation and constant folding, strength reduction, dead code removal, and loop invariant code motion. Compilers rely on program analyses both to identify candidates for these optimizations and to apply the code transformations correctly.

We can extend these optimizations to other domains in a straightforward manner by providing each program analysis pass with the information it needs about the operations of the domain. In many cases, the only additional information that an analysis pass needs is the data dependence information for each operation: which variables the operation accesses and which ones it modifies. For example, using data dependence information, we can seamlessly integrate a non-primitive operation into liveness analysis. Regardless of the particular operation, if its results are not used (and it has no other side-effects) then the operation is dead and may be removed. Removing high-level operations can have a greater impact on performance because the operations are more complex and require many instructions to execute.

2.1.3 Refinement and specialization

Another important task in the compiler is decomposing complex operators into simpler ones. In doing so, the compiler exposes the lower level operators to further analysis and optimization. For example, the array index operator in C is typically implemented as an address computation followed by an indirect memory access. However, in the special case when the index is zero, the address computation is unnecessary. Even when the index is not a constant, if the address computation is loop invariant then the compiler can hoist it out of the loop. Without this process of refinement and optimization, the encapsulation of programming abstractions would be a barrier to better performance.

Domain-specific operations often encapsulate more complex high-level operations than their built-in counterparts do. As a result, domain-specific operations offer many more opportunities to improve performance through decomposition and specialization. First, there are often many different ways to perform a complex computation. For example, a matrix library may offer several different matrix multiply routines that are optimized for specific matrix shapes. We can use domain-specific program analysis to automatically select from among these specialized routines. Second, programs can benefit more from decomposing high-level operations because these operations contain more code and more layers of abstraction. Exposing the lower layers of a complex routine allows the compiler to optimize it for the specific call-site. Such optimizations might include eliminating redundant parameter checks or unrolling loops to match input parameters. We can also use domain-specific information to make better decisions about inlining and specialization.

2.2 Goals and challenges

Designing a compiler for domain-specific programming constructs presents a number of challenges not encountered in traditional compiler design. In this section we discuss our specific goals for the Broadway compiler and their implications for the system design.

2.2.1 Support existing systems

One of the most fundamental aspects of our design is the choice to work within traditional programming practices. The goal is for Broadway to provide domain-specific compiler support for existing systems without substantially changing the software development process. Programmers can continue to use familiar programming languages and standard programming resources, while obtaining the benefits of domain-specific compilation. In addition, this feature allows us to demonstrate the benefits of our compiler on real, unmodified application programs.

- ***Design decision*: Broadway is a source-to-source C compiler.**

Our system supports domain-specific compilation within the C programming language. We focus on C because it continues to be the language of choice for system software and many performance-critical applications. In addition, the growth of open source projects provides us with a large pool of example programs written in C. However, it is worth noting that the modular programming style promoted by object-oriented languages such as Java and C++ is likely to yield even more opportunities for domain-specific compilation.

Since domain-specific optimizations occur at a higher level than traditional optimizations, a source-to-source compiler serves our purposes better than a compiler with a traditional back-end. Furthermore, by producing C as output we keep the Broadway compiler platform-independent. This feature is particularly important for high-performance applications, where architectures vary considerably and change frequently. After high-level compilation, we pass the generated code to traditional compiler tools. This process allows us to take advantage of all the low-level optimizations and code generation provided by existing compilers.

In traditional imperative languages like C, domain-specific programming constructs are represented by software libraries. Therefore, the focus of our system is on recognizing and exploiting the high-level semantics of library interfaces. Our approach is to view library routines as domain-specific operators that coexist with and complement the built-in operators of the language. C serves as the *base programming language*, to which we add the domain-specific semantics of libraries. We refer to this approach as *library-level* compilation. Standardized libraries are particularly good targets for compilation because they are widely used, they have stable and well-documented interfaces, and because they often represent a well-understood domain of computation.

- ***Design decision*: Exploit the domain-specific semantics represented by software libraries.**

An alternative approach, taken by a number of previous systems [78, 79], is to support domain-specific abstractions more directly by extending the base programming language or by developing a new programming language. While this is a legitimate long-term goal for programming languages, it would severely weaken our system and our results. Our experiments would be limited to applications that we were willing to redevelop using our new language, and we might be left wondering what role our coding played in the success of the system. Error checking results, in particular, are of dubious value when the authors of the error checking system are also the authors of the applications to be checked.

The benefits of supporting existing systems, however, are not without a cost. Unlike special-purpose languages, the domain-specific abstractions in software libraries are accessed exclusively through the mechanisms of the underlying base programming language. Therefore, in order to perform domain-specific compilation tasks in this setting, the compiler must be able to properly analyze and transform all of the low-level constructs provided by this programming language. In particular, library interfaces often have complex calling sequences and make heavy use of pointer-based data structures. For example, the Standard C Library provides file access through two different mechanisms, file streams and file descriptors, both of which behave like pointers and can have internal state shared between them. Precise analysis of these programming constructs is critical for accurate program checking and correct program transformation.

- ***Design decision*: Implement aggressive program analysis.**

The core of the Broadway compiler is a whole-program dataflow analysis framework. The framework includes a precise pointer analysis algorithm that allows us to accurately model the behavior of most library-created objects and data structures. The framework also builds a complete dependence graph (*use-def* chains) of the target program. While this level of precision can be quite costly to compute, we show that it is essential for many library-level compilation tasks. Chapter 3 describes the implementation of our analysis framework.

2.2.2 Many libraries, one compiler

One of the most appealing features of software libraries is that they are not fixed: programmers can obtain or produce new libraries at any time to serve particular programming needs. Therefore, unlike traditional compilers, we cannot hard-wire the Broadway compiler to handle a fixed set of primitive operations. We need the compiler to accommodate whatever extensions programmers use. Furthermore, the semantics of these constructs is often more varied and complex than their built-in counterparts—this characteristic makes them *high-level*. We need a way to model the behavior of programming domains as diverse as files, matrices, and graphics objects.

- *Design decision*: **Separate the general-purpose compiler mechanisms from the domain-specific library information.**

As the Broadway compiler implementers, we provide a number of powerful compiler algorithms that we have made configurable. A domain expert provides the extra information about the data types and the operations that make up the domain. This information is expressed using a lightweight specification language, which we refer to as the *annotation language*. While compiling an application program, the compiler reads the annotations for each library in order to apply library-specific analyses and transformations.

Alternatively, we could hope for advances in program understanding technology that allow the compiler to perform library-level compilation tasks without needing the annotations. In many cases, however, this expectation is not reasonable: we would be asking our compiler to automatically discover, just from the library implementation, the correctness and performance characteristics of the domain that the library represents. For example, we cannot expect a compiler to infer the properties of the cosine function from its implementation as a Taylor series expansion. Even more problematic is the fact that many system libraries ultimately invoke kernel system calls for which source may not be available, or even worse, they directly manipulate hardware resources. The design of the Broadway compiler specifically targets the opportunity to capture information that is difficult or impossible to extract from source code.

2.2.3 Domain experts are not compiler experts

One implication of the design decisions so far is that they place the burden of specifying library information on the domain expert – in many cases, the library writer. Since domain experts are rarely compiler experts, we need to make sure that they can use our system without having to learn how compilers work. This constraint creates a language design tradeoff between power of expression and ease of use.

- *Design decision*: **Design the annotation language for use by domain experts, not compiler experts.**

Our annotation language is clean and declarative, favoring simplicity and ease-of-use over complexity and power. For example, the language allows the annotator to define library-specific program analysis passes without having to understand the underlying lattice theory. We accomplish this goal by limiting these new passes to a particular class of analysis problems that is easy to understand and express but still useful for a variety of tasks. In many cases, we mitigate this tradeoff by including sophisticated compiler machinery in Broadway: the more compiler expertise we build into Broadway, the less information it needs from the annotations. For example, our formulation of one common optimization, dead-code elimination, uses a sophisticated and aggressive program analysis algorithm not found in traditional compilers. To take advantage of this analysis, however, the annotator just needs to indicate which arguments to each library routine are accessed and which are modified.

One alternative to our design is to develop a comprehensive domain-specific language for implementing compiler analyses [66] and optimizations [95]. The advantage of this approach over ours is that it supports a wider variety of compiler tasks. This expressive power, however, comes at the expense of usability: such languages require a considerable level of compiler expertise. Thus, while more comprehensive systems are useful tools for compiler implementers and researchers, they are not appropriate for our purposes.

Another alternative is to provide a programming interface, or "API", to the compiler internals, which programmers use to write their own compiler extensions [85, 36]. The advantage of this option is that it relies on existing programming languages, rather than introducing a new specification language, and it allows considerable flexibility. The downside of this approach, however, is that it directly exposes the annotator to the implementation of the compiler, including all of the associated programming problems, such as the need to learn calling sequences, design data structures, and debug the resulting code.

2.2.4 Information is the key

Almost all optimization passes consist of two parts: an analysis phase that identifies optimization opportunities, and a transformation phase that makes changes to the code. There is an asymmetry, however, between the two parts. The transformations are generally simple and straightforward: they typically involve either moving a code fragment, removing a code fragment, or replacing one fragment with another. On the other hand, program analysis comprises a wide range of complex algorithms for collecting information about how programs work and for determining optimization candidates. In fact, the innovations in program optimization are primarily innovations in program analysis: the better the analysis, the more effective the optimization. For example, the algorithm to identify redundant computations has progressed from a simple local analysis (within a basic block), to a global analysis (taking into account control flow) and finally to partial redundancy analysis (removing redundancy on specific execution paths). Nevertheless, the basic machinery for redundancy elimination remains essentially the same: the removal of redundant statements.

- ***Design decision*: Focus on employing domain-specific information to improve program analysis.**

The annotation language supports a basic set of code transformations for removing and replacing individual library routine calls in the application code. But its most significant feature is the ability to define new library-specific dataflow analysis passes. This allows the Broadway compiler to collect domain-specific information about how the application program uses—or misuses—the library interface. The annotations can then query these analysis results to determine how the application can use the library more effectively or more safely.

Without library-specific program analysis, library-specific error checking and optimization are not possible.

2.2.5 Scalability

Library-level compilation tasks are often far more taxing on compiler mechanisms than their traditional counterparts. These demands are particularly acute when compiling large and complex applications, such as system software and server applications. The reason is that important properties of libraries, such as the state of a library data structure, flow throughout the application program. For example, the lifetime of an open file or socket can span many procedures. At the same time, we need a high level of precision to discern individual library structures and accurately track their properties. These two requirements conspire to pose a significant challenge for the scalability of our system: we want to analyze large programs over a large scope, while still modeling the objects at a fine level of detail. We show in Chapter 7 that a naive approach yields an unusable algorithm, even for modest sized programs.

- ***Design decision*: Develop a new analysis algorithm that is both precise and scalable.**

In Chapter 7 we present our client-driven analysis algorithm, which controls the cost of precise analysis by adapting its precision to suit the needs of each library-level compilation task. We find that this algorithm can produce accurate analysis results at a fraction of the cost of comparable traditional algorithms. The key to the scalability of this algorithm is that it only applies extra effort where it is needed in each input program.

Recent research has produced a number of alternative strategies for scalable program analysis. A common approach is simply to choose a lower level of precision so that the analysis runs within reasonable time and memory constraints. We consider this option unacceptable because it excludes a number of useful library-level compilation tasks. Other approaches include memoizing partial analysis results, and using demand driven analysis. Both of these options are complementary to ours, and they could be used in conjunction with our client-driven algorithm.

2.3 System architecture

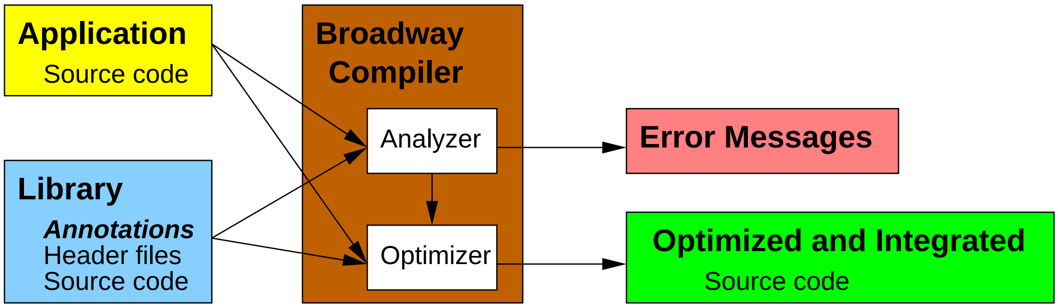

Figure 2.1: The Broadway compiler takes as input application source code, library source code, and library annotations and applies library-level compilation to produce error messages and optimized code.

Figure 7.2 shows the overall architecture of the Broadway compiler. The annotations capture domain-specific information about library routines, including library-specific program analyses and optimizations. The annotations are provided by a library expert and placed in a separate file that accompanies the usual library header files and source code. Broadway reads the annotations and makes the information available to different compiler components so that they can perform library-level error checking and optimization. The output from the compiler may include error messages generated by the error checking annotations, or it may consist of transformed code produced by optimization annotations. In the latter case, the resulting integrated system of library and application code is then compiled and linked using conventional tools.

2.4 Conclusions

This chapter has described our particular point in the design space of high-level, configurable compilers. This design is the result of a number of design decisions with numerous alternatives. In subsequent chapters we evaluate our approach on specific library-level compilation problems. We believe that the range of real, practical problems that can be solved using our system substantiates the tradeoffs that we make. In particular, it is interesting to note that we started the library-level error checking research after the system was fully implemented for library-level optimization. Aside from addressing scalability, the same system that we designed primarily to optimize high-performance scientific codes proves to be among the best available for detecting security vulnerabilities in system software.

Chapter 3

Compiler implementation

This chapter describes the architecture and implementation of our compiler and provides background on the underlying compiler algorithms. One of the key features of our design is the ability to define library-specific program analysis passes. Therefore, the core of our compiler is a program analysis framework that uses iterative dataflow analysis to solve both traditional analysis problems and library-specific analysis problems. Our framework contains a number of unique features, including integrated pointer analysis and configurable precision. Our annotation language, which we present in Chapter 4, provides a user-friendly interface to these compiler mechanisms, and we use the representation and notation defined here to describe the language semantics.

3.1 Overall system architecture

The Broadway compiler is a C source-to-source translator written in C++. Figure 3.1 shows the overall architecture of the compiler. We started by implementing the C-Breeze compiler infrastructure, which consists of a C89 parser front-end, and a set of C++ classes to represent the abstract syntax tree. The Broadway compiler is built on top of C-Breeze and performs all of the library-specific analysis and optimization.

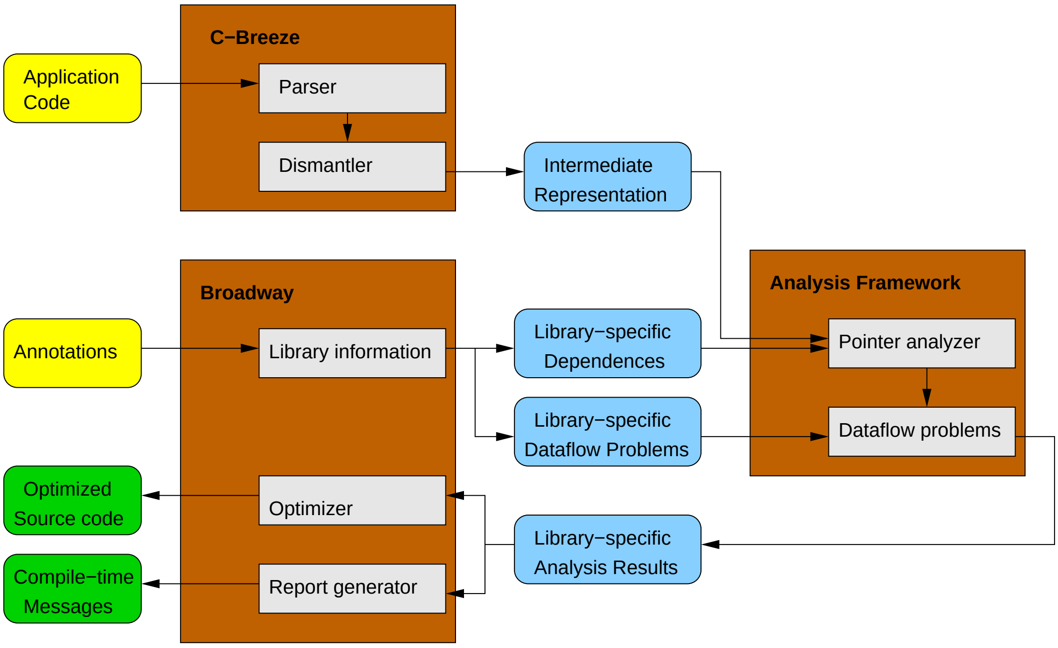

Figure 3.1: The architecture of the Broadway compiler.

At its core, the Broadway compiler is primarily a powerful program analysis engine. It consists of an iterative dataflow analysis framework, which manages the analysis algorithm, and a set of specific dataflow analysis problems, which are solved using the framework. It is an interprocedural and whole-program analysis framework, which provides the scope necessary to track information throughout the input programs. The framework includes an integrated pointer analyzer that constructs a detailed model of the program dependences and heap objects. We provide several built-in dataflow analysis passes, including constant propagation and liveness analysis, which use the memory model provided by the pointer analyzer.

We support library-level compilation using this framework in two ways. First, we integrate library routines into the built-in analysis passes, including the pointer analysis, using information provided by the annotations. For example, in order to integrate a library routine into liveness analysis, the annotations tell the compiler which variables the routine accesses and which ones it modifies. Second, we allow the annotations to define library-specific program analysis problems. The framework accepts a problem definition as input and uses iterative dataflow analysis to compute a solution for each input program. The annotations can use the results of this analysis to drive other library-level compilation tasks. This chapter focuses on the dataflow analysis algorithm, while Chapter 4 shows how the annotation language presents these capabilities to the annotator.

The analysis framework is highly configurable: it has many levels of precision and modeling. In addition to library-level compilation, we use the framework to perform general program analysis research. For example, in Chapter 7, we compare the relative benefit of several levels of precision on the same set of dataflow analysis problems. In this chapter, we describe the various precision and program modeling parameters and explain their implementation. In Chapter 7 we present our client-driven pointer analysis algorithm, which we implement as a policy on top of the configurable mechanisms. This algorithm is made possible not only by the configurability of the framework, but also by the tight coupling of the pointer analysis and the dataflow analysis in the framework. All of these analysis features are managed in a manner transparent to the annotations.

Built-in optimizations include constant propagation and constant folding, dead-code elimination, and control-flow simplification. Loop invariant code motion is not currently implemented, but it is a straightforward extension. Annotation-driven code transformations include removing a procedure call, inlining a procedure call, or replacing a procedure call with a code fragment.

3.2 C-Breeze compiler infrastructure

The Broadway compiler extends our own C compiler infrastructure called C-Breeze. C-Breeze consists of an ANSI C89 front-end, which parses source code written in C and represents it as an abstract syntax tree. The system is implemented in C++ using a clean object-oriented programming style. C-Breeze also includes a number of common analysis

Feature	Setting
Representation	Points-to sets using storage shape graph.
Flow-sensitivity	Configurable—on a per-object basis
Context-sensitivity	Configurable—on a per-procedure basis
Assignments	Uni-directional (subset-based)
Flow dependences	Factored use/def chains
Aggregate modeling	Optional—turned on by default
Program scope	Whole-program, interprocedural
Heap object naming	By allocation site (see note below)
Pointer arithmetic	Limited to pointers within an object
Arrays	All elements represented by a single node

Table 3.1: Specific features of our pointer analysis framework.

and optimization passes.

3.3 Program analysis framework

The core of the Broadway compiler is a program analysis framework. This framework provides iterative dataflow analysis supported by pointer analysis, dependence analysis, and constant propagation and constant folding. All of these analyses run concurrently and can interact with each other through a common representation of the program and the objects in the memory model. In this section we describe the details of our framework, including the overall architecture, the representation of analysis information, the analysis algorithm, and the implementation of the different precision policies. Figure 3.1 provides a concise summary of the features of our framework.

3.3.1 Program representation

In order to perform whole-program analysis, our compiler accepts as input a set of C source files, which it processes in several ways in preparation for analysis. The C-Breeze infrastructure parses the C code and builds an abstract syntax tree, which is then dismantled into a medium-level intermediate representation. This IR consists of simple assignment statements, similar to three-address instructions, organized into basic blocks, which are in turn organized into a control-flow graph. This representation preserves some of the high-level constructs of C, such as `struct` and `union` types, and array indexing.

In order to support context-sensitive analysis, we build a separate data structure that represents program locations. Using this data structure, the compiler has the option of instantiating a single instance of a procedure, or one instance for each invocation of a procedure. The location data structure is a tree constructed from three kinds of nodes:

- **Statement locations** represent individual statements in the program.
- **Basic block locations** represent basic blocks, and their children in the tree are the statement locations that represent the statements of the basic block.
- **Procedure locations** represent whole procedures, and their children in the tree are the basic blocks locations that represent the basic blocks in the procedure.

```
class procLocation {
  set< basicblockLocation > BasicBlocks
  stmtLocation * CalledBy
}

class basicblockLocation {
  set< stmtLocation > Statements
  set< memoryBlock * > PhiFunctions
  procLocation * InProcedure
}

class stmtLocation {
  exprTree * Expression
  procLocation Calls
  basicblockLocation * InBlock
}
```

Figure 3.2: Pseudocode representation of the location tree types

Figure 3.2 shows a simplified summary of the types that represent the location tree. In addition to containing the nodes below it, each node also includes a reference to its parent in the tree. Notice that when a statement is a call to another procedure, the statement location node has a single child, which is the procedure location node of the callee. Figure 3.3 shows an example code fragment, including a procedure that is called in two places, and Figures 3.4 and 3.5 show the corresponding location tree for both context-sensitive and context-insensitive analysis modes.

```
int global;
void inc_global()
{
  int temp;
  temp = global;
  global = temp + 1;
}

int main()
{
  global = 0;
loop:
  if (cond) goto the_end;
  inc_global();
  inc_global();
  goto loop;
end;
  return 1;
}
```

Figure 3.3: This code fragment shows the need for a separate representation of program locations: the def of `global` in the first call to `inc_global()` reaches the use of `global` in the second call.

Like many compilers, our system computes the dominator tree for each procedure. Our compiler, however, also supports a fast interprocedural dominance test that captures the dominance relationship between statements in different procedures, and between statements in different invocations of the same procedure. For example, in Figure 3.3 the procedure `inc_global()` reads the value of the global variable and updates it. Since `main()`

calls the procedure twice, the assignment to global in the first invocation dominates the access of the global variable in the second invocation. Therefore, the statement on line 6 interprocedurally dominates the statement on line 5, even though the opposite in true for intraprocedural dominance. The context-sensitive location tree in figure 3.5 shows how we solve the problem: it contains two instance of each statement, with different dominance relationships between them.

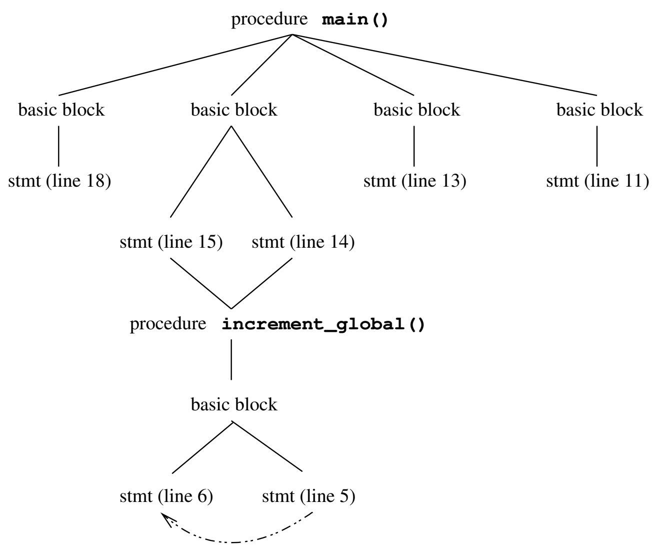

Figure 3.4: The context-insensitive location tree: the dashed arrow shows that the statement on line 5 dominates the statement on line 6.

One problem with constructing an interprocedural dominance tree is that the tree can become extremely deep, which makes the dominance test expensive. To avoid this cost, we use a tree numbering scheme that allows a constant-time dominance test [92]. The numbering scheme is computed by a depth-first search on the dominator tree that assigns consecutive numbers to each node it visits. The key is that it visits each node twice, once on the way down the tree in preorder traversal and once on the way back up in postorder traversal. The result is that each node in the tree has two numbers, and more importantly, the numbers associated with all the descendants of a node fall within the range of those two numbers. With these number pairs, we can test dominance between any two statements in

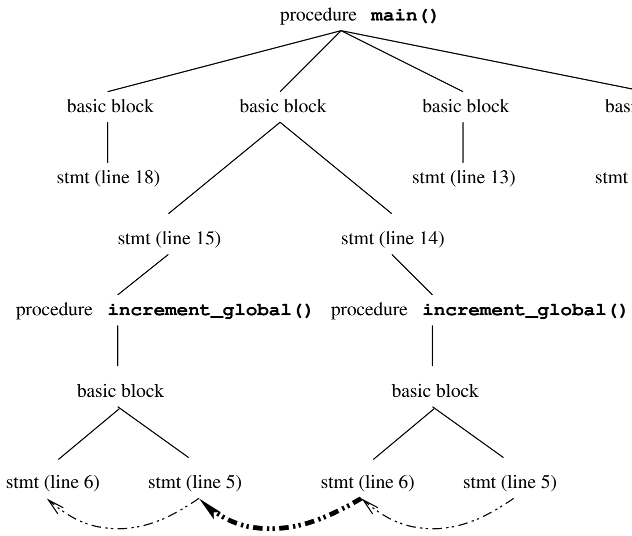

Figure 3.5: The context-sensitive location tree: notice the middle dashed arrow, which captures the fact that the statement on line 6 dominates the statement on line 5 across procedure invocations.

program with just two integer comparisons. Figure 3.6 shows an example of this numbering.

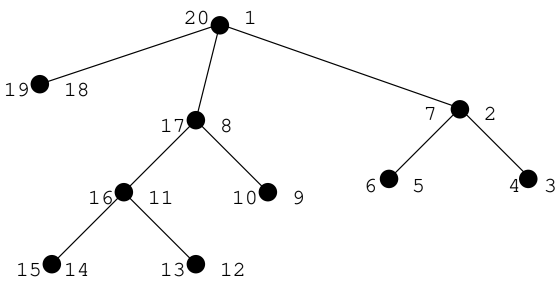

Figure 3.6: We use this numbering of the dominator tree to speed up the dominance test: the numbers assigned to all the descendants of a node fall in between the two numbers assigned to that node.

It is impractical to number all the locations in a program before starting the analysis. Therefore, we formulate an online version of the numbering algorithm that numbers program locations as they are processed by the analyzer. In order to produce the correct numbering, we order the basic block worklists according to a preorder traversal of the intraprocedural dominator tree. The algorithm performs the postorder portion of the numbering when it reaches a basic block that does not dominate any others.

3.3.2 Memory representation

Our representation of the objects in a program is based on the storage shape graph [18]. We adopt this representation for C programs, and we include a number of improvements from Wilson et al. [96].

The vertices of our storage shape graph are called *memoryBlocks*, and they represent all addressable objects in memory, such as variables, structures, arrays, and heap allocated memory. We decompose complex objects into finer structures in order to more accurately model their behavior. For example, each field of a structure is represented by a separate node, and each instance of a structure includes a full set of these field nodes. We represent all the elements of an array with a single memoryBlock.

We store memoryBlocks according to the location where they are created in the program. We index global variables using their declarations, since there is only one instance of each global. For local variables, however, we use a combination of the local declaration and the program location of the procedure. Since the program location can be context-sensitive, this indexing produces the expected behavior for different precision policies: in the context-insensitive case, the compiler generates one memoryBlock for each local variable, while in the context-sensitive case, the compiler produces one memoryBlock for each local variable in each calling context.

We index heap-allocated memory according the program location of the allocation—typically, a call to `malloc()` or `calloc()`. By using the program location as the index, we obtain the same naming behavior for heap allocated memory as for local variables: in the context-insensitive case, the compiler generates one memoryBlock for each static call to `malloc()`, while in the context-sensitive case, the compiler generates one memoryBlock for each compile-time instantiation of the call to `malloc()`.

Our storage shape graph uses two types of directed edges to connect related memoryBlocks: *containment* edges, which connect structure and array memoryBlocks to the objects they contain, and *points-to* edges, which connect a pointer to its target. A single structure memoryBlock might have multiple outgoing containment edges, one for each field it contains. In addition, our pointer analysis algorithm computes "may" points-to information, which conservatively allows a pointer to have multiple targets when the exact target cannot be determined. Therefore, a single memoryBlock might also have multiple outgoing points-to edges. Multiple incoming points-to edges indicate multiple ways of accessing an object, also called multiple *aliases*. The semantics of containment, however, require a single memoryBlock to have no more than one unique incoming containment edge.

Unlike containment, pointer relationships can change during the execution of a program. By default, our analyzer records this information in a flow-sensitive manner: points-to edges are organized into groups according to where in the program they are valid. We represent this information compactly by only recording the program locations where the information changes and then using the dominance test to determine the value at any other point.

Each memoryBlock has an associated list of *memoryDefs*, one for each program location that modifies the memoryBlock. Each memoryDef records the points-to edges established at that location, if there are any. In addition, each memoryBlock has an associated list of *memoryUses*, one for each program location that reads the value of the memoryBlock. For each memoryUse, the compiler computes the reaching definition, which is the memoryDef whose value reaches the use. We use a modified static single assignment form [24], which is described below, to make sure that each use has a single unique reaching definition. Figure 3.7 shows an example code fragment, and Figure 3.8 shows the corresponding storage shape graph.

Figure 3.9 shows a simplified summary of the types used to represent memory objects. Each memoryBlock has a set of uses and definitions. For structs, unions, and arrays, each memoryBlock also includes a set of contained memoryBlocks, which are indexed by

```
struct { int size;
         int * data; } my_array;
int * p;

if (some_condition)
  p = &x;
else
  p = &y;

my_array.data = p;
```

Figure 3.7: Example code fragment that establishes several pointer relationships.

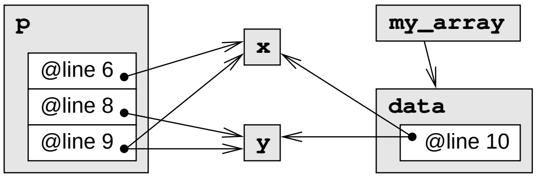

Figure 3.8: The storage shape graph for the code fragment in Figure 3.7.

the name of the field. The `Location` type refers to the node in the location tree where either the use or definition occurs.

Our general approach to flow-sensitive dataflow analysis is to associate dataflow facts with memoryUses and memoryDefs. The framework takes care of determining which memoryBlocks are read or modified, even if the accesses occur indirectly through pointers. This approach provides dataflow analysis problems with an accurate model of program behavior, without exposing the details of the pointer analysis.

3.3.3 Analysis algorithm

Our analysis framework is based on the iterative dataflow analysis algorithm introduced by Kildall [61]. We extend the algorithm in a straightforward way to interprocedural analysis: when the analyzer encounters a procedural call, it immediately begins analyzing the body of the callee procedure. Figure 3.10 shows a pseudo-code outline of the overall analysis algorithm. We define the various functions that make up this algorithm in the discussion

```
class memoryBlock {
  set< memoryDef > Defs
  set< memoryUse > Uses
  map< string, memoryBlock > Contains
}

class memoryDef {
  Location * Where
  set< memoryBlock * > PointsTo
}

class memoryUse {
  Location * Where
  memoryDef * ReachingDef
}
```

Figure 3.9: Pseudocode types for the memory representation

that follows.

The analysis framework performs two main tasks. First, it analyzes statements in the program and builds interprocedural factored def/use chains for the various memoryBlocks in the program, including pointers. Second, it manages other dataflow analysis problems through a series of hooks. Overall convergence of the analysis occurs when all of the analyses involved converge individually.

Single statement

In our dismantled program representation, each statement of a program is a simple assignment consisting of a right-hand side expression and a left-hand side expression. The right-hand side expression is limited to one computational operator, such as addition or multiplication, and one or more memory operators, such as pointer deference or array index. The analysis framework performs the following steps on each statement:

1. Evaluate the right-hand side, applying pointer operations as needed to determine the actual memoryBlocks that are accessed.

```
AnalyzeProcedure(procLocation P)
{
  // -- Manage basic blocks in a worklist
  worklist = P.BasicBlocks
  while (worklist not empty) {
    bb = remove next basicblockLocation
    set< memoryBlock > changes

    // -- Handle merge points for this basic block
    EvalPhiFunctions(bb, changes)

    // -- Visit each statement, record values that change
    for (all stmtLocations s in bb.Statements)
      if (schanges += EvalStatment(s)

    if (changes not empty) {
      PlacePhiFunctions(bb, changes)
      worklist += all basic blocks reachable from bb
    }
  }
}
```

Figure 3.10: Pseudo-code outline of our interprocedural/intraprocedural dataflow analysis algorithm.

2. Generate memoryUses for the right-hand side memoryBlocks and find their reaching definitions.

3. Collect the points-to edges for the right-hand side memoryBlocks.

4. Evaluate the left-hand side, applying pointer operations as needed to determine the actual memoryBlocks that are modified.

5. Generate memoryDefs for the left-hand side memoryBlocks and label them with the current program location.

6. Transfer the collected points-to edges to the left-hand side memoryDefs.

The assignment algorithm is shown in more detail in figure 3.11. We store the set of memoryDefs associated with a memoryBlock in a particular order that allows us to

```
EvalAssign(s, changes)
{
  set< memoryBlock * > rhs = EvalExpr(s.right)
  set< memoryBlock * > lhs = EvalExpr(s.left)
  // -- Collect points-to sets for right-hand side
  foreach r in rhs {
    memoryDef reaching = FindDominatingDef(s, r.Defs)
    rhs_points_to += reaching.PointsTo
  }
  // -- Is this a weak update?
  bool weak = lhs.size > 1
  // -- Transfer to lhs variables
  foreach l in lhs {
    memoryDef current_def = DefAt(s, l.defs)
    // -- Get the old points to set at this statement
    old_points_to = current_def.PointsTo
    // -- Construct the new points-to set
    new_points_to = old_points_to + rhs_points_to
    if (weak) {
      memoryDef reaching = FindDominatingDef(s, l.Defs)
      new_points_to += reaching.PointsTo
    }
    // -- Store new points-to set
    current_def.PointsTo = new_points_to
    // -- Did it change?
    if (new_points_to != old_points_to)
      changes += l
  }
}
```

Figure 3.11: Algorithm for evaluating an assignment

quickly find the reaching definition for any program location [96]: a memoryDef is never preceded in the list by another memoryDef that dominates it. We can find the reaching def by searching the list linearly: the first memoryDef that dominates the current program location is the nearest reaching definition. We use the same search to find the position for new memoryDefs in the list. This approach is not as fast, asymptotically, as the dominator skeleton tree proposed by Chase et al. [18], but it works well in practice.

The evaluation function recursively descends an expression, evaluating each part of the expression according to the following rules:

- **Identifier**: We look up the memoryBlock for the identifier in the memory model using the declaration and the current program location.

- **Dot operator**: Given a set of memoryBlocks and a structure field name, we follow the containment links associated with the field and return the memoryBlocks for the field. This operator creates fields on demand as they are encountered.

- **Star operator**: Given a set of memoryBlocks, we find their reaching definitions and follow the points-to edges to find the targets of the pointers.

- **Arithmetic**: We follow a commonly used model of pointers that views pointer arithmetic as a no-op. The rationale for this view is that pointer arithmetic is used to iterate through objects, but never to move between disconnected objects in memory. In particular, since we represent all elements of an array as a single memoryBlock, iterating through these elements has no effect on our model.

- **Memory allocation**: We look up the memoryBlock associated with the allocation site. Note that since the allocation site is specified using a node in the location tree, the context sensitivity of the analysis affects the number of memoryBlocks instantiated (see Section 3.3.4).

- **Procedure call**: We suspend analysis of the current procedure and immediately begin analyzing the callee. We evaluate the name of the procedure to determine the callee, and we assign the actual parameters to the formal parameters.

Since our pointer analysis is a "may" analysis, the left-hand side of an assignment might evaluate to a set of memoryBlocks, rather than one unique memoryBlock. In this case, the analyzer cannot tell which of the objects is actually modified. Figure 3.12 shows an example of this situation: the assignment through `p` could modify either `x` or `y`, but not both. We represent the two possibilities conservatively by applying a *weak update* to the variables, which merges the previous reaching value of the variable into the new value. A weak update captures the fact that an assignment might affect a variable, depending on what the actual target of the pointer is at run-time. Weak updates hurt accuracy but are necessary for correctness.

```
if (some_condition)
  p = &x;
else
  p = &y;

(*p) = 10;
```

Figure 3.12: A weak update occurs when the left-hand side of an assignment represents more than one possible object.

Since our algorithm is iterative, the analyzer often visits a single statement multiple times. Each time the evaluation function applies the left-hand side updates, it records whether any of the points-to sets change. It passes back to the main algorithm a list of the memoryBlocks whose points-to sets change.

Basic block

Traditional analysis of reaching definitions allows multiple definitions to reach each use of a variable [3]. We use a factored representation, similar to static single assignment form (SSA), which ensures that each use has a single unique dominating definition. Our representation inserts "phi" functions at control-flow merge points that merge information from multiple reaching definitions. Like SSA form, we insert these merge functions, as necessary, at the start of each basic block. However, unlike SSA form, we record these merge functions in a separate data structure rather modifying the code of the procedure. The reason is that assignments through a pointer can generate definitions of different variables in different calling contexts. During context-sensitive analysis, we only want to merge the actual target of the pointer. Figure 3.13 shows a code fragment that demonstrates this problem. The procedure `foo` needs a phi function after the condition to merge the changes to the target of p. The target of p, however, could be either x or y, depending on the calling context.

```
void main()
{
  int x;
  int y;

  foo(&x);
  foo(&y);
}

void foo(int * p)
{
  if (some_condition)
    (*p) = 6;
  else
    (*p) = 7;
  // -- Merge x or y ?
}
```

Figure 3.13: We cannot insert phi functions into the code because different variables are merged in different calling contexts.

In order to keep the merge points separate in each context, we store the list of memoryBlocks to merge on each basicblockLocation. The analyzer processes each basic block by first looking up any phi functions and merging the information associated with those variables. It then visits each statement, evaluating the expressions as described above, and collecting a list of modified memoryBlocks.

At the end of a basic block, the analyzer processes the list of changes and inserts phi functions according to the standard SSA algorithm. For each changed memoryBlock, it generates a phi function at each basic block in the dominance frontier of the current basic block. Since our analysis is interprocedural, the dominance frontier might be higher in the call stack. For example, if a change occurs at the exit basic block of a procedure, then we need to insert phi functions in the dominance frontier of the call site. Figure 3.14 shows the two components of the algorithm that handle merge points.

```
EvalPhiFunctions(bb, changes)
{
  // -- For each block merged at this location
  foreach m in bb.PhiFunctions {
    // -- Collect the incoming points-to sets by
    //    finding the defs that dominate the predecessors
    foreach predecessor pb of bb in the control-flow graph {
      memoryDef phi_input = FindDominatingDef(end of pb, m.Defs)
      merge_points_to += phi_input.PointsTo
    }
    memoryDef current_def = DefAt(bb, m.defs)
    // -- Get the old points to set at this statement
    old_points_to = current_def.PointsTo
    // -- Store new points-to set
    current_def.PointsTo = new_points_to
    // -- Did it change?
    if (new_points_to != old_points_to)
      changes += m
  }
}

PlacePhiFunctions(bb, changes)
{
  // -- Add the changed memoryBlocks to the set
  //    of phi functions on each basic block in
  //    dominance frontier of bb
  foreach fbb in DominanceFrontier(bb)
    fbb.PhiFunctions += changes
}
```

Figure 3.14: The components of the algorithm that evaluate and place phi functions in order to maintain SSA form

Procedure

The analyzer processes each procedure using a worklist of basic blocks. We order the worklist according to a depth-first search of the dominator tree, which is required by our numbering scheme. This ordering also conforms to a reverse post-order traversal of the control-flow graph, which ensures that each basic block is visited only after all of its predecessors have been visited. This ordering helps the analysis to converge more quickly.

At the start of the procedure, the analyzer puts all basic blocks on the list. The analyzer repeatedly takes a basic block off the list and processes it as described above. If any changes occur, it puts all of the basic blocks that are reachable in the control-flow graph back on the list. Previous research has showed that the performance of dataflow analysis is sensitive to worklist management. Our observations confirm the findings of Atkinson et al. [6] that the analysis converges more quickly when the analyzer visits all the basic blocks in order before going back to earlier blocks. When we change the algorithm to revisit earlier basic block as soon as possible, the analysis runs twice as slowly.

When analysis of the procedure finishes, the analyzer collects all of the changed memoryBlocks and filters them to produce a list of changes that are visible to the caller. For example, it removes local variables but keeps global variables. It passes these externally visible changes back to the caller.

3.3.4 Context insensitive analysis

The analysis algorithm described thus far is a fully context-sensitive algorithm: every invocation of a procedure is analyzed in its own context. While this algorithm produces extremely accurate results, the cost can grow exponentially in the size of the call graph. Therefore, we give the analyzer the ability to treat some or all of the procedures as context-insensitive. Context insensitivity reduces the precision of the analysis but it speeds up the analysis considerably.

Our strategy for implementing context-insensitive analysis is to instantiate the location tree for a procedure only once, as shown in Figure 3.4. The effect is that the analyzer generates only one instance of each local variable, including the formal parameters. Therefore, the one value associated with each formal parameter accumulates all the values of the actual parameters from all the call sites.

The problem with this approach is that by instantiating the location tree only once, we can no longer use the interprocedural dominance test. We address this problem by making explicit any memoryBlocks used or modified by a procedure that are visible outside the procedure. We refer to these memoryBlocks as the external inputs and outputs of the procedure, and we treat them as if they were real parameters to the procedure. These external inputs and outputs are analogous to the "extended parameters" introduced by Wilson et al. [96].

Each external input has a assignment at the interface of the procedure with a reaching definition in each calling context. This special def merges the information from the different contexts. Each external output has a def in each calling context that gets its last value from within the procedure. When copying a changed value back to all the calling contexts, we may need to force the analyzer to revisit those callers.

Context insensitivity speeds up the analysis for two reasons. First, the analysis of a context-insensitive procedure converges more quickly because most of its behavior is the same regardless of the calling context. Second, we can use the formal parameters and the external inputs to skip the analysis of a procedure completely: since we make explicit all of the input memoryBlocks, when none of them change there is no reason to analyze the procedure body.

3.3.5 Flow insensitive analysis

Another technique for speeding up dataflow analysis is to give up flow sensitivity. Flow-insensitive analysis does not keep dataflow facts from different parts of the program separate. In this mode, the analyzer stores only one flow value for each memoryBlock, and each update to a memoryBlock merges new dataflow facts in with the previous information. Flow-insensitive analysis dramatically reduces the cost of analysis, but it also severely degrades accuracy.

We implement flow-insensitive analysis on a per-memoryBlock basis by forcing each flow-insensitive memoryBlock to have only one memoryDef. All updates to the points-to set, and any other dataflow information, are accumulated in that one memoryDef, which the analyzer configures so that it dominates the whole program.

```
          // Traditional          Our implementation
p = &x;   //  p -> {x}                 p -> {x}
q = p;    //  q -> {x}                 q -> {x}
p = &y;   //  p -> {x,y}, q -> {x,y}   p -> {x,y}, q -> {x}
```

Figure 3.15: Our implementation of flow-insensitive analysis is more precise than the traditional definition because we respect the ordering of the statements.

Our implementation of flow-insensitivity is not identical to the traditional definition of flow-insensitivity because we still visit statements in a specific order: the order in which they are executed. As a result, our flow-insensitive analysis is more precise than an analysis that completely ignores the ordering of the statements. Figure 3.15 shows an example that highlights the difference in our algorithm. In a traditional flow-insensitive analysis, the presence of an assignment, such as `q = p`, forces the two variables to always be equal. In our implementation, we take advantage of the fact that the second assignment to `p` occurs after the assignment `q = p`, and therefore it cannot affect the value of `q`. Note that we continue to use iterative analysis even for flow-insensitive variables, which ensures correctness in loops.

3.3.6 Recursion

Recursion presents a challenge for context-sensitive analysis: the number of calling contexts of a recursive procedure is not fixed at compile-time. Our solution to this problem is to use context-insensitive analysis on all recursive procedures. This policy also includes all procedures involved directly in mutual recursion: all the procedures that belong to strongly connected components in the call graph. We iterate over the recursive cycle until no new changes occur.

3.3.7 Multiple instance analysis

Our analyzer treats the memoryBlocks that represent heap allocated memory differently from other memoryBlocks because heap allocation is a dynamic property of programs. In particular, a single allocation site (call to `malloc()`) can generate an unbounded number of objects at run-time. Since our representation creates one memoryBlock for each allocation site, that memoryBlock logically represents many possible memory locations. In order to maintain correctness, every update to a heap allocated object must be a weak update.

```
while (cond) {
  p = malloc(sizeof(Element));
  p->next = head;
  head = p;
}

head->value = 5;
head->next->value = 10;
x = head->value;
```

Figure 3.16: We apply weak updates to heap allocated memoryBlocks because they can represent multiple objects at run-time.

Figure 3.16 shows a code fragment that demonstrates this weak update requirement. The loop creates a linked list of heap allocated elements, and the subsequent statements set the first element to 5 and the second element to 10. Notice that the code contains only one call to `malloc()`, so the linked list is represented by a single memoryBlock with a pointer to itself. (Technically, it has a field called “next” that points back to the container.) If we allow strong updates to this memoryBlock, then the first assignment will set it to 5 and the second assignment will overwrite this value with 10. As a result, the analyzer will conclude that `x` equals 10, which is not correct. Applying a weak update forces the analyzer to include the previous value, which safely sets the value of `x` to lattice bottom.

It is often the case, however, that a heap allocated memoryBlock does represent a single object at run-time. Such a memoryBlock would admit strong updates, which improve the accuracy of the analysis. Our analyzer uses a special supplementary dataflow analysis, called *multiple instance analysis*, that attempts to determine whether an allocation site can generate multiple objects [18]. Our implementation of this analysis marks each heap allocated memoryBlock with a value, which we call *multiplicity*, that indicates its allocation status. The multiplicity values form a vertical lattice (a total order) from top to bottom:

- **Unallocated**: Represents the state of heap memoryBlocks before allocation and after deallocation.
- **Single**: Marks heap memoryBlocks that represent a single heap object at run-time.
- **Unbounded**: Marks heap memoryBlocks that may represent multiple heap objects at run-time.

Multiple instance analysis is flow-sensitive, and it updates the state of heap memoryBlocks at each allocation and deallocation. Figure 3.17 shows the transfer functions that the analyzer applies at each allocation, and Figure 3.18 shows the transfer functions that the analyzer applies at each deallocation. Notice that once a memoryBlock enters the unbounded state, there is no transition that allows it to return to the single state.

Before allocation		**After allocation**
Unallocated	$\rightarrow$	Single
Single	$\rightarrow$	Unbounded
Unbounded	$\rightarrow$	Unbounded

Figure 3.17: Transfer functions for multiple instance analysis at an allocation site.

Figure 3.19 shows a code fragment with two loops: one allocates many objects, while the other only allocates one at a time. During analysis, the call to `malloc()` in the first loop causes the multiplicity value of the allocated memoryBlock to go from unallocated to single in the first iteration, and from single to unbounded in subsequent iterations. The call to `malloc()` in the second loop causes the multiplicity value of its memoryBlock to go

Before allocation		After allocation
Unallocated	→	Unallocated
Single	→	Unallocated
Unbounded	→	Unbounded

Figure 3.18: Transfer functions for multiple instance analysis at a deallocation site.

from unallocated to single. The call to `free()`, however, causes the value to return to the unallocated state. The multiplicity analysis converges with a multiplicity value of single, which allows all computations on the memoryBlock in the `compute_using()` function to admit strong updates.

```
while (cond) {
  p = malloc();       // Unallocated -> Single -> Unbounded
  p->next = head;
  head = p;
}

while (cond) {
  q = malloc();       // Unallocated -> Single
  compute_using(q);
  free(q);            // Single -> Unallocated
}
```

Figure 3.19: Our multiple instance analysis properly determines that the top loop allocates many objects, while the bottom loop only allocate one at a time.

3.4 Dataflow analysis

The analyzer described so far builds a model of a program's pointer behavior and computes reaching definitions. This information is used to compute solutions to dataflow analysis problems. We use the same sparse iterative algorithm for these problems as we do for the pointer analysis—both kinds of analysis run at the same time. We represent dataflow facts as flow values and associate them with the uses and defs of memoryBlocks. The framework manages the propagation of the information and the convergence of the problem.

3.4.1 Defining a dataflow analysis problem

We adopt the traditional definition of dataflow analysis problems, which consists of a set of flow values and a set of transfer functions [3]. The flow values capture analysis information and the transfer functions define how statements in the program affect this information. The flow values form a dataflow lattice, which includes a *meet* function to conservatively combine flow values along different paths in the program. In the current implementation of Broadway, we focus on a particular class of sparse dataflow analysis problems that associate flow values with individual memory blocks. This allows us to use the same basic mechanisms to manage both the pointer analysis and other dataflow analyses.

For pointer analysis, the flow value consists of the points-to set and its meet function is set union. We can adapt the framework for other dataflow analysis problems by mirroring the pointer analysis algorithm, but replacing the points-to operations with general flow value operations. Figure 3.20 shows the assignment evaluation modified in this manner. Notice that the expression evaluation for the left and right sides uses the pointer analysis evaluation function to determine the memoryBlocks involved. We augment the algorithm for handling phi functions is a similar way, replacing the set-union operations with general meet functions.

3.4.2 Implementation

In order to perform other kinds of analysis, the analyzer contains "hooks" at various places that integrate other dataflow analysis problems. We implement these hooks in an object-oriented style using two abstract classes. This first class is called analysisProblem, and it represents the transfer functions for a dataflow analysis problem. It contains pure virtual functions for different program constructs. The second class represents the flow value for the analysis problem.

```
EvalAssign(s, changes)
{
  set< memoryBlock * > rhs = EvalExpr(s.right)
  set< memoryBlock * > lhs = EvalExpr(s.left)
  // -- Collect flow values for right-hand side using Meet
  foreach r in rhs {
    memoryDef reaching = FindDominatingDef(s, r.Defs)
    rhs_flow_value = Meet(rhs_flow_value, reaching.FlowValue)
  }
  // -- Is this a weak update?
  bool weak = lhs.size > 1
  // -- Transfer to lhs variables
  foreach l in lhs {
    memoryDef current_def = DefAt(s, l.defs)
    // -- Get the old flow Value
    old_flow_value = current_def.FlowValue
    // -- Construct the new flow value
    new_flow_value = Meet(old_flow_value, rhs_flow_value)
    if (weak) {
      memoryDef reaching = FindDominatingDef(s, l.Defs)
      new_flow_value = Meet(new_flow_value, reaching.FlowValue)
    }
    // -- Store new flow value
    current_def.FlowValue = new_flow_value
    // -- Did it change?
    if (new_flow_value != old_flow_value)
      changes += l
  }
}
```

Figure 3.20: General dataflow analysis assignment algorithm

To define a new dataflow analysis problem, we create a concrete subclass of both abstract classes. We design the flow value class to hold whatever information we want to collect, such as constant values or common subexpressions. We then override functions in the analysisProblem class to describe how various program constructs affect the flow value. For example, in constant propagation we implement the transfer function for binary operators by applying the binary operator to the constant values of the left and right operands.

The analyzer manages the traversal of the program, calling the user-defined transfer functions where appropriate. In addition, it hides many of the pointer operations from other analysis problems and instead it provides the actual sets of objects to which pointer expressions refer. For example, when processing an expression such as `(*p) + (*q)`, the analyzer handles the two indirections, and passes the resulting sets of objects to the binary operator hook.

3.5 Summary

The Broadway compiler infrastructure consists of a set of compiler mechanisms for library-level compilation. These mechanisms include many familiar compiler algorithms that have been augmented or extended to support the requirements of library-level compilation. In particular, the program analysis engine provides a powerful and flexible framework for solving both traditional and library-specific dataflow analysis problems. It includes an integrated pointer analyzer that provides the precision needed to analyze library data structures. In Chapter 4, we show how the annotation language provides access to these compiler mechanisms without exposing the annotator to the details of their implementation.

Chapter 4

Annotation language

This chapter describes the details of our annotation language, which captures library-specific information for use in the Broadway compiler. Using this language a library expert can easily integrate library routines into traditional compiler passes and define new library-specific analyses and optimizations. We address the tradeoff between the expressive power of the language and its usability by including many useful compiler mechanisms, such as program analysis, code transformations, and traditional optimizations, as configurable compiler tools. The annotations only need to provide enough information to configure these mechanisms, not to define entirely new compiler passes. This chapter presents an in-depth discussion of how this information is expressed, including the full syntax and semantics of the language. In Chapters 5 and 6 we show how to use the language to implement several library-level error detection and optimization passes.

4.1 Language design

The Broadway annotation language is a lightweight specification language that allows a library expert to capture domain-specific information and communicate it to the compiler. This language is arguably the most important component of our system because it defines exactly what domain-specific compilation capabilities we support. The key to our language design is that we view the annotations as configuration information for the Broadway compiler mechanisms, rather than as a way to express entirely new compiler algorithms. This approach reduces both the amount of information contained in the annotations and their complexity. While it is clear that more sophisticated specifications could support more sophisticated capabilities, we show that a few simple annotations can enable many useful library-level compilation tasks. Simplicity is critical because we expect annotators to be library experts who do not necessarily have expertise in compilers or formal specifications.

Since the capabilities of the language reflect the capabilities of the compiler, our discussion of the language necessarily includes many specific details of the underlying compiler algorithms. In many cases, however, we present only an informal description of these algorithms and the semantics of the language constructs that use them. Chapter 3 contains a complete discussion of the compiler implementation.

4.1.1 Capabilities

In Chapter 2 we outlined several specific goals for library-level compilation. The role of the annotation language is to express the domain-specific information needed to support these capabilities:

- Integrate library calls into traditional analyses and optimizations.
- Check programs for library-specific errors.
- Modify programs using library-specific code transformations.

We address the first goal by including in Broadway a number of traditional optimizations that we have modified to use information from the annotations when processing library calls. These versions treat most C language features in the traditional way, but when they encounter a call to an annotated library routine, they consult the annotations for the information they need. The advantage of this approach is that many of these passes only require simple dependence information. For example, dead code elimination relies on liveness analysis to determine which computations produce results that are unused. It computes this information by determining which variables are accessed and modified at each statement. Therefore, to extend dead code elimination to library calls, the liveness analysis just needs to know which variables the library call accesses and modifies.

In order to achieve the second and third goals, we manually studied the error checking and optimization opportunities that several real libraries present. In many cases it is the attributes of the objects passed into a library routine that determine when the call is erroneous or when a more efficient call is applicable. For example, it is an error to call a file read routine when the file stream object is not open for reading; similarly, we can replace a general matrix multiply routine with a more efficient algorithm when the input matrices are triangular. Therefore, the central language feature supporting library-specific compilation is the ability to define domain-specific characteristics, such as “open” or “triangular”, and associate them with objects in the application code. We refer to these as library *properties*, and by making them explicit, we can analyze programs directly in terms of the concepts and abstractions provided by the library.

Library-specific properties are similar to types in that they represent classes of objects that share common features. Unlike types, however, library properties can represent the object’s state over time: the property associated with a particular object can change. For example, a file stream might be open at one point in a program and closed at another. Whether or not the program contains a file access error depends on the state of the file stream at the place in the program where the access occurs. Therefore, in addition to defining the set of possible properties, the annotations also define how each routine affects the properties of the objects on which it operates. This design allows the compiler to treat each object as a finite state machine, with the library routines serving as the transition functions.

The Broadway compiler computes the property values for a program using iterative dataflow analysis [3, 61]. These properties drive all library-level error detection and opti-

```
f = fopen(filename, "r");   // Transition: closed --> open
fgets(buffer, 100, f);      // Okay, state is open
fclose(f);                  // Transition: open --> closed
fgets(buffer, 100, f);      // Error, state is closed
```

Figure 4.1: We can check file accesses by associating states, such as "open" and "closed", with the file objects.

mization capabilities: the annotations test the properties of library objects to decide whether to emit an error message or apply a code transformation. As a result, one of the most important mechanisms in the Broadway compiler is its analysis framework, which derives the property information for each input program. The analysis framework takes the property definitions, including the effects of each routine, and uses dataflow analysis to compute a consistent assignment of properties to objects in the program. The properties serve as the flow values for this analysis, and the effects of each library routine define the transfer functions. The language provides a mechanism to test the final computed values and either emit a message or perform a code transformation.

Library-level analysis problems can place considerable demands on a dataflow analysis engine. An object's state at any given point in the program is determined by the sequence of library functions applied to it. It is a straightforward task to compute the state by applying the sequence of property transitions that correspond to the sequence of library calls, as defined by the annotations. However, what makes this analysis challenging is determining what the sequence is—especially since we must determine it statically, at compile time. As with any static analysis, we often have to treat control flow conservatively. For example, at a conditional branch we can rarely tell which branch is taken and which is not. Therefore the analysis has to assume that either could be taken by analyzing both paths and merging the information together. In addition, the scope of the analysis needs to cover the whole program. For example, at a call to `fgets()` we might have to search all the back to the start of the program to find a matching call to `fopen()`. Library objects often have lifetimes that span many procedures, or even the entire program.

An added complication is that many libraries use pointers to pass objects from one call to another. Therefore, we must provide some form of pointer analysis just to determine which object a particular call manipulates. As an example, the code fragment in Figure 4.2 opens a file using one variable, but it closes the file using a different variable. Without pointer analysis, we could not tell that both variables refer to the same file stream, and we would probably miss the error. For a more difficult example, consider the code fragment in Figure 4.3. Here the alias is created internally by the call to `fileno()`, which returns a file descriptor that refers to the same file as the input stream. Also, notice that a file descriptor is just a small integer, not a pointer type.

```
FILE * f, * g;
f = fopen(filename, "r");   // Open using f
g = f;                      // g and f represent the same file
fclose(g);                  // Close using g
fgets(buffer, 100, f);      // Error: file is closed
```

Figure 4.2: We need pointer analysis to prevent aliases from obscure library behavior.

```
FILE * f;
int fd;
f = fopen(filename, "r");   // Open using f
fd = fileno(f);             // fd and f refer to the same file
close(fd);                  // Close using fd
fgets(buffer, 100, f);      // Error: file is closed
```

Figure 4.3: Some aliases are created by the library itself, and may not even involve pointer types.

To address these problems, we include annotations specifically for describing pointer behavior. A library expert uses these annotations to indicate when a library routine dereferences pointers and to describe any new pointer relationships it creates. We can then associate the properties directly with the underlying objects regardless of how they are stored or accessed.

4.1.2 Usability

The capabilities outlined above suggest a large and complicated language: one that supports detailed pointer analysis, custom dataflow analyses, error reporting, and code transformation. Our challenge in designing the language is to provide these capabilities while keeping the language usable for non-compiler experts. Our solution is to require only simple information from the annotator, but make the most of this information in the compiler. For example, we limit the property annotations to a relatively simple class of dataflow analysis problems, but we solve these problems on a powerful analysis framework. This gives the annotator access to advanced compiler algorithms without having to understand the underlying compiler theory, such as lattices and dataflow equations.

Specifically, we address language usability with the following features:

- **Simple declarative syntax.** The language favors simple declarative notation over constructs that are more complex. Part of the goal is to avoid requiring a complete formal specification of the library semantics, which is prohibitively difficult in many cases. Rather, the annotations provide a partial specification, which can vary in complexity depending on the effort of the annotator. We also avoid a procedural style specification, in which the annotator "programs" the compiler. The procedural approach is more flexible, but requires significant effort even for simple tasks.

- **Limited dataflow problems.** The property annotations can only define simple sets of object states, optionally organized into a hierarchy. The lattices for these properties are all tree-like structures, similar to subtyping relationships in object-oriented languages. As a result the compiler can automatically infer the lattice operations, such as the meet function and the test for convergence. The annotator only needs to specify the possible states and the effects of each library call on those states.

- **Powerful analysis framework.** While the library-specific analysis problems are relatively simple, the Broadway analysis framework that solves them includes many advanced features. It is a whole-program, interprocedural analyzer, with integrated pointer analysis, dependence analysis, and constant propagation. It also includes our client-driven pointer analysis algorithm, which automatically manages analysis precision. We provide all of these capabilities without exposing them directly to the annotator.

- **Minimal exposure to compiler algorithms.** We require very little additional information about library routines beyond their domain-specific behavior. We use this information to enable traditional optimizations, such as constant propagation and dead-code elimination, which provide proven optimization capabilities with little additional effort by the annotator.

- **Macro-like code transformations.** We limit code transformations to macro-like substitutions. The replacement code is just a C fragment with a few special meta-tokens that represent information from the original code. We avoid the need for a syntax that explicitly constructs code fragments, which is often tedious and error-prone.

4.1.3 Overview

The rest of this chapter describes the annotation language in detail. It is organized around the four main categories of information that the language captures:

- **Dependence information.** The language provides a convenient and concise syntax for specifying the pointer and dependence behavior of each library routine. We use these annotations to describe the shape of pointer-based data structures and to indicate how the routine accesses and modifies them. This information provides accurate dependence information that drives many traditional optimizations, such as dead-code elimination, at the library level.

- **Library-specific dataflow analysis.** The language allows the library expert to define new library-specific dataflow analysis problems. We avoid much of the complexity of specifying these problems by limiting the dataflow values to simple tree-like lattices of categories. For each library routine, the library expert specifies how the routine affects the states of the objects it manipulates—the transfer function.

- **Error messages.** The language provides a way to test the results of the library-specific analysis and emit error messages based on those results. The message syntax includes a number of special tokens to indicate contextual information, such as the location of the error in the application source.

- **Code transformations.** The language allows each library routine to specify a number of code transformations, with an optional guard condition that tests the analysis results. Currently, the transformations may only operate locally: they can only remove, replace, or inline a given call to a library routine. The code replacement syntax is similar to hygienic macro substitution, and it allows a library call to be replaced with an entire code fragment, including locally defined temporary variables.

4.2 Overall annotation structure

Each library has its own annotation file, which includes all the information necessary to compile applications that use that library. The annotation file consists of a list of top-level annotations, which fall into four categories:

1. C-code blocks: we provide a way to include native C code, including header files and other declarations.

2. Properties: These annotations define the flow values for library-specific dataflow analysis.

3. Procedures: Each procedure annotation specifies all the relevant information for one routine in the library interface, including the dependence information, pointer behavior, and its effects on object properties.

4. Global variables: The annotations can define global variables, which are useful for representing hidden or abstract state information.

For each type of annotation we give the overall purpose of the annotation, the syntax of the annotation, and an informal description of the semantics. We describe the syntax using a familiar language production notation. We use the following conventions for the grammar:

- *non-terminals*: non-terminal symbols are given in italics.
- TERMINALS: terminal symbols, such as identifiers, and numbers are given in small caps.
- **`literals`**: literals, such as keywords and operations, are given in teletype font.
- Productions consist of a left-hand non-terminal, followed by an arrow $\rightarrow$, followed by the list of terminals, non-terminals, and literals.
- We employ some common abbreviations, such as vertical bar ($|$) for alternatives, star ($\star$) for zero or more repetitions, and square brackets ([*opt*]) for optional parts of the production.

annotation-file	$\rightarrow$	*annotation* $\star$
annotation	$\rightarrow$	*header*
	$\mid$	*property*
	$\mid$	*procedure*
	$\mid$	*global*

Figure 4.4: Overall grammar for the annotations: it consists of a list of top-level annotations that include header file information, property definitions, library procedures, and globals.

Figure 4.4 shows the overall grammar for the annotation file. The rest of this chapter describes the syntax and semantics of the various annotations that make up the language. We use a fictional library of matrix operations to illustrate these annotations.

4.2.1 C-code blocks

C code blocks provide a way to include arbitrary code fragments in the annotations. The Broadway compiler parses this C code as it would parse the code in an input program, including checking syntax and constructing a symbol table. The most common use of code blocks is to include the library header files. This information serves several purposes. First, it allows Broadway to make sure that annotated library routines exist and to make sure they have the right number of arguments. Second, it provides access to special values and constants defined in the library. By including the header file, the annotator can use the defined names of these values rather than hard-wiring the actual values into the annotation file. Third, it allows the annotations to check and update the state of global variables provided by the library. Broadway can properly track the states of these variables whether the library modifies them or the application modifies them.

Figure 4.5 shows the grammar for including C code blocks. Any text between the special delimiters is parsed as ANSI C. However, the most common use of the C code blocks is simply to include the library header file. Occasionally it is also useful to define or undefine values. Figure 4.6 shows an example C code block that includes a header file, undefines a flag, and declares an external global variable.

header	$\rightarrow$	`%{`
		C CODE
		`%}`

Figure 4.5: C code blocks, enclosed in the special delimiters, provide access to information in the header file, such as macros, constants, and library interface declarations.

In order to handle preprocessor directives, Broadway first passes the entire annotation file through the standard C preprocessor. A secondary benefit of this approach is that the annotations themselves can use preprocessor features. For example, we can use conditional compilation directives, such as `#ifdef`, to control which annotations we use. We

```
%{
#include "MyMatrixLibrary.h"

#undef SOME_FLAG

extern int some_global;
%}
```

Figure 4.6: The annotations can include C code blocks, which provide access to symbols and values in the library header file.

can also include other annotation files, just as we would include other C header files.

4.3 Procedure annotations

Each library routine has a procedure annotation that holds all of the information that is specific to that routine. This annotation first gives the procedure name and the list of formal parameters, without types. The names of the parameters need not match those in the header file, but we require the number of parameters to be the same. The procedure annotation contains several other kinds of annotations, which we define later in the appropriate section. Figure 4.7 shows the syntax of the annotation.

procedure	$\rightarrow$	**`procedure`** IDENTIFIER **`(`** *identifier-list* **`)`** { *procedure-annotation* $\star$ }
procedure-annotation	$\rightarrow$	*pointer-annotation*
	|	*dependence-annotation*
	|	*analysis-rule-annotation*
	|	*report-annotation*
	|	*action-annotation*
identifier-list	$\rightarrow$	IDENTIFIER [**`,`** *identifier-list*]

Figure 4.7: Each **procedure** annotation holds all the information about a single routine in the library interface.

4.4 Pointer and dependence behavior

With very few exceptions, most library routines operate on pointers and pointer-based data structures, which represent objects in library domain. Part of the reason for this is practical: since C passes parameters by value, library routines often need to use pointers to emulate pass-by-reference so that they can update objects passed to them. More significantly, library routines often create and manipulate pointer-based data structures in order to represent complex abstractions from the library's domain. Therefore, the first step in annotating a library is describing how its routines create, traverse, and modify data structures.

4.4.1 Syntax

Each library routine has a set of pointer and dependence annotations that describes its behavior. In this discussion, we focus on the general form of these annotations, but it may be helpful to refer to Figure 4.9 for examples of the concrete syntax. There are four kinds of annotations that make up this information:

- The `on_entry` annotation describes the pointer structures expected as input to the library routine. This annotation tells the compiler how to traverse the pointer structures and it provides names to the internal objects.
- The `access` annotation lists the objects that the routine accesses. This list can refer to variables from the interface or to objects introduced by the `on_entry` annotations.
- The `modify` annotation lists the objects that the routine modifies. Since this information is only for dependence analysis there is no need to describe *how* the routine modifies them.
- The `on_exit` annotation describes any changes to the pointer structure effected by the routine.

In compiler terms, the `access` and `modifies` annotations specify the "uses" and "defs" of the routine, respectively. Note that dereferencing pointers is automatically recorded as an access, and updating a pointer is automatically recorded as a modification.

pointer-annotation	$\rightarrow$	**`on_entry`** { *pointer-structure* ⋆ }
	$\mid$	**`on_exit`** { *pointer-structure* ⋆ }
	$\mid$	**`on_exit`** { *cond-pointer-structure* ⋆ }
cond-pointer-structure	$\rightarrow$	**`if (`** *condition* **`)`** { *pointer-structure* ⋆ }
	$\mid$	**`default`** { *pointer-structure* ⋆ }
pointer-structure	$\rightarrow$	[**`I/O`**] IDENTIFIER
	$\mid$	IDENTIFIER **`-->`** [**`new`**] *pointer-structure*
	$\mid$	IDENTIFIER { *pointer-structure* ⋆ }
	$\mid$	**`delete`** IDENTIFIER
dependence-annotation	$\rightarrow$	**`access`** { *identifier-list* }
	$\mid$	**`modify`** { *identifier-list* }

Figure 4.8: Pointer and dependence annotations provide a way to describe how the library traverses and updates pointer-based data structures.

Figure 4.8 shows the grammar for these four annotations. The pointer relationships are given declaratively, rather than operationally. That is, we describe the structure explicitly, using a "points-to" operator, rather than using the C "ampersand", "star", or "arrow" operators. For example, in C we can establish a points-to relation using ampersand: `p = &x;`. In the annotation language, we provide this information more explicitly by describing the resulting structure: `p --> x`. This syntax allows us to use one operator for all the pointer relations. We can also specify structures by enclosing a group of objects in curly braces. We can use both the `-->` operator and the structures together to specify complex data structures in a simple and concise manner.

Taken together the pointer annotations specify a directed graph. Each identifier in the annotations represents a node in the graph. The `-->` annotation represents a points-to edge between two nodes. We specify containment using syntax similar to `struct` definition in C. We first give an identifier to represent the whole object, followed by a list of field names enclosed in curly braces. These two constructs can be combined in a recursive manner to create complex graph configurations. For example, the target of a pointer can be decomposed into fields, which in turn can point to other objects. When a node is contained in another node, we call it a *field node*; otherwise, we call it *top-level node*.

We require each top-level node to have a unique name within a procedure annotation. Since multiple objects may have the same fields (for example, if they represent the same underlying C struct type), field nodes are identified using a "dot" notation similar to C. We take the name of the container and add the field name, separated by a period. For nested containment, we add as many dot-separated field names as we need, starting with the unique top-level node. The annotation language grammar handles these names by including period in the lexical definition of identifiers. This scheme allows every node in the graph to have a unique name.

The pointer annotations can also express heap allocation and deallocation. The `new` keyword specifies objects that a library routine allocates on the heap. Since allocation is part of the effects of the routine, the `new` keyword is only valid within the `on_exit` annotation. The `delete` keyword specifies objects that the routine deallocates. For reasons explained below, the `delete` keyword is only valid within the `on_entry` annotation.

The pointer behavior of a routine occasionally depends on arguments passed to it. For example, a library routine might work by writing into a buffer passed in as a pointer, unless the pointer is null, in which case it allocates the buffer on the heap. In order to express this behavior, the `on_exit` annotation supports a conditional syntax, in which different pointer structures take effect depending on the truth-value of the condition. We describe the syntax and semantics of conditions later in the chapter.

Many library routines do return a value using the C return value mechanism. Rather than introduce a separate return statement annotation, we reserve the `return` identifier and use it to refer to the return value. The compiler recognizes this special identifier and propagates the updates back to the call site.

4.4.2 Example

Figure 4.9 shows some example pointer and dependence annotations for two routines from our fictional matrix library. This library represents a matrix as a structure with three fields: the number of rows in the matrix, the number of columns in the matrix, and a pointer to the data itself.

```
procedure createMatrix(rows, cols)
{
  access { rows, cols }
  modify { matrix.rows,
           matrix.cols,
           matrix_data }
  on_exit { return --> new matrix { rows,
                                    cols,
                                    data --> new matrix_data }}
}

procedure multiplyMatrix(A, B, C)
{
  on_entry { A --> matrixA { rows,
                             cols,
                             data --> dataA }
             B --> matrixB { rows,
                             cols,
                             data --> dataB }
             C --> matrixC { rows,
                             cols,
                             data --> dataC } }
  access { dataA, dataB,
           matrixA.rows, matrixA.cols,
           matrixB.rows, matrixB.cols,
           matrixC.rows, matrixC.cols }
  modify { dataC }
}
```

Figure 4.9: Annotations for a fictional matrix library.

The `createMatrix()` routine creates a new matrix structure and the underlying data object. The `access` annotations show that the routine reads the number of rows and columns. The `on_exit` annotation specifies that the routine returns a pointer to a new object called "matrix", which has three fields called "rows", "cols", and "data". The data field is itself a pointer, which points to a new object called "matrix_data".

The `modify` annotation indicates that the routine modifies "rows" and "cols" fields of the matrix object and the newly created data. Notice that we refer to the fields using their fully qualified names, which distinguishes them from the formal parameters of the same names.

The `multiplyMatrix` routine multiplies two matrices together and stores the result in the third matrix. In this case we use the `on_entry` annotation to traverse the input pointers and retrieve the underlying structures and data. The annotations provide names for these objects. For example, "dataA" refers to the object pointed to by the data field of the "matrixA" object.

In order to multiply the matrices, this routine accesses the sizes of all three matrices. All matrix objects have the same fields, so we use the qualified names to distinguish them: "matrixA.rows" refers to the number of rows in the first matrix. The matrix multiply operation itself reads data from the first two matrices, "dataA" and "dataB", and updates the third matrix, "dataC". The routine has no effect on the pointer structures, so no `on_exit` annotation is needed.

4.4.3 Semantics

The semantics of the pointer and dependence annotations consist of a mapping from operations in the annotations to operations on the underlying memory model of the analysis framework. The mapping allows the annotations to interact with the representation of the calling program. For example, pointer relationships or structure fields established in the calling program are visible and accessible to the annotations. We describe the semantics of these operations in terms of the memory representation presented in Chapter 3.

At a call to an annotated library routine, the pointer analyzer matches the graph given by the annotations with the state of the points-to graph at the library routine call site. This matching process produces a binding between identifiers in the annotations and actual nodes in the points-to graph. Subsequent operations, including pointer modification, operate indirectly on the points-to graph though these bindings.

The process of mapping the `on_entry` annotations to the actual points-to graph is entirely mechanical. The compiler starts with the bindings between the formal parameters and the actual parameters, and it proceeds top-down through the pointer annotation graph. At each edge, the source is required to be in the set of bindings. The compiler looks up the source binding and then applies the given operator (either `-->` or dot) to determine the actual targets. It then updates the bindings to include the new information and continues down the structure. When this process is complete, each node in the pointer annotation graph has a binding to a set of nodes in the actual points-to graph. In addition, the compiler applies any uses of the `delete` keyword to the specified objects so that the allocation information is updated properly.

The `on_exit` annotations specify two kinds of changes to the points-to graph: new edges and new nodes. We specify new edges using the `-->` operator, where the source and target represent existing nodes. We specify new nodes using the `new` keyword after the `-->` operator. The source node represents existing nodes, while the target is a newly created node. Each use of `new` creates a new node in the graph for every call site in the application program. This scheme helps to improve precision by distinguishing different objects created at different points in the program. For example, when analyzing a program that uses the matrix library we would like each call to `createMatrix()` to result in a separate set of nodes so that we can track the objects separately.

The conditional form of the `on_exit` annotation allows the annotator to specify different pointer behavior in different circumstances. The compiler evaluates each condition expression and applies the pointer structures of the first one whose condition is true. If none of them is true, then it applies the default structures. We present the syntax and semantics of the condition expressions later in this chapter.

4.5 Defining library-specific analysis problems

The pointer and dependence annotations described above allow the Broadway compiler to integrate library routines into traditional analysis. These annotations capture information about the library routines in terms of the semantics of the base programming language—in this case, C. In this section we describe annotations for defining new library-specific program analyses passes. The library-specific information computed by these analyses drives all of the library-specific error detection and optimization passes.

The purpose of library-specific analysis is to collect information about an application program in terms of the concepts in the library's domain. Deciding which concepts to model and how to model them varies considerably between libraries. One approach is to examine the concepts in the domain and then codify those concepts using the annotation language. Another approach is to start with the desired error detection or optimization capabilities and work backward to determine what information is needed to enable them. In the following examples, we use a combination of the two approaches.

Our example library encapsulates the notion of a matrix and provides a set of operations on matrices. In general, a matrix can represent any linear system of equations. However, matrices often occur in a number of special forms, such as triangular, symmetric, or diagonal, which represent systems of equations in particular configurations. What makes these cases special is that we can pass them to specialized versions of the library routines, which are often much more efficient. For example, multiplying two triangular matrices requires half the number of operations of the general matrix multiply. Our goal for the example library is to describe the notion of matrix special forms, which we will use later to select more efficient library calls when possible, and to make sure that these specialized library routines are only used when appropriate.

Each library-specific analysis problem consists of two parts: a `property` annotation that defines the information to collect, and a series of `analyze` annotations, one for each library routine, that describe how the information propagates. The annotation file for a single library may define multiple analysis problems that collect different information. The property annotation gives a name for the property, followed by a specification of the particular information to collect. Figure 4.10 shows the grammar for properties.

property	$\rightarrow$	**`property`** IDENTIFIER **:** *enum-property-definition*
	$\mid$	**`property`** IDENTIFIER **:** *set-property-definition*

Figure 4.10: Each property defines a single analysis problem. It consists of a name and a specification of the information to collect.

The annotation language supports two kinds of properties: *enum properties* and *set properties*. As their name suggest, enum properties consist of a set of named values. The analyzer associates one or more of these named values with each object in the program. The set properties manage collections of objects or relations between objects.

The Broadway analysis framework supports both forwards analysis and backwards analysis, and each property annotation specifies its direction; forward analysis is the default. Forward analysis is useful for performing high-level emulations of program behavior. For example, an analysis for checking file accesses emulates the behavior of the program with respect to the state of the file streams. Backward analysis is useful for determining future behavior of a program. For example, we can use backward analysis to determine when an opened file will never be accessed.

During forward analysis, information propagates forward through the program just as it would during program execution. In particular, properties flow from the right-hand side of an assignment to the left-hand side, and then on to any uses that the left-hand def reaches. In backward analysis, information flows back through the program, from the left-hand side of assignments back to the right-hand side, and then back to the reaching definition.

4.5.1 Enum-like properties

The enum properties associate a named value with objects in the program. We refer to this named value as a *property value*. We can use property values for a number of different analysis tasks, such as defining the states of a state machine or categorizing objects that have special characteristics. Unlike the `enum` syntax in C, our enums can be hierarchical: a property value can contain other property values.

Syntax

enum-property-definition	→	[*direction*] *property-values* [*initial-value*]
direction	→	**`@forward`**
	|	**`@backward`**
initial-value	→	**`initially`** IDENTIFIER
property-values	→	{ *property-value-list* }
property-value-list	→	*property-value* [**`,`** *property-value-list*]
property-value	→	IDENTIFIER [*property-values*]

Figure 4.11: Grammar for defining enum properties.

Figure 4.11 shows the grammar for enum properties. Each property has a name, which is used to identify it in other annotations. The direction can be forward or backward, but it defaults to forward. The property values are organized as nested lists using curly braces. The optional initial value must be one of the defined property values.

Example

For our example matrix library, we use an enum property to capture several different matrix special forms: upper triangular, lower triangular, diagonal, and identity. Notice that there is a natural structure to this information. For example, the identity matrix is a special case

```
property MatrixForm : { Dense { Triangular {  Upper,
                                              Lower }
                                Diagonal { Identity } }
                        Zero }
              initially Dense
```

Figure 4.12: The MatrixForm property captures different matrix special forms.

of a diagonal matrix. We can use the nested structure of the enum property to express these relationships. Figure 4.12 shows the property definition for the matrix special forms.

Semantics

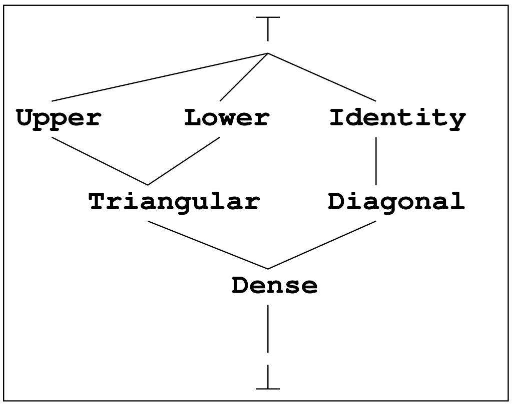

Figure 4.13: The lattice described by the MatrixForm property above.

Each property annotation specifies the flow value for a dataflow analysis problem. Dataflow analysis is a standard program analysis technique based on lattices that is employed by many compilers. The nested structure of the property values implies the underlying dataflow analysis lattice. The lattice organizes property values into a partially ordered set in which each property value is less-than its nested property values. Specifically, for each syntactic construct of the form `val { val1,...,valn }` we define the lattice such that $val < val_i$ for $1 < i < n$. Lattices are often shown in graph form, where each node is a lattice element and edges between the elements represent the less-than operation. We adopt the compiler view of lattice orientation that places larger elements higher in the graph and lesser elements lower in the graph. To these lattice elements we add the *top* and *bottom* elements, which are unique elements that represent the greatest and least elements respectively. Figure 4.13 shows the lattice described by the `MatrixForm` property in Figure 4.12.

The conservative nature of static analysis often requires the analysis framework to combine flow values, resulting in approximate information. For example, at a control-flow merge point (e.g., at the end of an if-else statement) the analysis framework may have to combine conflicting information about the two branches. Figure 4.14 shows a code fragment that can create either an upper triangular or a lower triangular matrix. The lattice structure helps to minimize the loss of information at merge points by providing meaningful approximations. We use the lattice *meet function* to choose the most specific information that still satisfies all the incoming information. This often allows the analyzer to avoid worst-case assumptions. In the example, we conclude that matrix `m` is at worst a triangular matrix of some sort.

```
if (some_condition)
  m = createUpperTriangularMatrix(rows, cols);
else
  m = createLowerTriangularMatrix(rows, cols);

// -- Merge point: what kind of matrix is m?
//       => at least triangular

printMatrix(m);
```

Figure 4.14: The nested structure of property values allows us to avoid worst-case assumptions when information is merged.

Our analysis framework associates property values from each property with objects in the input programs. For each object it collects two kinds of information: flow-sensitive and flow-insensitive. The flow-sensitive information records the property value of the object at each point in the program. The flow-insensitive information represents the aggregation of all the property values that the object ever assumes.

4.5.2 Set-like properties

The set properties allow the annotator to collect and organize groups of related objects. For example, in our matrix library we might want to collect the set of all matrices with a particular characteristic. We currently support two kinds of set-like properties:

- **Sets**: a set property provides a name for a group of objects. Later in this chapter we describe the annotations for adding and removing from the set and the annotations that test for an element in the set.

- **Equivalences**: an equivalence property maintains an equivalence relation over a set of objects. We can use annotations to declare that two objects are equivalent under the relation and to test for equivalence.

Like the enum properties, set-like properties need a mechanism for handling conservative analysis. For example, during analysis a set might contain different elements on two different paths through the program. At a merge point, the analysis framework needs to reconcile the conflicting information. We give the annotator two options for merging sets and equivalence relations: union and intersection. The choice depends on the meaning of the particular analysis, but in general, the union operation represents an optimistic property, while the intersection operation represents a pessimistic property.

Syntax

Figure 4.15 shows the grammar for the set-like property definitions. The definition simply selects the type of container, set or equivalence, and the merge operation. Currently, the analysis framework only supports forwards analysis for these properties.

$$
\begin{array}{lcl}
\textit{set-property-definition} & \rightarrow & \{\texttt{\textbf{union-set}}\} \\
 & | & \{\texttt{\textbf{intersect-set}}\} \\
 & | & \{\texttt{\textbf{union-equivalence}}\} \\
 & | & \{\texttt{\textbf{intersect-equivalence}}\}
\end{array}
$$

Figure 4.15: Grammar for defining set-like properties.

Example

In Figure 4.16 we define two example set-like properties for our matrix library: one set property that collects all the matrices allocated at any given point, and one equivalence property that records which groups of matrices are the same size. Notice the naming style for equivalence relation: we include the preposition "As" because the name is used as the operator when referring to the equivalence relation.

```
property AllMatrices : {union-set}

property SameSizeAs : {intersect-equivalence}
```

Figure 4.16: These properties oganize objects into semantically significant groups.

The `AllMatrices` property is a union-based set. We chose the union operation because we conservatively assume that a matrix exists if it is created on any path in the program. This choice reflects our goals for this particular analysis pass, and we could easily choose the intersection operator if we want to compute the set of matrices that exist on all paths. The `SameSizeAs` property is an equivalence relation, but it uses intersection to merge information. The reason for this choice is that if the size information for a matrix conflicts on two different paths, then we cannot safely assume any information about it. The code fragment in Figure 4.17 illustrates the two kinds of properties.

```
if (some_condition)
  m = createMatrix(rows, cols);

// -- Merge point 1: does m exist?
//       => to be safe assume it does

if (other_condition)
  m = copyMatrix(a);  // m is SameSizeAs a
else
  m = copyMatrix(b);  // m is SameSizeAs b

// -- Merge point 2: is m SameSizeAs a or b?
        => no, unless a and b are the same size
```

Figure 4.17: The set-like properties can use union or intersection to combine information, depending on their semantics.

Semantics

We implement the set-like properties in the analysis framework by introducing a special synthetic object that represents the state of the property. The compiler accesses and updates set properties through these special synthetic objects, which allows it to treat these properties uniformly throughout the analysis process. Therefore, no special code is needed to manage uses and defs, merge points, parameter passing, and analysis precision.

The flow-value for the set properties consists of a set of objects. At a merge point we compute the merged value by applying either union or intersection to the reaching sets, as specified in the annotations. The flow-value for the equivalence properties consists of a set of equivalence classes, with each class containing the set of objects that are equivalent. Adding a new equivalence relation can force the merging of classes, in the case of a union-based equivalence, or the splitting of classes.

4.5.3 Analysis annotations

The property annotations described above only specify the kinds of library-specific information to collect. The analysis annotations described here specify how library routines generate and manipulate this information. We express these effects as a set of rules, one set for each property and each library routine. Together these annotations specify complete analysis problems.

In our approach, each library routine describes its effects on the various properties of the objects it accesses. In many cases, these effects are unconditional. For example, if our matrix library contains a routine to create an identity matrix, then this routine unconditionally sets the `MatrixForm` property of the new matrix to `Identity`. However, we often need to express effects that depend on various conditions. For example, a routine for creating diagonal matrices could be used to create an identity matrix by passing 1.0 as the diagonal value. In this case, the `MatrixForm` property of the new matrix depends on this value. Therefore, our language supports both conditional and unconditional analysis rules.

analysis-rule-annotation	$\rightarrow$	**`analyze`** IDENTIFIER { *analysis-rule* $\star$ }
	$\mid$	**`analyze`** IDENTIFIER { *analysis-effect* $\star$ }
analysis-rule	$\rightarrow$	**`if (`** *condition* **`)`** { *analysis-effect* $\star$ }
	$\mid$	**`default`** { *analysis-effect* $\star$ }

Figure 4.18: Grammar for analysis rules.

Figure 4.18 shows the overall grammar for the analysis annotations. The annotation first specifies the property it affects, followed by a list of conditional or unconditional analysis rules. Each conditional rule specifies a boolean condition and a set of changes that take effect if the condition is true. The condition can test any available dataflow information, including flow values from other properties. However, the effects can only update the value of property specified in the *analysis-rule-annotation*. The unconditional rule just consists of a set of effects.

The analysis annotations, described below, support a rich and complex set of conditional tests. In practice, we find that most analysis rules are simple and many are unconditional. Instead, much of the complexity of these constructs is needed to support the conditions that guard the code transformations and error messages. Several examples are provided later in this chapter, as well as in Chapters 5 and 6.

The procedure annotation for each library routine may contain an instance of this annotation for each property. For our matrix library, each routine could contain two instances: one for the `MatrixForm` property and one for the `SameSizeAs` property. However, library routines that do not affect a particular property may omit the corresponding analysis annotation.

Conditions

Each conditional rule has a boolean expression that controls whether the rule takes effect. The condition consists of various atomic tests, which can be combined into more complex formulas using logical connectives. The language provides a variety of tests that access dataflow information, constant values, and pointer information. We use the same condition syntax to drive the error reporting and code transformation mechanisms described later in this chapter.

Figure 4.19 shows the overall grammar of a condition expression. The tests fall into four general categories:

- **Enum property tests**: This type of test accesses the enum property values of objects, allowing the annotations to test for a particular property value or to compare the property values of two objects.
- **Set property tests**: This type of test accesses the various set and equivalence properties.
- **Numeric tests**: This type of test accesses the constant propagation information computed by the analysis framework.
- **Binding tests**: This type of test accesses the pointer and binding information, allowing the annotations to check for conditions such as pointer aliasing and empty

bindings.

condition	$\rightarrow$	*test*
	|	*condition* **||** *condition*
	|	*condition* **&&** *condition*
	|	**!** *condition*
	|	**(** *condition* **)**
test	$\rightarrow$	*enum-property-test*
	|	*set-property-test*
	|	*numeric-test*
	|	*binding-test*

Figure 4.19: Grammar for conditional expressions.

Tests

The annotation tests are predicates on the objects in the annotations. The enum property tests allow the annotations to query the current property value of an object and compare it to a particular property value or to the property value of another object.

A property test consists of four components: (1) the particular property to test, (2) the way to compare the property values (the comparison operator), (2) the objects and values to test against (the operands), and (3) the kind of information to test. The property to test is given by the optional "IDENTIFIER `:`" syntax at the beginning of the test. When omitted, the property defaults to the one being analyzed. The operands can be either variables or specific property values. Figure 4.20 shows the grammar for enum property tests.

The left-hand side of the comparison specifies the object to test. In order to test an object, we need to retrieve its property value from the analyzer. The temporal operator allows the annotation to retrieve the property value at different points in the program, relative to the current call site. Our language supports four options:

- `@before`: This operator requests the current reaching property value of the object, before applying any rules. It is the default when the temporal operator is omitted.

enum-property-test	→	[IDENTIFIER **:**] IDENTIFIER *temporal-operator* **`is-??`**
	|	[IDENTIFIER **:**] IDENTIFIER *temporal-operator* *enum-property-operator* IDENTIFIER
temporal-operator	→	**`@before`**
	|	**`@after`**
	|	**`@always`**
	|	**`@ever`**
enum-property-operator	→	**`is-exactly`**
	|	**`is-atleast`**
	|	**`could-be`**
	|	**`is-atmost`**

Figure 4.20: Grammar for testing various dataflow facts.

- `@after`: This operator requests the property value of the object after the analysis rules have been applied. It is not allowed in analysis rules, but is useful for reports and code transformations.

- `@ever`: This operator requests the set of all the property values ever assumed by the object.

- `@always`: This operator produces a single value by combining all the property values ever assumed by the object.

The right-hand side of the comparison is either a specific property value or another object. When the right-hand side is an object, we retrieve its value using the same temporal operator as the left-hand side.

We provide four operators for comparing property values. Two of these operators, `is-atleast` and `is-atmost`, compare the values using the structure of the underlying lattice.

- `is-exactly`: This comparison returns true only when the left and right sides have the exact same property value.

- `is-atleast`: This comparison returns true when the left-hand property value is at least a sub-category of the right-hand property value. For example, the expression `Upper is-atleast Triangular` is true. This operation corresponds to the lattice greater-than-or-equal operation.

- `is-atmost`: This comparison is the natural opposite of `is-atleast`. It returns true if the left-hand property value represents a category that contains the right-hand value. For example, `Diagonal is-atmost Identity` is true. This operation corresponds to the lattice less-than-or-equal operation.

- `could-be`: This comparison tests to see if the object on the left could have a particular property value, even if the information is obscured by conservative analysis. We implement this operator by keeping a set of possible property values.

Notice that `is-atleast` and `is-atmost` are not logical opposites because a lattice is only a partial order, not a total order. Therefore, a property can contain property values for which neither comparison is true. For example, `Upper` and `Lower` triangular matrices are not comparable.

```
if (some_condition)
  m = createUpperTriangularMatrix(rows, cols);
else
  m = createIdentityMatrix(rows, cols);

// -- Merge point: what kind of matrix is m?
//      => lattice meet function returns Dense
//            m is-atleast Lower is true
//      => could-be Upper or Identity
//            m could-be Lower is false
```

Figure 4.21: The `could-be` operator preserves specific property values across merge points.

The `could-be` comparison is most useful for error detection problems because it distinguishes the possible property values without combining them. Figure 4.21 shows a code fragment that demonstrates this feature. The if-else statement constructs either an upper triangular matrix or an identity matrix. After the merge point, the meet function combines the two values `Upper` and `Identity` to produce `Dense`. The `is-atleast` operator applied to `Dense` returns true for all other values, even ones that the matrix never assumes. The `could-be` operator does not merge the information, which allows the annotations to determine that `Lower` is not a possibility.

Figure 4.22 shows the grammar for testing the set-like properties. The first case covers the equivalence relations, and we use the name of the relation as the binary operator. The test evaluates to true if the object on the left and the object on the right are equivalent under the given relation. There are two tests for the set properties. The first test evaluates to true if the object on the right is a member of the set on the left. The second test evaluates to true if the set if empty.

set-property-test	→	IDENTIFIER IDENTIFIER IDENTIFIER
	|	IDENTIFIER **`is-element-of`** IDENTIFIER
	|	IDENTIFIER **`is-`**{}

Figure 4.22: Grammar for testing set properties, such as set membership or equivalence.

Library routines often have special constant values that change their behavior. For example, a matrix multiply routine might have a boolean argument that indicates whether or not it should transpose the input matrices. To accommodate these conditions we provide tests for constant values. Figure 4.23 shows the grammar for these tests. The syntax includes the common numeric comparisons as well as a complete set of computational operators.

<table>
<tr><td>numeric-test</td><td>→</td><td>num-compare-expression</td></tr>
<tr><td></td><td>|</td><td>IDENTIFIER is-#</td></tr>
<tr><td>num-compare-expression</td><td>→</td><td>num-expression == num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression != num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression < num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression <= num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression > num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression >= num-expression</td></tr>
<tr><td>num-expression</td><td>→</td><td>CONSTANT</td></tr>
<tr><td></td><td>|</td><td>IDENTIFIER</td></tr>
<tr><td></td><td>|</td><td>(num-expression)</td></tr>
<tr><td></td><td>|</td><td>num-expression + num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression - num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression | num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression ^ num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression & num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression * num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression / num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression % num-expression</td></tr>
<tr><td></td><td>|</td><td>- num-expression</td></tr>
<tr><td></td><td>|</td><td>+ num-expression</td></tr>
<tr><td></td><td>|</td><td>num-expression</td></tr>
</table>

Figure 4.23: Our language supports a complete set of numerical operators.

Occasionally, the behavior of a routine depends on the relationships between the sets of inputs. For example, a general matrix multiply routine may require that all three matrices be distinct (e.g., it cannot compute $C \leftarrow A * C$). We allow the annotations to directly test properties of the bindings. Figure 4.24 shows the grammar for the binding tests. We offer three binding operators:

- `is-aliasof`: Returns true if there is any overlap between the bindings of the two variables.
- `is-sameas`: Returns true if the two variables have exactly the same bindings.
- `is-empty`: Returns true if the binding contains no objects.

binding-test	$\rightarrow$	IDENTIFIER **`is-aliasof`** IDENTIFIER
	|	IDENTIFIER **`is-sameas`** IDENTIFIER
	|	IDENTIFIER **`is-empty`**

Figure 4.24: Grammar for testing object bindings, such as aliases.

Effects

Each analysis rule specifies the effects to apply when the condition is true. These effects update the analysis information, which propagates to later parts of the program. For enum properties and constant values, the annotations can specify a new value for an object or assign it the value of some other object. For the set properties, we can add or remove elements from the set. For the equivalence relation properties, we can declare two variables equivalent. Figure 4.25 shows the grammar for updating the various kinds of analyses.

The effects of an analysis annotation can only update the property specified in that annotation. Therefore, in order to allow the annotations to update constant values we define a special property called "constants" that refers to this information. The syntax and

analysis-effect	→	*numeric-assignment*
	|	*enum-property-assignment*
	|	*set-propert-operation*
numeric-assignment	→	IDENTIFIER **=** *num-expression*
enum-property-assignment	→	IDENTIFIER **<-** IDENTIFIER
	|	IDENTIFIER **<-+** IDENTIFIER
set-propert-operation	→	**`add`** IDENTIFIER
	→	**`remove`** IDENTIFIER
	|	IDENTIFIER IDENTIFIER IDENTIFIER

Figure 4.25: Grammar for specifying analysis effects: these constructs update the dataflow information.

semantics of the analysis rules for the constants property are exactly the same as the other properties.

4.5.4 Example

With the annotations described above, we can define the effects of several of our matrix library routines on the two properties, `MatrixForm` and `SameSizeAs`. Figure 4.26 shows three matrix creation routines; we omit the `access` and `modify` annotations for clarity. The `createMatrix` routine creates a new dense matrix. The analysis annotation unconditionally sets the matrix form to `Dense`. The `createDiagonalMatrix` creates a diagonal matrix, setting the value of the diagonal elements to the input `diag`. The analysis rule indicates that when this value is 1.0 the resulting matrix is actually an identity matrix; otherwise it is just diagonal. Finally, the `copyMatrix` routine creates a copy of the input matrix. The analysis annotations capture two important properties of a copy: first, that it has the same form as the original, and second that it is the same size.

```
procedure createMatrix(rows, cols)
{
  analyze MatrixForm { matrix_data <- Dense }

  on_exit { return --> new matrix { rows,
                                    cols,
                                    data --> new matrix_data }}
}

procedure createDiagonalMatrix(diag, rows, cols)
{
  analyze MatrixForm {
    if (diag == 1.0) {
      matrix_data <- Identity
    }
    default {
      matrix_data <- Diagonal
    }
  }

  on_exit { return --> new matrix { rows,
                                    cols,
                                    data --> new matrix_data }}
}

procedure copyMatrix(in)
{
  on_entry { in --> in_matrix { rows,
                                cols,
                                data --> in_data }}

  analyze MatrixForm { matrix_data <- in_data }

  analyze SameSizeAs { matrix_data SameSizeAs in_data }

  on_exit { return --> new matrix { rows,
                                    cols,
                                    data --> new matrix_data }}
}
```

Figure 4.26: Example annotations for creating matrices.

Figure 4.27 shows an example using the special constants property. The size of a matrix is the product of the row and column sizes.

```
procedure sizeMatrix(in)
{
  on_entry { in --> in_matrix { rows,
                                cols,
                                data --> in_data }}
  analyze constants {
    return = in_matrix.rows * in_matrix.cols
  }
}
```

Figure 4.27: Example annotations that use the `constants` property to update constant propagation information.

4.5.5 Semantics

Our analysis framework interprets property annotations as flow values and interprets analysis rules as transfer functions. The analyzer uses this information when it encounters a call to an annotated library routine in the application program. At each call site, it uses the procedure annotations to perform the following steps:

1. Establishes the variable bindings. The analyzer starts with the bindings between the formal parameters and the actual parameters at the call site and uses the `on_entry` and `on_exit` annotations to establish bindings to the internal objects.

2. Evaluate rule conditions. The analyzer first evaluates all the condition rules for all the analysis annotations. By evaluating them all simultaneously we prevent the effects of one rule from interfering with condition on another rule.

3. Select a rule to trigger. For each property, the analyzer chooses a rule to apply. It is possible for more than one condition to evaluate to true. In this case the analyzer chooses the first one in the list that is true. This means that the order of rules within

an analysis annotation is significant. If none of the conditions is true then it selects the default rule. For unconditional rules there is no choice.

4. Apply the rule effects. The assignments are performed simultaneously by evaluating all the right-hand sides first and then applying them to the left-hand side objects. This prevents interference between effects in the same rule.

For each object and each property, the analyzer maintains three kinds of information in the mapping. The effects of the analysis rules update all three kinds of information at the same time:

***Now* information:** This information records the property value of an object at each point in the program. In compiler terms, this information is flow-sensitive "may" information. We use this information to compute the `@before` and `@after` values.

***Always* information:** This information summarizes the possible property values of an object over the whole program. It is computed by merging all the property values that are assigned to the object. In compiler terms, this information is flow-insensitive "must" information.

***Ever* information:** This information collects the possible property values of an object over the whole program without merging them. It consists of a list of all the values that the object ever assumes. The *Always* information is equivalent to merging the values on this list.

Because our pointer analysis is a "may" analysis, it is possible for a single variable in the annotations to have multiple actual objects in its bindings. Therefore, when computing the property value for such a variable the analyzer first combines the information from the different objects before applying any property value comparison operators.

The current implementation divides the analysis problems into two groups: forward analyses and backward analyses. It performs the forward analysis pass first, which includes computing all the pointer analysis information and constant propagation information along with all the forward analysis properties. It then performs a second pass that handles all the backward properties. This scheme allows backward properties to refer to information in the forward properties, but not the other way around.

4.6 Report annotations

Once the dataflow analysis phase is complete, we can query the results and print the information. A report annotation serves this purpose. It can be used to print library-specific errors or warnings, or to display informational messages about library usage in the application. We use these annotations in Chapter 6 to detect library-level errors and security vulnerabilities.

4.6.1 Syntax

A report annotation consists of an optional condition, followed by a sequence of report query elements. The conditions use the same syntax and semantics as the analysis rule conditions, except that all four temporal operators are allowed. Each report query element specifies a piece of information to print out. The simplest element is just a literal string. Figure 4.28 shows the grammar for the report annotation. The syntax allows the annotator to distinguish error reports from other kinds of messages. This information drives the client-driven pointer analysis algorithm, which we describe in Chapter 7.

If the condition evaluates to true (or if the condition is omitted), then the compiler goes through the list of report query elements and generates a string for each. It concatenates the result and prints it to the screen.

We support several kinds of report query elements:

String literal: The string literal is given in the form of a C string constant, and it supports all the escape characters (such as "\n" and "\t") that C supports. It is printed exactly

report-annotation	→	**`report`** *report-element-list* **`;`**
	|	**`report if (`** *condition* **`)`** *report-element-list* **`;`**
	|	**`error`** *report-element-list* **`;`**
	|	**`error if (`** *condition* **`)`** *report-element-list* **`;`**
report-element-list	→	*report-element* [**`++`** *report-element-list*]
report-element	→	STRING
	|	IDENTIFIER **`:`** IDENTIFIER *temporal-operator*
	|	*num-expression temporal-operator*
	|	*program-location*
	|	**`[`** IDENTIFIER **`]`**
program-location	→	**`@callsite`**
	|	**`@context`**

Figure 4.28: We use report annotations to generate library-specific errors message or other library-specific informative messages.

as specified.

Property query: Like property value tests, this element determines the property value of a variable by querying the dataflow analyzer. In this context, the name of the property to query is mandatory. The optional temporal operator defaults to `@before`.

Constant expression: This element evaluates the constant value of variable. These values may be combined into an expression using the normal computational operators. The only temporal operators allowed in this context are `@before` and `@after`.

Location query: The two special tokens `@callsite` and `@context` generate information about the location of the library call in the application program. The `@callsite` token is replaced with the source file and line number of the current library call. The `@context` token is replaced by a full, context-sensitive path through the program.

Bindings query: Often it is useful to print out the names of the objects that are actually bound to the variables in the annotations. For example, we may want to know the name of the actual parameter to a library call. The square brackets take the name of a variable and print out the bindings of that variable.

4.6.2 Example

In our matrix library we can use reports to make sure that the special-purpose routines are used correctly. For example, we can check that the input to a triangular matrix multiple is in fact a triangular matrix. Figure 4.29 shows the annotations for the triangular matrix multiple routine, including the error report. Figure 4.30 shows an example of the output, with actual values plugged in for the report elements.

```
procedure multiplyUpperTriangularMatrix(A, B, C)
{
  on_entry { A --> matrixA { rows,
                             cols,
                             data --> dataA }
             B --> matrixB { rows,
                             cols,
                             data --> dataB }
             C --> matrixC { rows,
                             cols,
                             data --> dataC } }

  error if (( ! MatrixForm : dataA is-exactly Upper) ||
            ( ! MatrixForm : dataB is-exactly Upper))
   "Error␣at␣" ++ @callsite ++ ":␣" ++
      "matrix␣" ++ [ A ] ++ "␣is␣" ++
      MatrixForm : dataA ++ "\n" ++
      "matrix␣" ++ [ B ] ++ "␣is␣" ++
      MatrixForm : dataB ++ "\n";
}
```

Figure 4.29: This report checks to make sure that the input matrices are triangular.

```
Error at myfile.c:67: matrix Mat1 is Upper
      matrix Mat2 is Dense
```

Figure 4.30: An example of the report output: one of the input matrices is triangular, but the other is dense.

4.6.3 Semantics

The compiler evaluates each report annotation separately at each call site. In addition, the precision policy used during analysis affects the way that the compiler evaluates report annotations. Chapter 7 addresses the management of precision policies, but in this discussion we just describe how different levels of precision affect the reports.

In the context-insensitive case, the compiler will emit one report for each static callsite in the text of a program. For example, if a program contains a function that makes one call to the `createMatrix` shown in Figure 4.29, then the compiler will emit at most one error message regardless of how many times that function is called.

In the context-sensitive case, the compiler can emit a message for every calling context that reaches the library call. The number of possible messages depends on the number of paths through the call graph of a program to each library routine callsite. In some cases, the number of paths can become quite large. Nevertheless, the extra information provided by having the full path through the program often helps the programmer to pinpoint and fix errors.

We can see the difference between these two analysis modes in Figure 4.31, which shows an example of a program call graph. In context-insensitive analysis, the compiler can emit one message for the call to `createMatrix`. In context-sensitive analysis, the compiler can emit two messages: one for the call through `func1()` and one for the call through `func2()`.

4.7 Code transformation annotations

Each library routine can specify code transformation annotations that direct the compiler to replace calls to the routine with other code. Like the report annotations, these annotations can query the analysis results to decide whether to apply the transformation. However, unlike the reports, we require the condition to be true in all calling contexts in order to

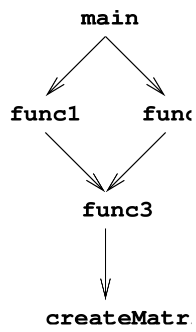

Figure 4.31: This example call graph contains one static call to `createMatrix` with two possible calling contexts, one through `func1` and one through `func2`.

trigger the transformation. The reason for this requirement is that code transformations change the text of the program, and thus they are visible to all the callers.

Our annotations support two kinds of code transformations: (1) inline the library routine by replacing the call with the source of the routine, or (2) replace the library call with an arbitrary code fragment. Figure 4.32 shows the overall grammar for the code transformations.

action-annotation	→	**when (** *condition* **) replace-with** %{
		C CODE
		%}
	|	**when (** *condition* **) inline ;**

Figure 4.32: Overall grammar for specifying code tranformations.

4.7.1 Code replacements

The `replace-with` transformation directs the compiler to replace a library call with the specified code fragment. Syntactically, the code fragment is treated as a list of C statements, and we use the C-Breeze front-end to parse it in. The advantage of using the C parser is that we can check the code fragments at compile time to help prevent the annotations from introducing syntax errors into target programs.

When specifying a replacement code fragment, we often need to refer to information from the call site, such as the actual parameters passed to the library call. The language augments the standard C syntax with special tokens that represent this information. The replacement code can refer to variables from the callsite using `$` followed by the local variable name. Since a library call can occur on the right-hand side of an assignment, we reserve the token `$return` to refer to the left-hand side.

Figure 4.33 shows an example for replacing a call to the math library power function with a multiplication when the exponent is known. Figure 4.34 shows the effect of this transformation on an example library call.

```
procedure pow(value, exponent)
{
  when (exponent == 2.0)
    replace-with %{ $return = $value * $value }%
}
```

Figure 4.33: Replace a call to `pow()` with multiplication when the exponent is known.

```
y = pow(x, 2.0);

     =>

y = x * x;
```

Figure 4.34: An example of the `pow()` transformation specified above.

4.7.2 Library routine inlining

The `inline` transformation replaces a library call with the source for that routine. We implement this inlining process as a source-level transformation by copying the library source into the caller and then fixing the parameters and return statement.

Our support for inlining raises an issue about protecting source code, because it requires the library annotator to make the library source available. This does not present any problems for open-source libraries. However, it may not be appropriate in other cases. One solution is for the library annotator to provide only those parts of the library source that would benefit from inlining. Another possible solution would be to provide library source in an internal format that only the compiler can read.

4.7.3 Semantics

The compiler applies code replacements using a process similar to hygienic macro expansion [62]. Internally, we represent each code fragment as an abstract syntax tree, which ensures that the fragment is syntactically correct. We also use the compiler's symbol table facilities to look up the identifiers in a code fragment. With the symbol table we can make sure that variables introduced in the code fragment are not captured by local variables at the call site.

At each library routine call site the compiler performs the following steps:

1. **Test conditions.** The compiler evaluates all the conditions that guard the code transformations. It evaluates each condition separately in each calling context, but the transformation is only enabled if the condition is true in all contexts.

2. **Select a transformation.** The step above can enable more than one code transformation at a given call site because more than one condition might evaluate to true. Since we can only apply one transformation, we use the order of the annotations to select it: the compiler chooses the first code transformation annotation with a true condition. This policy provides a convenient way to define an ordering on transformations that starts with cases that are more specific and falls through to cases that are more general.

3. **Generate replacement code.** With a call site and a transformation selected, the compiler can generate a customized code fragment to replace the call. In the case

of `replace-with` annotations, it copies the code fragment and then replaces the meta-tokens with information from the call site. In the case of `inline` annotations, it copies the body of the inlined procedure and then adds code to assign the actual arguments from the call site to the formal parameters.

4. **Replace the library call.** The compiler then removes the library routine call and replaces it with the new code.

5. **Name mangling.** The names of variables in the newly generated replacement code might conflict with the names of variables in the caller. This situation is common when inlining a library routine because the caller often uses the same name for the actual arguments to the call as the library implementation uses for the formal parameter. Since we maintain symbol information for each code fragment, name conflicts do not interfere with correct program analysis. When the compiler outputs the transformed code, however, it applies a name mangling algorithm to eliminate conflicts.

Once the compiler has performed all the applicable code transformations, it post-processes the resulting code to return it to the canonical intermediate representation. It breaks down any complex operators and control flow, and it rebuilds the control-flow graph. The code is then ready for another analysis and transformation pass.

4.8 Globals

In addition to describing library procedures, the annotation language provides a way to define global variables and data structures. Globals are useful for representing abstract global state information, modeling internal static data structures, or representing input/output devices. The global variable annotations allow the annotator to define new global variables, to connect them together into data structures, and to initialize the dataflow properties associated with them.

Unlike local variables declared inside the `procedure` annotations, global variables do not represent bindings to actual objects in the input programs. Instead, they introduce objects into the program representation so that other annotations can use them. Therefore, the annotations should not use this facility to declare global variables that are already defined by the library header file. The annotations can refer to global variables from the header file without using the global annotations.

Figure 4.35 shows the grammar for defining the global variables. The syntax and semantics for defining pointer structures are the same as the `on_exit` annotations shown in Figure 4.8, with two exceptions. First, when defining global variables the conditional form is not allowed. Second, in this context we allow the `I/O` designation on a variable, which tells the compiler to treat all updates to the variable as side-effects. This option is useful for modeling library routines that access input/output devices, such as writing to the disk or sending a message, which would otherwise appear to have no effect.

global	$\rightarrow$	**`global`** { *pointer-structure* $\star$ }
	|	*analysis-rule-annotation*

Figure 4.35: The annotation language provides a way to define global variables and structures, and to initialize their property values.

In addition to defining global variables, we can use the annotation language to set their initial state. Figure 4.35 includes the grammar for the analysis rule annotations. The syntax and semantics are the same in the global context, except that we disallow temporal operators and conditions.

4.8.1 Example

We can define several useful global variables for our fictional matrix library. Many real libraries have special routines to initialize and finalize the library, and all other library routines must occur between these two calls. We can check this requirement by defining a global variable that records the state. Figure 4.36 shows the annotations for the global, along with the annotations for the initialize and finalize routines.

```
property LibraryState : { Initialized, NotInitialized }

global { is_initialized }

analyze LibraryState {
  is_initialized <- NotInitialized
}

procedure initializeMatrixLibrary()
{
  analyze LibraryState {
    is_initialized <- Initialized
  }
}

procedure finalizeMatrixLibrary()
{
  analyze LibraryState {
    is_initialized <- NotInitialized
  }
}
```

Figure 4.36: We can define a global variable to keep track of whether or not the library has been properly initialize.

We can use global variables to represent the initial state of global data structures. For example, our matrix library might include a global zero matrix that the library uses to initialize other matrices. Figure 4.37 shows the annotations to declare the matrix and initialize it to the zero form.

4.8.2 Semantics

Our compiler represents global variables by creating synthetic objects and adding them to the memory model. The compiler applies the initial pointer structure and analysis values to these variables by associating them with the entry point of the `main` function in a target program.

```
global { AllZeros --> new Zero_matrix { rows,
                                        cols,
                                        data --> new Zero_data } }

analyze MatrixForm {
  Zero_data <- Zero
}
```

Figure 4.37: The `AllZeros` global variable represents an internal matrix that the library initialized with all zeros.

4.9 Discussion and future work

In this chapter we have described our annotation language for supporting library-level compilation. Anecdotal evidence suggests that the language is usable and intuitive: several Broadway users, with little compiler experience, have been able to quickly develop useful annotations. In subsequent chapters, our experiments show it is effective for expressing a variety of library-level analyses and optimizations. The syntax and semantics of this language are the result of a considerable amount of design and experimentation. Nevertheless, we have identified a number of future enhancements and extensions that would further improve the language.

- **Encapsulating common annotations.** We find that the annotations for large libraries often contain repetitive patterns. For example, the `on_entry` and `on_exit` annotations are essentially the same for every routine that accepts a particular data structure. Similarly, there are often multiple routines that have the same analysis behavior. The addition of reusables to the language would simplify the annotations and help make them more maintainable.

- **Generating annotations.** Developing the annotations for a library is a considerable amount of work, especially at the beginning of the process. Collecting the list of routines and determining the data structures is tedious and time consuming. We could speed up this task, and at the same time help encourage correctness, by providing a tool to automatically generate an initial skeleton of the annotation file. Such a tool could even perform some analysis of the library source code to produce candidates for the basic dependence annotations.

- **Annotation checking.** Like any other form of code or specification, annotations can contain errors. The current implementation of the Broadway compiler only performs basic syntactic checking on annotation files. A more helpful system would check for a variety of possible semantic problems in the annotations, such as inconsistent pointer structures and potential infinite loops in the code transformations. In particular, there are a number of tests we could perform on the library-specific dataflow analyses, such as ensuring that analysis annotations cover all cases.

- **Annotation debugging.** Developing annotations also involves some debugging and experimentation. The current compiler provides some minimal feedback during normal compilation, and it includes a verbose mode that emits extremely detailed information about the compilation process. Ideally, the compiler would include an annotation debugging mode, in which the annotator could step through the analysis and optimization passes.

- **Language extensions.** Since the current language is quite simple, we have identified many possible ways to extend its functionality. We present some of these possibilities in subsequent chapters where specific problems motivate the need for additional capabilities. We expect the language to evolve slowly, however, because we want to prevent it from becoming overly complex.

Chapter 5

Library-level Optimization

Various techniques have been used to improve the information available to optimizing compilers. Existing approaches include widening the scope of program analysis, exploiting runtime information from profiling, and performing dynamic compilation. An alternative approach, which has received little attention, is to provide information that describes higher levels of abstraction and that could raise the level at which compilers optimize programs. In this chapter, we demonstrate the benefits of this approach by using our library-level compiler to apply high-level optimizations to several performance-critical applications. We show that even though the applications and supporting libraries are already highly tuned, our compiler still improves their performance because it finds optimization opportunities that traditional compilers cannot.

5.1 Introduction

High performance computing, and scientific computing in particular, relies heavily on software libraries. Libraries are attractive because they provide an easy mechanism for reusing code. Moreover, each library typically encapsulates a particular domain of expertise, such as graphics or linear algebra, and the use of such libraries allows programmers to think at a higher level of abstraction. Unlike the built-in abstractions of a programming language, however, library abstractions are accessed entirely through one syntactic construct: the procedure call. This procedural interface is significant because it provides a uniform and familiar syntax for accessing all the varied and complex capabilities offered in library form. Unfortunately, traditional compilers do not attribute any special meaning to library interfaces. With few exceptions, compilers treat each invocation of a library routine the same as any other procedure call. Thus, many optimization opportunities are lost because compilers ignore the semantics of libraries.

In Chapter 4 we described a fictional example of a high-level optimization: we can replace a general matrix multiply routine with a more efficient specialized routine when the input matrices are triangular. In our approach, we use the annotation language to make concepts like "triangular" explicit in the compiler. However, it is natural to ask whether this information is necessary: can traditional compiler algorithms achieve the same result? To answer this question, consider the implementation of a general matrix multiplication routine, which might consist of three nested loops. An aggressive optimizer might perform a number of sophisticated optimizations, including loop interchange, loop unrolling and loop tiling, to improve performance. Unfortunately, this approach has a fundamental limitation: the best it can do is find a more efficient way to perform all the computations required for a dense matrix multiplication. It has no way to recognize that half of the computations are unnecessary—that fact is a property of the domain of matrix computations, not of the underlying programming language.

The problem with optimizing compilers is that they have a limited ability to understand what programs do. Given a set of primitive operators, a compiler can only extract a limited amount of information about computations that are composed from those operators. The traditional approach to improving optimization is to develop increasingly sophisticated analysis techniques that try to derive more information from the same low-level operators. Our approach acknowledges the limits of automatic program understanding. We give the compiler new operators, in the form of library routines, which provide a higher level starting point for program analysis and optimization.

We show that library-level optimizations exploit rich and diverse types of semantic information, but that these optimizations share a small number of central ideas when considered from the compiler's point of view. Our key insight is to combine powerful library-specific dataflow analysis with simple pattern-based code transformations. The complementary nature of these two tools allows each to be relatively simple, while still enabling powerful optimizations.

Our approach has a number of benefits, not just for program performance but also for the usability of high-performance libraries.

- Library-level optimization overcomes the inherent limitation of traditional compilers that only recognize and optimize the built-in operators of their programming languages. The annotation language effectively extends the semantics of a programming language to include domain-specific concepts from the library.

- Library-level optimization overcomes several limitations of software libraries. First, libraries are typically implemented, compiled, and optimized in isolation from the programs that use them. Rather than view a library as a fixed entity, our compiler uses the annotations to customize the library for each application. Second, libraries often expose performance issues in their interfaces by providing many specialized routines that offer better performance for particular circumstances. Without compiler support, the programmer is responsible for choosing the appropriate routine and using it in the correct manner.

- The annotations capture and codify library expertise that is often lost or is only expressed in an informal manner, such as a user manual. The annotations not only free the programmer from having to acquire this expertise, they allow the compiler to apply it automatically. The automation ensures that optimizations are applied correctly and comprehensively. Although developing the annotations can be a substantial task, this effort is amortized over all the programs that use the library.

- Our compiler architecture provides an important conceptual benefit: a clean separation of concerns. The compiler encapsulates all compiler analysis and optimization machinery, while the annotations describe all library knowledge and domain expertise. Together, the annotations and compiler free the programmer to focus on application design rather than on performing manual library-level optimizations.

5.1.1 Overview

In this chapter we show how to use the Broadway compiler for library-level optimization. We first describe our overall strategy for supporting these optimizations and we show how the annotation language can express the analyses and code transformations. We demonstrate these capabilities on programs written using the PLAPACK parallel linear algebra library. We find that library-level optimization can substantially improve performance, even when starting with a highly tuned library implementation.

This chapter makes the following contributions:

- We use our annotation language to specify library-level optimizations. We show that our approach effectively captures the expertise necessary to improve performance beyond the capabilities of traditional optimization compilers.

- We describe the compiler mechanisms needed to exploit library-level optimizations, including the specific compiler passes and our overall compilation strategy.

- We present fully automated results for several real programs written using a production-quality high-performance library. Our compiler can produce code that approaches the performance of hand-coded implementations written by the library authors.

5.2 Optimization opportunities and strategies

The goal of library-level optimization is to improve performance by exploiting the domain-specific semantics of libraries. Libraries represent a wide range of domains, and different domains often present considerably different optimization opportunities. Our annotation language, presented in Chapter 4, handles the diversity of libraries by providing access to flexible program analysis and transformation capabilities. In order to be widely applicable, however, our compiler needs to support a general optimization strategy that does not serve particular domains in an ad hoc manner. In this section we develop such a strategy by examining the optimization opportunities that libraries have in common, despite representing different domains.

5.2.1 Opportunities

In order to motivate our strategy of library-level optimization we need to understand how these optimization opportunities arise. In the discussion above we argue that library interfaces often have specialized routines that offer improved performance in certain circumstances. We also argue that library calls can benefit from traditional optimizations, such as dead-code elimination and constant propagation. It is natural to ask, if a better routine exists why the programmer wouldn't use it in the first place? Or if code is dead, why the programmer wouldn't eliminate it? Part of the answer, which we present above, is that the specialized routines are difficult to use and require some level of expertise.

More significantly, many optimization opportunities arise from the modularity and layering that programmers use to manage the complexity of software development. Layering helps decompose systems into manageable pieces, and it creates reusable modules [30]. In particular, developers of large scientific software systems have come to depend more and more on layering, since such systems tend to represent a broad spectrum of specialized programming domains. For example, the POOMA framework consists of five layers [73], with the higher layers representing abstractions in the problem domain, such as solvers and complete simulations, and the lower layers representing abstractions in the implementation, such as communication, data distribution, and sequential kernels. Many other systems are similarly layered [15].

One well-known problem with layers is the performance degradation that comes from not optimizing operations in an end-to-end manner. These issues have been addressed in certain domains, typically by leveraging domain-specific information about the layers. For example, systems have been introduced to optimize layered communication protocols [2] and toolkits for distributed computing [48]. There have been few attempts, however, to optimize layered scientific software, perhaps because such systems span such a broad range of domains. Instead, the thrust of such systems has been to optimize each layer or component independently. This situation provides an answer to our question above: a programmer cannot always choose the best library calls because the layered structure obscures the full context of their use.

Therefore, many performance improvements are made possible by breaking open one layer to expose the layer beneath it. Once in the context of the layer above or the application code, this lower layer often yields opportunities that were not visible at the higher layer. As a simple example, imagine a program fragment that contains a series of library calls on the same set of objects. Internally, each call may perform a set of checks to make sure the arguments are valid. In the context of the program fragment, these checks may be redundant or unnecessary. If we can expose the implementation, then our compiler can identify the redundant checks and remove them. Notice, however, that the compiler will often need domain-specific information in order to identify which checks are redundant and which are not.

5.2.2 Compilation strategy

Our approach to compiling layered systems balances two competing sources of performance improvement: (1) exploiting layer boundaries to facilitate high-level analysis and

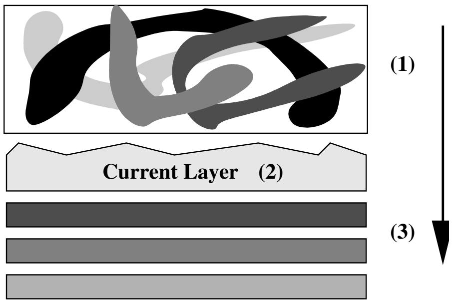

Figure 5.1: Compilation proceeds top-down: The upper layers **(1)** are integrated first to maximize the use of high-level information. The current layer **(2)** is compiled in the context of this integrated code to maximize specialization, before exposing the implementation of the lower layers **(3)**.

optimization, and (2) systematically dissolving layer boundaries to expose new optimization opportunities. Satisfying the first goal produces more effective use of the abstractions within each layer, while satisfying the second encourages safe and effective specialization across layers. Therefore, the overall compilation strategy, shown graphically in Figure 5.1, is a top-down approach: We apply all the possible optimizations at one layer before proceeding to the next. This preserves the high-level information as long as possible, while still exposing each layer to as much program context as possible.

Our approach allows programmers to obtain the benefits of layering and encapsulation without paying the performance penalty. In fact, we view layering as an asset rather than a liability because layer boundaries make programming abstractions explicit, which allows us to annotate them for the compiler. In addition, our system provides an alternative to the difficult and error-prone process of hand-tuning high-performance software. The library annotations describe a systematic set of transformations to dissolve layer boundaries and to exploit contextual information to improve performance. Our program analysis framework ensures that these transformations are applied uniformly, thoroughly and safely.

5.2.3 Annotation strategy

Within a particular layer of abstraction, there are many different ways to use the annotation language for describing optimizations. While the capabilities of the language are quite general, we have identified two overall strategies for using them. These two strategies are not explicitly represented in the annotation language or the compiler. Rather, we present them as general principles to guide the development annotations. Later in this chapter we provide concrete examples of these strategies.

The first strategy is to define a set of annotations for automatically specializing general-purpose library routines. This strategy consists of two parts: first, we inline the general-purpose routines to expose their implementations to the caller, and then we apply a series of lower level optimizations that take advantage of information from the calling context. We use library-specific analysis to decide when to inline a routine so that inlining is only applied in profitable circumstances. The lower level analyses and optimizations work on the next layer down in the system, and they allow the compiler to perform a domain-specific partial evaluation of the inlined code. The advantage of this strategy is that with a suitable set of lower level optimizations the compiler can specialize any of the general-purpose routines for many different calling contexts. The disadvantage is that the quality of the resulting code depends on how well specialization works. In some cases, we find that the implementation of the general-purpose routine does not lend itself to specialization. In other cases, we find that the low-level optimizations are not powerful enough to produce the best result.

The second strategy is to define annotations that replace general-purpose library calls with more efficient special-purpose calls or with code fragments that capture library call idioms. This strategy overcomes some of the limitations of the first strategy by skipping directly to the final specialized code. The advantage of this approach is that it can encode special cases that are not easily derived from the general cases. The disadvantage of this strategy is that each optimization typically applies in only one or two situations. Therefore, writing a complete set of annotations may involve enumerating a large number of special cases. For example, if a particular library routine takes four arguments, and each argument can take on one of five different domain-specific states, then we may need to define 20 special cases.

In the experiments presented below, we focus primarily on the general-purpose specialization strategy, but the two strategies are complementary. In fact, the low-level optimizations of the first strategy typically consist of idiomatic specializations applied at a lower level of abstraction.

5.3 Optimizing PLAPACK

In this section we give concrete examples of the annotations to demonstrate the application of our technique to the PLAPACK [86] parallel linear algebra library. We first provide background about PLAPACK abstractions and their role in optimization. We present a layer decomposition of the PLAPACK system and describe the abstractions at each layer. We then describe individual optimizations and categorize them according to the layer to which they apply. Finally, we encode these optimizations using our annotation language.

In order to explain our technique in depth, we go into considerable detail about the target library and its abstractions, the mechanics of the optimizations, and their representation in the annotations. Before delving into these details, it is worth enumerating the important points of this section:

- In a complex domain like parallel linear algebra, there is a wide range of potential optimizations. We show that our annotations can capture many of these optimizations with a small number of language constructs.

- The complexity of the PLAPACK interface itself presents a significant challenge to applying the optimizations. The compiler mechanisms we provide help to overcome these difficulties.

- Most of these optimizations are valid only under particular conditions that are highly domain-specific. Without the configurable dataflow analyzer, the compiler cannot collect the necessary information.

In Section 5.5 we show the impact of these optimizations on the performance of several PLAPACK programs.

5.3.1 Concepts

PLAPACK is a library for coding parallel linear algebra algorithms in C. It consists of approximately 45,000 lines of C code and provides parallel versions of many of the same kernel routines found in the BLAS [32] and LAPACK [4]. At the highest level, it provides an interface that hides much of the parallelism from the programmer.

A PLAPACK application operates on linear algebra objects, such as matrices and vectors, that are partitioned and distributed over the processors of the target computer. The application manipulates these objects indirectly though handles called *views*. A view specifies an index range that selects some or all of a distributed object for subsequent computations. PLAPACK provides routines to create views, shift views, and split views into pieces. The following figure shows a split that logically divides a matrix into four smaller ones:

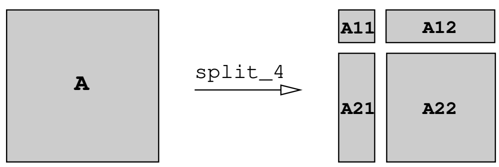

Figure 5.2: PLAPACK algorithms operate at a higher level than traditional linear algebra algorithms by splitting matrices into logical pieces, called *views* and operating on the pieces.

A typical algorithm starts with an entire object, like A, and splits it into manageable pieces. It computes directly on A_{11}, A_{12} and A_{21}, and then continues iteratively by splitting the large remaining piece, A_{22}, until the entire data set has been visited. Often, a view captures part of a matrix or vector that has special properties. Understanding and exploiting these properties can lead to significant performance improvements.

PLAPACK kernel routines, such as parallel matrix multiply, are implemented using a lower level set of routines that make data distribution and movement explicit. At this level, the library creates objects with special distribution properties and then uses the `PLA_Copy()` routine to transfer data between them.

For example, Figure 5.3 shows how to compute an outer product from a matrix column and a matrix row. The algorithm first creates two overlapping replicated panels, one for the rows and one for the columns. It then uses the `PLA_Copy()` routine to copy the data from the original panels into their replicated forms. The result is that each processor contains the right pieces of the panels to perform a local matrix multiply, which completes the computation.

5.3.2 Optimizations

We now describe the specific optimizations that are used to produce the results in Section 5.5. We categorize them according to the PLAPACK layer to which they apply. Figure 5.4 shows the three conceptual layers that make up the PLAPACK implementation. Each layer has its own programming abstractions, and thus its own optimizations. While these optimizations vary considerably in their semantics, we will show that our annotation language can effectively encode them.

This set of PLAPACK optimizations derives from a number of different sources. In some cases, we codify techniques suggested in documents written by the PLAPACK authors [7]. In other cases, we examine PLAPACK programs ourselves to determine possible performance improvements. When we discover a potential optimization, we determine the circumstance under which it applies and then formulate a program analysis pass to detect that circumstance.

Programming at the highest level is desirable because it provides the most powerful

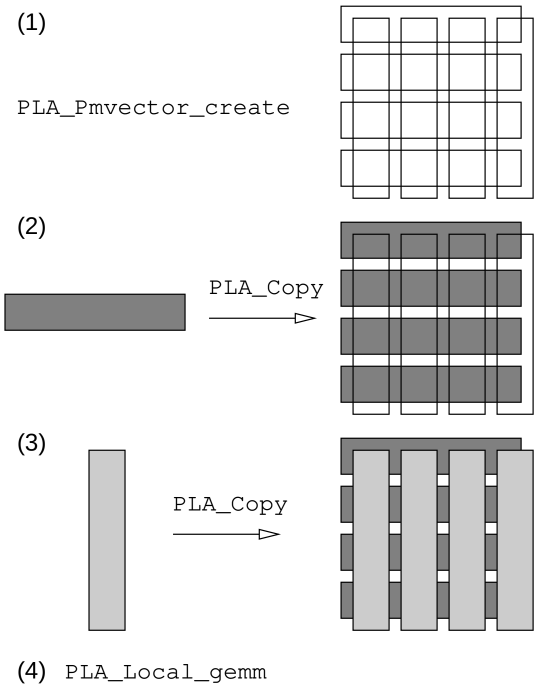

Figure 5.3: Algorithm to compute distributed outer-product using explicit data replication and local computation.

abstractions, and it hides the complexities of parallel programming. It also leverages a large body of reusable code underneath. However, by working at this high level, we miss many optimization opportunities that exploit explicit parallelism. Unfortunately, if we program a low level, we lose the high-level abstractions, and we have to deal directly with parallelism. As a result, we end up replicating much of the high-level code.

Global Layer: Parallel BLAS

The highest layer provides parallel linear algebra operations that completely hide the parallelism from the application developer. It consists of operations that work on any view, regardless of where the data resides. At this level, optimizations work in terms of the matrix domain. They exploit such notions as “triangular”, “diagonal” and “symmetric”, which

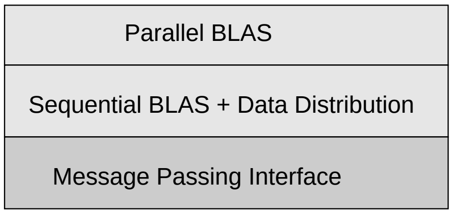

Figure 5.4: Logical layers of the PLAPACK implementation.

describe particular classes of matrices.

- **Scalar algebra.** The `PLA_Scal()` routine multiplies the elements of a matrix or vector by the given scalar constant. If that constant is known to be one, then the call has no effect and can be removed. If that constant is zero, we can replace the call with a special call that sets all elements to zero.

- **Matrix algebra.** Like the scalar algebra above, we can exploit the matrix multiplication identities. The `PLA_Gemm()` routine computes something of the form $C \leftarrow A * B + C$, so the optimizations are slightly different. If A or B are zero matrices, then the code has no effect and can be removed. However, if A or B is the identity matrix, then the call essentially computes a matrix addition. We can replace this call with code that explicitly adds the elements, which is an entirely local operation that requires no communication between processors.

- **View Query Simplification.** When properties of a view are known at compile time, we can replace queries of those properties with the known result. First, direct queries to the type of object can be replaced with static values. Second, for empty views, any size query can be replaced by zero. Third, certain query idioms can be replaced with static values. For example, if the global size of a view is the same as its local size, then it is a local view. If we discover this fact during compilation, we can avoid the test.

Middle Layer: Data Distribution

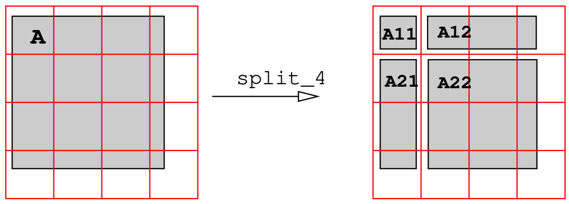

Figure 5.5: PLAPACK distributes matrix data across the processors. Split operations often result in special-case distributions, such as sub-matrices that reside entirely on one processor.

The middle layer hides some of the parallelism but makes the notion of data distribution and locality explicit. In Figure 5.5 we show the same four-way split as Figure 5.2 with an overlaid grid that represents the partitioning of the data over a grid of processors. The actual partitioning is more complex than a simple block distribution [33], but the basic observations still hold. Notice that that after splitting the matrix, certain pieces reside entirely on columns of processors, rows of processors, or on a single processor. In the figure above, the four-way split yields one *local* view (A_{11}), which resides entirely on one processor, one *column panel* (A_{21}), which resides on a column of processors, one *row panel* (A_{12}), which resides on a row of processors, and a fully distributed submatrix (A_{22}). We can take advantage of this information to improve performance. In particular, algorithms that are designed to process distributed matrices can often be significantly simplified when customized for row panels or column panels. This layer consists of sequential BLAS calls, which operate only on local pieces of data, and invocations of the `PLA_Copy()` routine, which move data around on the processor grid.

The most effective optimizations that we found result from breaking open the global layer routines to expose their middle layer implementations. The reason is that the global layer routines are designed to work with any kind of linear algebra object, regardless of its size and distribution. However, applications often pass particular special distributions into these routines, and we can take advantage of this extra information to create a customized version of the routine for that particular distribution.

Rather than enumerate all of these special cases, we define a set of optimizations that together can transform a general-purpose implementation into a customized version:

- **Special-case routine selection.** Internally, many PLAPACK routines have multiple implementations that work better in different situations. For example, the general matrix multiply routine, `PLA_Gemm()`, is implemented internally by three different algorithms. At run-time the routine chooses from among the algorithms by comparing the relative sizes of the input matrices. Often we can use library-specific analysis to identify these cases at compile time, thus avoiding the run-time cost.

- **View optimizations.** We can often simplify the PLAPACK splitting routines when the input view is already a special-case distribution. For example, there is no need to vertically split a column panel because it already resides on a single column of processors. Such optimizations can eliminate entire loops from the code.

- **Empty views.** Any computation on an empty view can be removed. The computational routines (for example, `PLA_Gemm()` and `PLA_Trsm()`) check for empty views already, but this is done at run time and can incur synchronization overhead. We can avoid this cost by removing the code at compile time.

Lower layer: MPI communication

The lower layer contains explicit communication using MPI, the Message Passing Interface. We have performed several preliminary experiments with lower-layer optimizations. For example, we could analyze the matrix splitting pattern in an application to determine where a point-to-point broadcast might yield software pipelining. However, these experiments require additional annotations and are part of our future work.

5.3.3 Analysis annotations

Here we describe the annotations for the optimizations described above. Specifying these annotations consist of three general parts: (1) convey the data dependence information about the routines, (2) define the flow values and transfer functions for the domain-specific program analyses, and (3) translate the optimizations themselves into code transformations.

Note that in the following discussion we often simplify the annotations examples to avoid clutter. In each example we highlight the relevant annotations by omitting the others. In the actual annotation file, there is only one set of annotations for each PLAPACK routine and those annotations contain all of the analysis and optimization information for that routine.

Modeling PLAPACK objects

The first step in encoding the PLAPACK optimizations is to provide an accurate dependence model for the dataflow analysis framework. This information is critical for both accurate and precise analysis and for preserving the program semantics. At the application level, linear algebra objects appear as variables of an opaque type called `PLA_Obj`. However, these objects do not behave like scalars, and the compiler cannot treat them that way. It is important to model their structure because data dependences may exist between their internal components that, if violated, could change the program's behavior.

As an example, consider the `PLA_Obj_split_4()` PLAPACK routine, whose behavior is shown graphically in Figure 5.2. This routine logically splits the matrix into four pieces by returning objects that represent ranges of the original matrix index space. It does not actually split the matrix data into four pieces. Internally, the library defines a *view* data structure that consists of row and column index ranges, along with a pointer to the actual matrix data.

To see how this complicates analysis, consider the code fragment in Figure 5.6. The first line declares several variables of type `PLA_Obj`, which is an opaque pointer to a *view* data

```
PLA_Obj A, A11, A12, A21, A22, B;
PLA_Create_matrix(num_rows, num_cols, &A);
PLA_Obj_split_4(A, 10, 10, &A11, &A12,
                           &A21, &A22);
B = A22;
```

Figure 5.6: This example creates a matrix and logically splits it into four submatrices.

structure. The second line creates a new matrix (both *view* and data) of the given size. The third line creates four *views* into the original data by splitting the index space into four logical quadrants. The fourth line performs a simple assignment of one *view* variable to another.

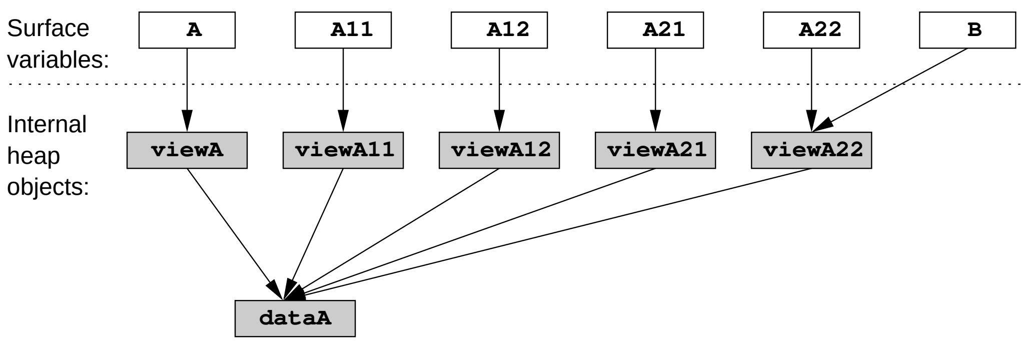

Figure 5.7: Library data structures have complex internal structure.

Figure 5.7 shows the resulting data structures graphically. The shaded objects are never visible to the application code, but accesses and modifications to them are still critical to preserving program semantics. For example, regardless of whether `A`, `A22`, or `B` is used, the compiler cannot change the order of library calls that update the data.

To avoid improper analysis and transformation, we use pointer and dependence annotations to communicate to the compiler the shapes of these data structures and the manner in which the library routines manipulate them. Figure 5.8 shows the pointer annotations for the `PLA_Obj_split_4()` routine. The input object `A` points to a view structure, which in turn pointer to the data itself and to the template that defines the data layout. The routine produces four new views, which are passed in by reference. The `on_entry` annotations describe the structure of the view and the four input pointers. The `on_exit` annotations indicate that the routine creates four new view structures, but that they all refer to the same underlying data and template.

Object type analysis

In PLAPACK the `PLA_Obj` data type represents all linear algebra objects. However, the library can create and manipulate many different kinds of objects, such as matrices, vectors, and scalars (which are called *multi-scalars* when they are replicated across processors). The internal library data structures maintain this type information at run-time so that the various library routines can handle these objects in the appropriate manner. The `PLA_Copy` routine, in particular, needs to know the type of the objects to decide how to perform the data copying.

We use the annotation language to track this information at compile time. Since object types are explicit in the creation routines, this analysis often succeeds at accurately determining their types statically. We use this information for two purposes. First, we can make sure that the types passed into a computation match the expected types. For example, the `PLA_Gemv` routine expects a matrix and a vector as input. We use the object type analysis to validate this requirement at compile time. If the compile-time check succeeds, then we can eliminate the run-time check to improve performance. If the compile-time check fails, we issue an informative message describing the nature and location of the error, which allows the programmer to fix it without having to execute and debug the program.

```
procedure PLA_Obj_split_4( A, length, width,
                           A11_ptr, A12_ptr, A21_ptr, A22_ptr)
{
  on_entry {
    A --> view_A { length, width,
                  data --> dataA
                  template --> the_template }
    A11_ptr --> A11
    A12_ptr --> A12
    A21_ptr --> A21
    A22_ptr --> A22
  }

  on_exit {
      A11 --> new view_A11 { length, width,
                             data --> dataA
                             template --> the_template }

      A12 --> new view_A12 { length, width,
                             data --> dataA
                             template --> the_template }

      A21 --> new view_A21 { length, width,
                             data --> dataA
                             template --> the_template }

      A22 --> new view_A22 { length, width,
                             data --> dataA
                             template --> the_template }
  }
}
```

Figure 5.8: The `on_entry` and `on_exit` annotations describe the shapes of the PLAPACK data structures.

The second use of the object type information is to perform algorithm selection at compile-time. In combination with the distribution analysis described below, we can often avoid the cost of the run-time switches that ordinarily make these choices. By itself, this optimization does not yield significant performance improvements. With run-time switches removed, however, we can often inline and further optimize the implementation of the chosen algorithm.

The `ObjType` property, shown in Figure 5.9, provides names for the different kinds of linear algebra objects. The base types are matrix, vector, projected vector (Pvector), and multi-scalar (Mscalar). An ordinary vector is distributed over the processor grid in a manner that improves matrix-vector operations [33]. A projected vector is a vector that is distributed like a column or row of a matrix. Multi-vectors consist of several vectors stored together. A duplicated projected multi-vector is a projected multi-vector that is replicated across the rows or columns of the processor grid. Figure 5.3 shows graphically two examples of projected multi-vectors being copied to distributed projected multi-vectors.

```
property ObjType : { Matrix,
                     Vector, Mvector,
                     Pvector, Pmvector, Dpmvector,
                     Mscalar }
```

Figure 5.9: The ObjType property captures the different kinds of linear algebra objects supported by PLAPACK.

We annotate each object creation routine to record the type of the object. Figure 5.10 shows the annotations for the routine that creates matrices. Note that we associate the type with the view structure. This decision will allow us to change the type of an object when it suits the computation better. For example, we can treat a panel of a matrix as a projected multi-vector, which helps reduce the amount of work in the copy routine.

```
procedure PLA_Matrix_create( datatype, length, width,
                             template_ptr,
                             v_align, h_align, matrix_out )
{
  on_entry { matrix_out --> the_matrix }

  analyze ObjType { the_view <- Matrix }

  on_exit { the_matrix --> new the_view { length, width,
                                          data --> new data } }
}
```

Figure 5.10: The object creation routines set the type of the object.

Special form analysis

Figure 5.11 shows the property that captures matrix special forms. This analysis plays a relatively minor role in our current PLAPACK optimizations, but we could improve it to capture more information about matrix structure.

```
property SpecialForm : { Zero,
                         Diagonal { Identity },
                         Triangular { Upper, Lower } }
```

Figure 5.11: The SpecialForm property expresses special case configurations of the elements of a matrix.

Distribution analysis

The most significant PLAPACK optimizations result from recognizing and exploiting special-case object distributions. Figure 5.12 shows the property annotations for tracking distribution. We define two separate properties, one for the rows of an object and one for the columns of an object, because the distribution of rows and columns can vary independently.

The distribution of an object is determined initially by the routine that creates it and subsequently by any splitting operations applied to it. Figure 5.13 shows representative analysis annotations for the `PLA_Obj_split_4` routine, which codify the effect of the

```
property RowDistribution : { Unknown { NonEmpty { Distributed,
                                                  Local { Duplicated },
                                                  Vector },
                                       Empty } }

property ColDistribution : { Unknown { NonEmpty { Distributed,
                                                  Local { Duplicated } },
                                       Empty } }
```

Figure 5.12: These two properties describe the different ways that the rows and columns of a matrix can be distributed.

routine on the possible input views. The actual annotations contain all of the cases, and they model the ability of the routine to split a matrix relative to any of the sides of the matrix. Figure 5.14 depicts these rules graphically.

Notice that in many instances the split routine produces one or more empty views. We can eliminate any subsequent computations on an empty view. Furthermore, consider what happens to a loop that repeatedly splits a matrix: if the matrix is already in the desired form, then the first iteration of the loop consumes all of the data and no further loop iterations are needed.

```
analyze RowDistribution {

  if (view_A is-exactly Distributed)
    { view_A11 <- Local
      view_A12 <- Local
      view_A21 <- Distributed
      view_A22 <- Distributed }

  if (view_A is-atleast Local)
    { view_A11 <- Local
      view_A12 <- Local
      view_A21 <- Empty
      view_A22 <- Empty }

  if (view_A is-exactly Empty)
    { view_A11 <- Empty
      view_A12 <- Empty
      view_A21 <- Empty
      view_A22 <- Empty }
}

analyze ColDistribution {

  if (view_A is-exactly Distributed)
    { view_A11 <- Local
      view_A12 <- Distributed
      view_A21 <- Local
      view_A22 <- Distributed }

  if (view_A is-atleast Local)
    { view_A11 <- Local
      view_A12 <- Empty
      view_A21 <- Local
      view_A22 <- Empty }

  if (view_A is-exactly Empty)
    { view_A11 <- Empty
      view_A12 <- Empty
      view_A21 <- Empty
      view_A22 <- Empty }
}
```

Figure 5.13: Analysis annotations for the `PLA_Obj_Split_4()` routine.

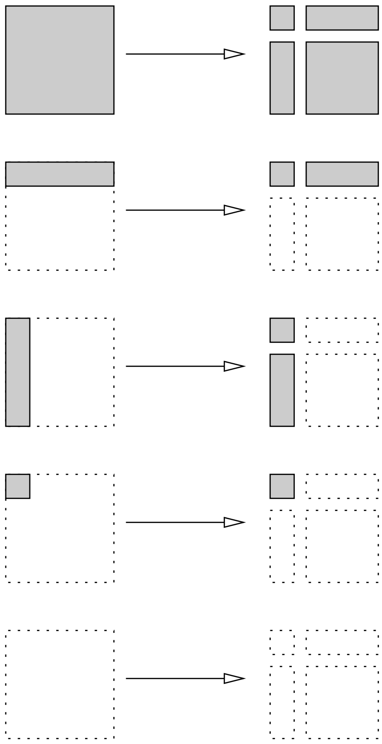

Figure 5.14: Different cases of the split operation. Depending on the distribution of the input matrix, the split routine can create one or more empty views.

Unused objects

The optimizations described above often remove unnecessary computations because they operate on empty views. However, the views and objects themselves are still created and destroyed by the program. Here we define an analysis to determine when an object or view is unused, in which case we can remove the code that creates it and the code that deletes it.

```
property ViewUsed : @backward { Used { Unused }}
```

Figure 5.15: The `ViewUsed` property defines a backward analysis that detects when objects are created but never used.

Figure 5.15 shows the property annotation for this analysis, which is a backward analysis. The idea is to mark objects as unused at the end of the program and then move backward through the code noting which objects actually participate in computations. If the object is still unused by the time we reach its creation, then it is never used. This analysis is essentially a library-specific formulation of liveness analysis, which augments the capabilities of the built-in liveness analysis.

The `PLA_Obj_free()` routine deletes views and objects, so we annotate it to set the object to the `Unused` state. The various computation routines mark their arguments as `Used`. Figure 5.16 shows representative annotations for this analysis.

5.3.4 Optimization annotations

The Broadway compiler uses the property and analysis annotations above to perform PLAPACK-specific program analysis. Once complete, it uses the result of this analysis to drive the PLAPACK code transformations.

Special-case inlining

The first step towards generating customized routines is to expose the implementations of the global layer routines. Unlike general approaches, we use the library-specific analysis

```
procedure PLA_Obj_free(obj_ptr)
{
  on_entry {
    obj_ptr --> obj --> view { data --> the_data }
  }

  analyze ViewUsed { view <- Unused }
}

procedure PLA_Scal(alpha, a)
{
  on_entry { alpha --> view_alpha
             a --> view_a }

  analyze ViewUsed {
    view_alpha <- Used
    view_a <- Used
  }
}
```

Figure 5.16: The `PLA_Obj_free()` routine initializes objects as unused. Any use of the object sets the state to used.

results to decide when to perform inlining. The advantage of this approach is that we can apply our library expertise to decide when inlining a routine is likely to be beneficial. For most of the level 3 BLAS routines, we use the following policy: if either the row or the column distributions of input objects are local, then inline the implementation. The motivation for this policy is that most of our specializations depend on local distributions, while the general purpose routines operate on fully distributed matrices. Figure 5.17 shows the annotations for the `PLA_Trsm()` routine, which performs a triangular solve with multiple right-hand sides.

Special-case routines

The PLAPACK analyses above allow us to identify opportunities to use special-case routines. These optimizations represent expert knowledge about the performance of the library that we learned from the library writers. We codify these "programming tricks" so that all

```
procedure PLA_Trsm(side, uplo, transa, diag, alpha, a, b)
{
  on_entry { alpha --> view_alpha
             a --> view_a
             b --> view_b }

  when (RowDistribution : view_a is-atleast Local ||
        ColDistribution : view_a is-atleast Local ||
        RowDistribution : view_b is-atleast Local ||
        ColDistribution : view_b is-atleast Local)
    inline;
}
```

Figure 5.17: We use the dataflow analysis information to define library-specific inlining policies.

users of the library can benefit from them.

In the first case, we exploit the fact that PLAPACK provides a routine to add two matrices, which is not typically part of the BLAS interface. A PLAPACK user might not know about this routine and would be likely to implement the addition as a matrix multiply where one input is the identity: $C < -C + A$ is equivalent to $C < -C + A * I$. We can use the special form analysis to recognize this pattern and replace it with the more efficient call. Figure 5.18 shows the annotations for this optimization. The replacement code fragment implements a local matrix copy by creating a temporary matrix that is the same shape and distribution and the `c` matrix, copying the data from the `b` object into the temporary, and then performing a local add operation.

This optimization is significant for several reasons. First, the replacement code sequence runs considerably faster than the original multiplication. Second, the library cannot efficiently recognize this special case at run-time because it would need to inspect the data to detect the identity matrix. Finally, we define annotations that will remove the identity matrix if it is no longer used anywhere in the program.

In the second special case, we improve the performance of the `PLA_Copy()` routine by treating a local piece of a matrix as a multi-scalar. PLAPACK programs often use

```
procedure PLA_Gemm(transa, transb, alpha, a, b, beta, c)
{
  on_entry {  alpha --> view_alpha
              a --> view_a
              b --> view_b
              beta --> view_beta
              c --> view_c        }

  when (SpecialForm : view_a is-exactly Identity)
    replace-with %{ PLA_Obj temp;
                    PLA_Matrix_create_conf_to($c, & temp);
                    PLA_Copy($b, temp);
                    PLA_Scal(temp, $alpha);
                    PLA_Scal($c, $beta);
                    PLA_Local_add(temp,$c);
                    PLA_Obj_free(temp); }%
}
```

Figure 5.18: Matrix multiplication is an inefficient way to add two matrices together. We can recognize this case and replace it with a much more efficient matrix addition routine.

the copy routine to duplicate a local piece of a matrix on all processors so that the data is co-located for subsequent computations. The copy routine implements a general algorithm for collecting the pieces of matrix and then duplicating them on the processors. However, when the submatrix to copy resides on one processor, this redistribution can be performed by a single broadcast operation. We force the `PLA_Copy()` routine to choose this implementation by telling it that the matrix is actually a multi-scalar. Figure 5.19 shows the annotation for this optimization, which inserts a `PLA_Obj_objtype_cast()`. Note that we avoid triggering this optimization repeatedly on the same call by checking first to make sure that the cast is necessary.

Algebraic simplifications

At both the global layer and the middle layer, we define optimizations that take advantage of algebraic identities. Figure 5.20 shows two examples for the `PLA_Scal()` routine, which applies a scalar multiplier to all the elements of a matrix. When the scalar is equal to one,

```
procedure PLA_Copy( source, target )
{
  on_entry { source --> source_view
             target --> target_view   }

  when (RowDistribution : source_view is-atleast Local &&
        ColDistribution : source_view is-atleast Local &&
        ! ObjType : source_view is-exactly Mscalar)
    replace-with %{
      {
        PLA_Obj_objtype_cast(${source}, PLA_MSCALAR);
        PLA_Copy(${source}, ${target});
      }
    }%
}
```

Figure 5.19: Some copy operations run much more efficiently when we treat a local piece of a matrix as a multi-scalar.

the multiplication has no effect, and we can remove it. When the scalar is zero, we can avoid the multiplication operations and just set the matrix to zero.

```
procedure PLA_Scal(alpha, a)
{
  on_entry { alpha --> view_alpha { length, width,
                                    data --> data_alpha }
             a --> view_a  }

  when (data_alpha == 1.0)
    replace-with %{ ; }%

  when (data_alpha == 0.0)
    replace-with %{ PLA_Obj_set_to_zero( $a ); }%
}
```

Figure 5.20: We can exploit domain-specific algebraic identities.

It might seem unlikely that such opportunities would occur in real programs, but they often appear after inlining. For example, the `PLA_Scal()` routine is called inside the implementation of `PLA_Gemm()` to handle the coefficients alpha and beta. In almost all cases these values are zero, one, or minus one. However, until we inline the routine we cannot take advantage of this information.

View split optimizations

Global layer routines, such as `PLA_Gemm()` and `PLA_Trsm()`, use the split routines to repeatedly carve off manageable pieces of the input matrices. However, when the input matrices are already in the desired form, the split operations serve no purpose. Figure 5.21 show optimizations that recognize and exploit this fact. For example, we can remove the vertical split routine when the input matrix is already a column panel.

```
procedure PLA_Obj_horz_split_2(A, length, upper_ptr, lower_ptr)
{
  on_entry { A --> viewA }

  when (RowDistribution : viewA is-atleast Local)
    replace-with %{
      PLA_Obj_view_all(${A}, ${upper_ptr});
    }%
}

procedure PLA_Obj_vert_split_2(A, width, left_ptr, right_ptr)
{
  on_entry { A --> viewA }

  when (ColDistribution : viewA is-atleast Local)
    replace-with %{
      PLA_Obj_view_all(${A}, ${left_ptr});
    }%
}
```

Figure 5.21: We can avoid splitting objects that already have the desired distribution.

Notice that we replace the split call with a call to `PLA_Obj_view_all()`, which makes a copy of the input view, instead of removing the call altogether. The reason we preserve the copy is that the program might alter the new view later on, which might have an unintended side-effect on the original view. We could use a separate analysis, inspired by copy propagation, to determine when such copies can be removed without altering the program semantics.

Empty view removal

We can remove any operation on an empty view. This optimization applies equally to computational operations as well as view operations. Figure 5.22 shows examples of the annotations for this optimization. Notice that if *any* of the dimensions of the inputs are empty, then we remove the call.

```
procedure PLA_Trsm(side, uplo, transa, diag, alpha, a, b)
{
  on_entry { alpha --> view_alpha
             a --> view_a
             b --> view_b }

  when (RowDistribution : view_a is-exactly Empty ||
        ColDistribution : view_a is-exactly Empty ||
        RowDistribution : view_b is-exactly Empty ||
        ColDistribution : view_b is-exactly Empty)
    replace-with %{ ; }%
}
```

Figure 5.22: We can remove operations on empty views.

Like the algebraic optimizations above, it seems surprising that a program would contain computations on empty views. These optimization opportunities arise when specializing a general-purpose global routine, such as `PLA_Trsm()`, for a context where the input matrices are not fully distributed. We show a concrete example of this optimization in our case study of the Cholesky factorization application.

View query simplifications

PLAPACK provides a number of routines that return information about a view. Using our compiler analysis we can often provide the answers to these queries at compile time. For example, the size of an empty view is zero. Figure 5.23 shows some examples of these optimizations.

While these optimizations seem simple and unlikely to affect performance, they

```
procedure PLA_Obj_global_length(obj, length_ptr)
{
  on_entry { obj --> view
             length_ptr --> length }

  when (RowDistribution : view is-exactly Empty)
    replace-with %{ ${length} = 0; }%
}

procedure PLA_Obj_global_width(obj, width_ptr)
{
  on_entry { obj --> view
             width_ptr --> width }

  when (ColDistribution : view is-exactly Empty)
    replace-with %{ ${width} = 0; }%
}
```

Figure 5.23: When the row distribution of a view is empty, its length is zero; when its column distribution is empty, its width is zero.

play an important role in simplifying the code. A typical PLAPACK algorithm works by repeatedly carving off and processing pieces of the input matrix until there is none left. This exit condition is tested by querying the current views to determine their size: the loop exits when the query returns zero. Therefore, the view query simplifications often allow the compiler to eliminate certain loops altogether. We show a concrete example of this process in the case study of Cholesky factorization later in this chapter.

Unused view removal

We use the `ViewUsed` analysis to find objects that are never used and remove the PLAPACK calls that create them. This analysis runs backwards through the program, starting at the deletion routines and propagating information towards the creation routines. At the creation routines we test to make sure that each view created is used for some subsequent computation. Figure 5.24 shows the annotations for the `PLA_Mscalar_create()` routine, which creates a multi-scalar. The only way that a multi-scalar can arrive at this routine in the Unused state is if the value is not accessed or modified between this program point and the point at which the object is deleted.

```
procedure PLA_Mscalar_create(datatype, owner_row, owner_col,
                             length, width,
                             template_ptr, mscalar_out )
{
  on_entry { mscalar_out --> the_mscalar }

  on_exit { the_mscalar --> new the_view { length, width,
                                           data --> new data } }

  when (ViewUsed : the_view is-exactly Unused)
    replace-with %{ ${the_mscalar} = 0; }%
}
```

Figure 5.24: We use the ViewUsed analysis to discover objects that are never used.

Notice that we replace the call to PLA_Mscalar_create() with an assignment that sets the object to zero. We then annotate the PLA_Obj_free() routine to look for null arguments and remove those calls. This routine can properly handle a null argument, but it is more efficient to remove it at compile-time, if possible. Figure 5.25 shows this annotation.

```
procedure PLA_Obj_free(obj_ptr)
{
  on_entry {
    obj_ptr --> obj --> view { data --> the_data }
  }

  when (obj == 0)
    replace-with %{ ; }%
}
```

Figure 5.25: Remove calls to PLA_Obj_free when the argument is null.

Redundant copy removal

PLAPACK programs use the copy routine to redistribute data so that it is in a suitable form for subsequent computations. For example, if we need to multiply two local submatrices, we can use the copy routine to make sure they are on the same processor. The general-purpose routines often make heavy use of the copy routine because they cannot rely on any particular distribution of their input parameters. We occasionally find a call site, however, where the input submatrices are already suitably distributed, and therefore no copying is necessary.

This situation occurs in the specialization of the triangular solve routine `PLA_Trsm()`. Unfortunately, the current annotation language cannot express this optimization because it requires the compiler to recognize and replace a *sequence* of library calls. We have defined the syntax for such an optimization and we anticipate having this capability in the future. For the experiments below, we use the idiomatic optimization strategy described in Section 5.2.3 to handle this particular case of the triangular solve routine.

Traditional optimizations

Traditional optimizations play an important role in simplifying the program during optimization. Constant propagation and dead-code elimination, in particular, help to eliminate unused code. For example, many of the PLAPACK computational routines accept special integer arguments that specify how to treat the input matrices. The general matrix multiply routine `PLA_Gemm()` accepts two such arguments, which tell the routine whether to transpose the input matrices. Once we inline the implementation of the routine, we expose the code that tests the transpose flag. Since the flag is often a compile-time constant, we can bypass the unused cases.

5.4 Experiments

In this section we present performance results obtained by applying our library-level optimization method to a group of PLAPACK applications and kernels. We find that the annotations effectively specify library-level optimizations and that these optimizations produce significant performance gains across layers of abstraction

5.4.1 Methodology

We start with straightforward versions of three PLAPACK programs, which implement three different equation solvers. The initial implementation of each program serves as a baseline and represents our ideal programming style: the code focuses on clearly expressing the algorithm rather than obscuring it with hand-coded optimizations. The programs generally use the highest layer of PLAPACK, but they are by no means poor implementations. They perform competitively with similar programs written using other parallel programming technologies. We apply the library-level optimization techniques to all three programs using a single set of PLAPACK annotations.

We run several passes of the optimization process to each program. Each pass first performs the library-specific analysis, followed by the library-specific code transformations. We then apply a set of "cleanup" optimizations, including constant propagation, control-flow simplification, and dead-code elimination. We repeat this process until no new code transformations occur, which requires approximately four or five iterations in most cases.

We compare the performance of the original baseline version of each program against the version produced by Broadway. In addition, we show the performance of the programs without the redundant copy idiom, which allows us to remove a redundant copy. For one of the programs, we also compare the performance of our approach against a hand-coded version produced by the authors of the library. We measure the execution time of each program version on a variety of problem sizes and different numbers of processors.

5.4.2 Programs

We use the following three programs for these experiments:

- **LU factorization.** This program uses the familiar Gaussian elimination algorithm to factor a matrix into lower and upper triangular parts. For a matrix A it solves the equation $A = LU$ to compute lower triangular matrix L and upper triangular matrix U.

- **Cholesky factorization.** This program factors a symmetric, positive-definite matrix into the product of a lower triangular matrix and its transpose. For a matrix A it solves the equation $A = LL^T$ to compute lower triangular matrix L.

- **Kernel from Lyapunov equation solver.** The Lyapunov equation [9] arises in control theory applications. It is more complex than the two problems above and poses a more challenging optimization problem for our approach. The PLAPACK authors provided our baseline implementation [71].

As mentioned earlier, we start with a simple and clear implementation of each solver. In particular, we want the baseline programs to highlight the mathematical algorithms, not the underlying implementation and performance issues. Figure 5.26 shows the baseline implementation of the Cholesky factorization algorithm—it consists of only seven statements. PLAPACK is an ideal target in this respect because the high-level library routines allow users to write code that looks very much like familiar mathematical equations and algorithms.

5.4.3 Annotations

Earlier in this chapter we present a number of examples of the PLAPACK annotations. For space reasons, however, we do not include the entire annotation file. The following summary provides some overall information about the contents of the annotation file and their relationship to the PLAPACK library.

```
while ( 1 ) {
    PLA_Obj_split_size ( a_next, PLA_SIDE_TOP,
                         & size_top, & owner_top );
    PLA_Obj_split_size ( a_next, PLA_SIDE_LEFT,
                         & size_left, & owner_left );

    if ( size = min ( size_top, size_left )) break;

    PLA_Obj_split_4 ( a_next, size, size, & a_cur, PLA_DUMMY,
                                          & a_col, & a_next );

    PLA_Local_chol( uplo,  a_cur );

    PLA_Trsm ( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
               PLA_TRANS, PLA_NONUNIT_DIAG,
               one, a_cur, a_col );

    PLA_Syrk ( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
               min_one, a_col, one, a_next );
 }
```

Figure 5.26: The main loop of the baseline Cholesky implementation corresponds almost exactly to the mathematics that it implements.

- The PLAPACK library consists of about 45K lines of C code. The PLAPACK annotation file consists of about 3400 lines of annotations.

- We annotated 85 PLAPACK routines – each routine averages about 40 lines of annotations. The actual number of lines per routine, however, varies quite widely. Most routines require only about 20 lines to annotate, but several routines, such as the view splitting routines, require as many as 200 lines to handle all of the analysis cases.

- In total, about 30% of the annotation file is devoted to the pointer and dependence information. In the current implementation this information must be repeated in each routine.

- The annotations define seven library-specific program analyses (property annotations). Only one of them, the ViewUsed property, is a backward analysis.

- There are 48 error reporting and debugging annotations.
- There are 70 code transformation annotations, which specify library-level optimizations. Of these, the majority specify optimizations that remove useless computations – for example, computing on an empty view. Many of the remaining optimizations describe the conditions for inlining the implementation of a routine. This emphasis reflects our goal of generating customized code from general-purpose routines.

5.4.4 Platform

For these experiments we use Broadway as a cross compiler: we optimize the programs locally on our Pentium 4 workstation running Linux, and then copy the source over to the parallel environment. We compile and execute the programs on an IBM Power4-based multiprocessor. This system consists of a tightly-bound network of three 16-way symmetric multiprocessors (SMP), one 32-way SMP, and 32 4-way SMPS. Each processor runs at 1.3Ghz. We compile using the vendor-supplied tools, and we link against the vendor supplied Message Passing Interface (MPI), which handles the non-uniform memory architecture.

5.5 Results

For each of the three test programs, we measure the execution time of the base version and two Broadway optimized versions: one with the redundant copy idiom and one without. For the Cholesky factorization program, we also time the version written by the PLAPACK implementation team. We run each program on a range of input matrix sizes, from 1000x1000 to 8000x800, and on a range of processor grids, from 2x2 processors to 10x10 processors.

We find the following general results:

- Our PLAPACK annotations consistently improve performance. Depending on the program, the problem size, and the number of processors the improvement ranges

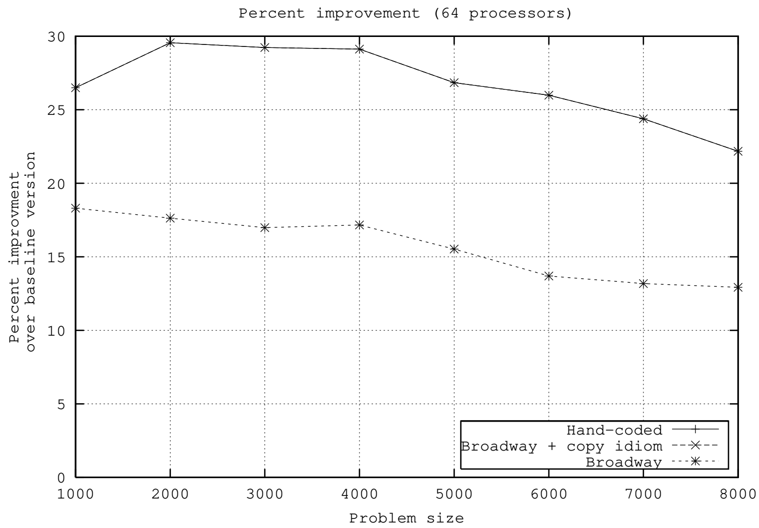

Figure 5.27: Percent improvement for Cholesky on 64 processors. (Note that the top line represents both the hand-coded case and the Broadway+copy-idiom case.)

from just a few percent to 30 percent.

- Overall, the performance improvement increases as we increase the number of processors and decreases as we increase the problem size. This suggests that our annotations are effectively eliminating the software overhead associated with the library layers.

- While the redundant copy idiom noticeably improves performance, the rest of the annotations also contribute significantly. We will continue to use the idiom until the annotation language can express redundant copy removal.

For each program we show the performance improvement obtained by using Broadway. The three program-specific graphs show the percent improvement in execution time over the baseline version for 64 processors (an 8x8 grid) over a range of problem sizes. For each program we show the results obtained by using the specialization strategy alone as well as the results obtained by including the redundant copy idiom.

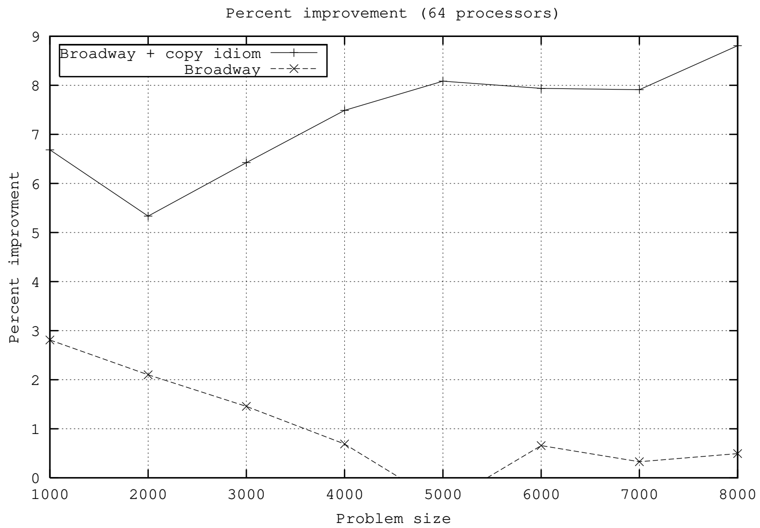

Figure 5.28: Percent improvement for LU on 64 processors.

Figure 5.27 shows the results for the Cholesky factorization program. The code generated by the specialization strategy alone runs 13 to 18 percent faster than the baseline version. When we include the redundant copy idiom, the improvement jumps to between 22 and 29 percent. In this case, our Broadway-generated version runs as fast as the hand-coded Cholesky factorization written by the library authors, which serves as an upper bound for our approach. In fact, the performance curves for these two versions are practically identical. The reason is that many of the optimizations codified in our annotations come from insights into this hand-coding process. In annotation form, however, we can easily apply the same optimizations to the other two test programs.

Figure 5.28 shows the result for the LU factorization. This program is dominated by calls the triangular solve routine, so the redundant copy idiom makes a significant difference. Manual inspection of the Broadway-generated code indicates that there are few additional optimization opportunities at the PLAPACK level of abstraction.

Figure 5.29 shows the result for the Lyapunov equation solver. These results represent a more significant test of our approach because the program is more complex than the

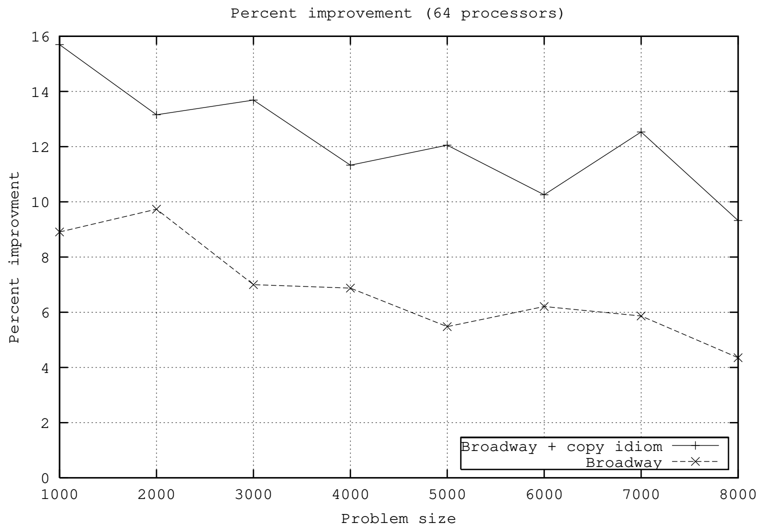

Figure 5.29: Percent improvement for Lyapunov on 64 processors.

others. The specialization strategy improves performance by 5 to 10 percent. The addition of the redundant copy idiom improves performance by 9 to 15 percent.

Figure 5.30 shows the results for all three programs on a large fixed-size problem, plotted against the number of processors. For Cholesky factorization and the Lyapunov solver, the library-level optimizations provide consistent and scalable performance improvement. The LU factorization appears to scale more poorly beyond 36 processors, but still maintains a consistent improvement. We believe that for a fixed problem size, as the number of processors increases the software overhead of managing the distributed algorithms has a more significant impact on performance. Our optimizations remove some of that overhead, which helps fixed-size problems to benefit more from larger machines.

Figure 5.31 shows the execution times of the baseline and Broadway optimized versions of all three programs. The IBM Power4 multiprocessor provides consistently high performance on our applications. While significant advances in parallel computer hardware, such as lower communication latency and faster processor speeds, tend to hide advances in compiler optimization, our approach still yields valuable performance gains.

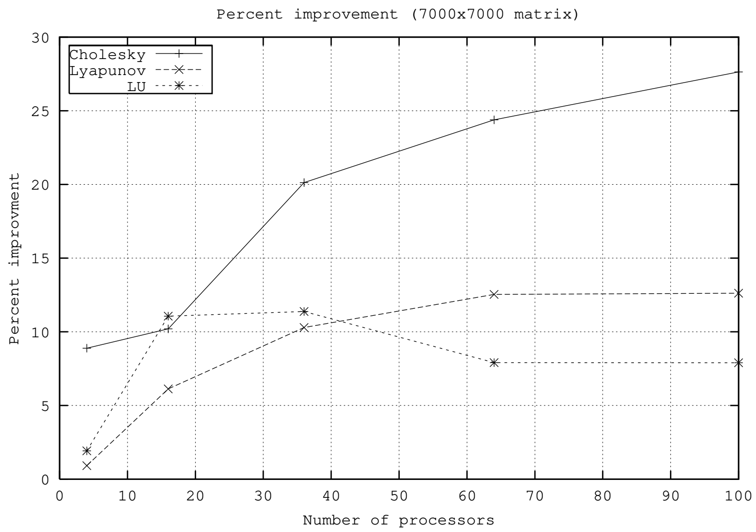

Figure 5.30: Our optimizations are consistent and scale well.

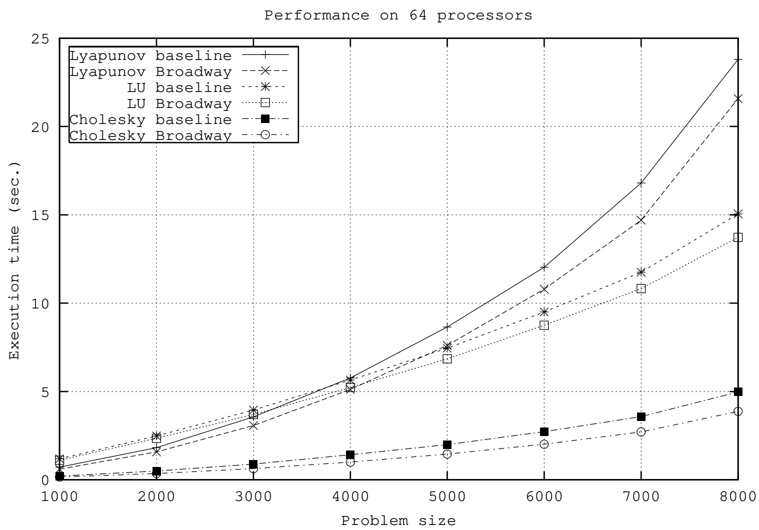

Figure 5.31: Execution time for the three programs.

5.5.1 Discussion

The experiments described above lead us to believe that library-level optimization is an effective way to optimize layered scientific systems. Several observations about the experiments contribute to this conclusion:

- The technique works because it exploits domain-specific semantics that would otherwise be ignored by conventional techniques. Without a notion of matrices and data distributions, none of the optimizations we applied to PLAPACK are possible.
- The technique is effective because it crosses software layers, optimizing each layer in the context of the application and the layers above. Our design allows the compiler to shift from one domain to the next, systematically processing all the layers.
- Even with limited configurability, the annotations capture useful and interesting properties of the layer abstractions. We find only a few optimizations that we could not adequately express in the language; these optimizations work on MPI routines, and require an accurate model of the communication patterns.
- Developing the annotations can be a difficult task, but it is mitigated by two factors. First, we can develop annotations incrementally, adding new optimizations as we discover them. Second, one set of annotations can be used to optimize any application that uses that library.
- Applying the process manually is not reasonable, partly because the task is so difficult, and partly because the resulting code is incomprehensible and unmaintainable. In early optimization experiments, before we had a complete implementation, we applied these optimization by hand. We had to pour over the programs searching for optimization opportunities, and then we had to carefully modify the programs to make sure they still ran correctly. This often involved tediously applying the same

optimizations over and over in different places. Furthermore, small changes to the baseline programs translate into significant recoding efforts in the optimized versions.

5.6 Conclusions and future work

In this chapter we present a new approach to optimization that exploits domain-specific information at the library interface level. The combination of domain-specific program analysis and simple code transformations is an effective tool for expressing a variety of optimizations. Our approach changes modularity and encapsulation from a performance liability into an important asset for performance improvement.

While this foundational work has produced promising results, we believe that it only scratches the surface of large untapped source of optimization. We have already identified a number of potential improvements and future directions:

- **More types for program analysis.** Our annotation language currently supports a relatively simple class of program analysis problems. We have designed a more general approach that provides the annotator with a variety of commonly-used data structures. With these capabilities, the compiler can construct more complex models of the library's domain.

- **Code patterns.** The current compiler only allows the annotations to replace individual library calls with other code. However, libraries often contain composite routines that implement a particular sequence of calls in a more efficient manner. In the future, we would like to provide an annotation that recognizes stylized patterns of library routine usage and can replace or alter the entire sequence.

- **Domain-specific traditional optimizations.** In the current compiler implementation, the traditional optimizations, such as constant propagation and dead-code elimination, work on library routines in exactly the same way that they work on primitive operations. For other traditional optimizations, however, we can formulate analogous optimizations that work on library routines by analogy to their primitive counterparts. For example, if we tell the compiler that a particular library routine effectively creates a copy of an object, then it can apply a domain-specific version of copy propagation. Other traditional optimizations lend themselves to this technique: common subexpression elimination, management of resources, and scheduling. By exploiting existing algorithms, we can continue to keep the annotations simple.

Our work also suggests a new approach to designing software libraries that takes advantage of compiler support. In the future, such a library might consist of two distinct interfaces, one for the programmer to use and one for the compiler to target. The programmer's interface focuses on providing straightforward and intuitive access to the library's domain without exposing implementation and performance details. This high-level interface serves two purposes: first, it makes the programmer's job easier, and second, it provides domain-specific information for the compiler. The second part of the library interface consists of low-level library routines that the compiler can use as a code generation target. At this level, the routines implement the basic building blocks of the domain. The compiler analyzes the high-level interface and generates an appropriate implementation by assembling these building blocks. The low-level routines might have explicit performance characteristics and complex usage rules. For example, we might choose to separate parameter checks from computational routines and then only generate calls to the checks where they are needed.

Chapter 6

Library-level Error Detection

In this chapter we present the Broadway compiler as a tool for error checking. In the previous chapter we used analysis annotations to identify optimization opportunities: places where a program uses library routines *inefficiently*. Here we use the same machinery to detect programming errors and security holes: places where a program uses library routines *incorrectly* or *unsafely*. We show that our annotation language can capture a range of error checking problems and that the compiler can detect these errors accurately.

6.1 Introduction

As software has become more pervasive, the impact of programming errors has grown dramatically. More and more critical applications, such as business, medical and government operations, are controlled by software. At the same time, as software systems become larger and more complex, the likelihood that they contain errors grows. Furthermore, many of these applications provide services over the Internet, which means that programming errors can become remotely exploitable security holes.

In practice, error checking is a time consuming task with inconsistent results. It typically involves a combination of code reviews and testing, both of which can miss errors—

especially subtle errors in infrequently executed code.

In theory, error checking is a task for formal verification, which can guarantee program correctness. Unfortunately, despite many successes checking hardware systems, formal verification of large software systems remains a challenging and labor intensive task [56]. A significant obstacle, even for experts, is coming up with a complete formal specification of the program. The job involves translating every detail of the program's intended behavior into a logical form, often using an idiosyncratic notation particular to the chosen verification tool. Even with such a specification, many of the tools are only partially automatic and may require the user to provide intermediate results such as loop invariants. Finally, these tools tend to consume extreme amounts of computer time and memory, even for small programs.

A promising direction for error checking research is towards intermediate solutions that overcome many of the drawbacks of full formal verification by giving up some of its power. These partial program checkers verify a limited class of program properties, but are more efficient, fully automatic, and require relatively simple specifications. A number of partial program checkers, in both industry and academia, have recently demonstrated the effectiveness of this approach [35, 76, 27]. In this chapter we present our experience with using the data-flow analysis capabilities of the Broadway compiler for partial program checking.

6.1.1 Library-level error detection

Since we designed the annotation language to describe library interfaces, we focus on detecting errors that result from incorrect or unsafe uses of library routines. While this limits the kinds of errors we can catch, it has a number of important benefits.

First, the improper use of library routines is a significant source of programming errors, especially errors with security implications. The reason is that libraries provide much of the functionality that allows programs to interact with one another, both locally and across a network. While erroneous application logic is ultimately to blame for most security holes, library routines are often the doorway by which intruders gain access. So, rather than fully verifying the application logic, we can produce a conservative approximation of potential vulnerabilities by analyzing the I/O routines and the flow of data between them. As a simple example, a program that never creates an Internet domain socket is unlikely to have any remotely exploitable vulnerabilities. (Note however, that a program that does create such a socket is not necessarily vulnerable.)

Second, as programming languages and run-time systems become more robust, many popular low-level attacks, such as buffer overflows, will no longer work. For example, Java does not allow direct manipulation of pointer values, and therefore it is less vulnerable to memory manipulation attacks such as stack smashing. Hackers will need to focus their efforts on the higher levels of software systems in order to find vulnerabilities.

Third, by recognizing the domain-specific meanings of the library functions, we can collect deeper information about how a program works, and consequently what it might be doing wrong. For example, cryptography libraries introduce a semantic distinction between plain text, which is readable by anyone, and cipher text, which is safely encrypted. We can use this distinction to detect if a program accidentally transmits private information in an insecure way. Note however, that the two different categories of text only exist as a result of the library specification. We could never expect a program analysis tool to discover these high-level properties automatically, even with access to the library source code. In fact, breaking the encapsulation actually hurts program analysis by exposing the compiler to complex implementation details or machine-specific system code. By capturing the library-specific information in annotations, we simultaneously increase the range and quality of errors that we can detect, while reducing the amount of work the compiler has to do to detect them.

6.1.2 Overview

In this chapter we show how to use the Broadway compiler to find a range of programming errors. We describe a set of representative error checking problems and show how to express them using the annotations. We run the compiler on a number of real C programs and measure the run time of the analysis and the accuracy of the results.

This chapter makes the following contributions:

- We show that interprocedural dataflow analysis is a good tool for error checking. It works on a variety of problems including state machine checks and information flow checks.
- We evaluate our approach on several real, unmodified Linux system tools. For one significant class of security vulnerabilities, our compiler is able to accurately identify all of the known bugs with no false positives.

6.2 Error checking problems

In this section we formulate a number of library-level error checking problems and show how to express them using the annotation language. Developing these annotations can be a challenging task that often amounts to formalizing an aspect of the library that is only expressed informally (as in user documentation) or in the implementation. Therefore, our approach in this section is to show how we express an error checking problem using annotations, including design decisions and tradeoffs, in addition to presenting the final annotations.

Our selection of problems reflects several goals of this research. First, we target realistic errors that actually occur in programs and that could cause significant damage. For example, one of the errors we detect, the format string vulnerability, has been the subject of numerous computer security alerts because it can be used to gain complete control over a remote computer. To this end we use the C standard library for our experiments because it is so widely used and controls access to almost all important system services. Second, we bring together a number of error checking problems that previous research efforts have only presented in isolation. We will not consider language-level errors at all, such as array bounds checking or null pointer checking.

The five errors we detect are:

- File access error: make sure that files are open when accessed.
- Format string vulnerability (FSV): make sure that format strings do not contain untrusted data.
- Remote access vulnerability: make sure that a remote hacker cannot control sensitive functions, such as execution of other programs.
- Remote FSV: an enhanced version of the format string check that determines when vulnerabilities are remotely exploitable.
- FTP behavior: make sure that the program cannot be tricked into reading and returning the contents of arbitrary local files.

6.2.1 File access errors

Most libraries have specific rules that govern the use of the library routines. These include constraints on the combinations of routines used, the ordering of the calls, and the objects passed in. While many of the rules are basic and essential to the use of the library, usage errors still arise and they should be checked automatically.

Library usage rules are often difficult to check manually. Many libraries have complex interfaces with large numbers of routines, and the programmer may not fully understand their use. In particular, errors often arise when the programmer tries to use the more advanced routines in a library, usually to improve performance. However, even when the programmer is a library expert, library calls may be scattered over many procedures and source files, making it difficult to check that they are correct. For example, when using locks, in order to avoid a double locking bug, the programmer must ensure that the lock is free on all possible paths leading to the lock call. By codifying the usage rules as annotations and then automating the program checking, we can make sure that the rules are enforced uniformly and thoroughly.

Incorrect programs may fail at run time without providing adequate information to diagnose the error. For example, a failure could manifest itself as a segmentation violation deep inside the library code, which is typically compiled without debugging symbols. Even worse, the program could execute for some time before encountering the error. The annotations allow us to report usage errors in a meaningful, library-specific way that helps the programmer understand and fix the program. By performing the checks at compile time, the programmer can discover errors before the program is deployed.

File access rules are one example of this kind of usage constraint. A program can only access a file between the proper open and close calls. The purpose of this analysis is to detect possible violations of this usage rule. The flow value for the analysis consists of the two possible states, "Open" and "Closed". Figure 6.1 shows the annotation that defines the flow value.

To track this state, we annotate the various library functions that open and close files. Figure 6.1 shows the annotations for the `fopen()` function. The `on_entry` and `on_exit` annotations describe the pointer behavior of the routine. This routine returns a pointer to a new file stream, which points to a new file handle. The `analyze` annotation sets the state of the newly created file handle to open.

Our language can handle much of the complex behavior of UNIX file access. For example, we can use the pointer annotations to model functions such as `fileno()` and `dup2()`, which create multiple aliases for the same file handle.

At each use of a file stream or file descriptor, we check to make sure the state is open. Figure 6.1 shows an annotation for the `fgets()` function, which emits an error if

```
property FileState : { Open, Closed }
              initially Closed

procedure fopen(path, mode)
{
  on_exit { return -->
                   new file_stream -->
                             new file_handle }
  analyze FileState {
    file_handle <- Open
  }
}

procedure fgets(s, size, f)
{
  on_entry { f --> file_stream --> handle }
  error
    if (FileState : handle could-be Closed)
     "Error:␣file␣might␣be␣closed";
}
```

Figure 6.1: Annotations for tracking file state: to properly model files and files descriptors, we associate the state with an abstract "handle".

there is any possibility that the file is closed.

6.2.2 Format string vulnerability (FSV)

A number of output functions in the C standard library, such as `printf()` and `syslog()`, take a format string argument that controls output formatting. The format string vulnerability (FSV) occurs when untrusted data ends up as part of the format string [68]. A hacker can exploit this vulnerability to gain control over the program [13]. Unfortunately, a number of well-known programs that contain this vulnerability are system daemons, such as the FTP daemon and the domain name server, that run with root privilege. As a result, this vulnerability poses a serious security problem and has been the subject of many CERT advisories [16, 17].

Figure 6.2 shows an example of the bug. The data read into `buffer` is concate-

```
read(buffer, size, socket);  // -- Get data from a client
strcat(format, buffer);
printf(format);         // -- Format string contains client data
```

Figure 6.2: Code fragment with a format string vulnerability: if the client data contains ”%s”, then printf improperly accesses the stack looking for a string pointer argument.

nated onto the format string. If this data contains any of the special `printf()` character sequences, such as `"%s"`, the call to printf may fail. A hacker exploits the vulnerability by sending the program a carefully crafted input string that causes part of the code to be overwritten with new instructions. This simple example belies the difficulty of detecting this vulnerability. Many of our input programs have complex logging and error reporting functions, which construct format strings dynamically at run-time. These functions are called in many places and accept many different kinds of data from different sources.

To detect format string vulnerabilities, we define an analysis that determines when data from an untrusted source can become part of a format string. Following the terminology introduced in the Perl programming language [91] and adopted by Shankar et. al. [76], we consider data to be *tainted* when it comes from an untrusted source. We track this data through the program to make sure that all format string arguments are *untainted*.

Our formulation of the taint analysis starts with a definition of the taint property, shown at the top of Figure 6.3, which consists of two possible values, `Tainted` and `Untainted`. We then annotate the standard C library functions that produce tainted data. These include such obvious sources of untrusted data as `scanf()` and `read()`, and less obvious ones such as `readdir()` and `getenv()`. Figure 6.3 shows the annotations for the `read()` system call. Notice that the annotations assign the `Tainted` property to the contents of the buffer rather than to the buffer pointer.

Taintedness is a property that can propagate across functions that manipulate the data. For example, if the program concatenates two tainted strings, the result is also tainted. Therefore, we also annotate various string operations to indicate how they affect the taint-

```
property Taint : { Tainted, Untainted }
           initially Untainted

procedure read(fd, buffer_ptr, size)
{
  on_entry { buffer_ptr --> buffer }
  analyze Taint {
    buffer <- Tainted
  }
}

procedure strdup(s)
{
  on_entry { s --> string }
  on_exit { return --> string_copy }
  analyze Taint {
    string_copy <- string
  }
}

procedure syslog(prio, fmt, args)
{
  on_entry { fmt --> fmt_string }
  error
    if (Taint : fmt_string could-be Tainted)
      "Error:␣tainted␣format␣string.";
}
```

Figure 6.3: Annotations defining the `Taint` analysis: taintedness is associated with strings and buffers, and taintedness can be transfered between them.

edness of their arguments. Figure 6.3 shows the annotations for the `strdup()` library function.

Finally, we annotate all the standard C library functions that accept format strings (including `sprintf()`) to report when the format string is tainted. Figure 6.3 shows the annotation for the `syslog()` function, which is often the culprit in FSV exploitation.

6.2.3 Remote access vulnerability

As programs become more robust to low-level attacks, such as buffer overflows, high-level attacks that exploit information flow have become an important class of vulnerabilities. Programs that provide critical system services, such as daemons and servers, can be tricked into divulging private information or performing unintended tasks. Here we develop an analysis to determine whether a remote client can gain control over sensitive system calls, such as opening files or executing other programs.

Hostile clients can only manipulate the behavior of a server or daemon program through the various data input sources. We can approximate the extent of this control by tracking the data from these sources and observing how it is used. We label input sources, such as file handles and sockets, according to the level that they are trusted. All data read from these sources is labeled likewise.

For our analysis we model three levels of trust—internal (trusted), locally trusted (for example, local files), and remote (untrusted). Figure 6.4 defines the flow value for this analysis.

We start by annotating functions that return fundamentally untrusted data sources, such as Internet sockets. Figure 6.4 shows the annotations for the `socket()` function. The level of trust depends on the kind of socket being created. When the program reads data from these sources, the buffers are marked with the Trust level of the source.

The Trust analysis has two distinguishing features. First, data is only as trustworthy as its least trustworthy source. For example, if the program reads both trusted and untrusted data into a single buffer, then we consider the whole buffer to be untrusted. The nested structure of the lattice definition captures this fact. Second, untrusted data has a domino effect on other data sources and sinks. For example, if the file name argument to `open()` is untrusted, then we treat all data read from that file descriptor as untrusted. The annotations in Figure 6.4 implement this policy.

As with the taint analysis above, we annotate string manipulation functions to prop-

```
property Trust : { Remote
                     { External
                         { Internal }}}

procedure socket(domain, type, protocol}
{
  on_exit { return --> new file_handle }
  analyze Trust {
    if (domain == AF_UNIX)
       file_handle <- External
    if (domain == AF_INET)
       file_handle <- Remote
  }
}

procedure open(path, flags)
{
  on_entry { path --> path_string }
  on_exit { return --> new file_handle }
  analyze Trust {
    file_handle <- path_string
  }
}
```

Figure 6.4: Annotations defining the Trust analysis. Note the cascading effect: we only trust data from a file handle if we trust the file name used to open it.

agate the Trust values from one buffer to another. We generate an error message when untrusted data reaches certain sensitive routines, including any file access or manipulation, or reaches any program execution, such as `exec()`.

6.2.4 Remote FSV

The taint analysis defined above tends to find many format string vulnerabilities that are not exploitable security holes. For example, the code fragment in Figure 6.5 has a vulnerability only if the intruder already has root permission and can modify files in the `/etc` directory.

To identify exploitable format string vulnerabilities more precisely, we can combine the taint analysis with the Trust analysis, which specifically tracks data from remote

```
f = fopen("/etc/config.file", "r");
fgets(buffer, 100, f);
printf(buffer);
```

Figure 6.5: This format string vulnerability is only exploitable by the super-user.

sources. Figure 6.6 shows a revised annotation for the `printf()` that also checks the Trust property.

```
procedure syslog(prio, fmt, args)
{
  on_entry { fmt --> fmt_str }
  error
    if ((Taint : fmt_str could-be Tainted) &&
        (Trust : fmt_str could-be Remote))
     "Error:␣exploitable␣format␣string.";
}
```

Figure 6.6: The fgets() function prints an error if the file might be closed.

6.2.5 FTP behavior

The most complex of our client analyses checks to see if a program can behave like an FTP (file transfer protocol) server. Specifically, we want to find out if its possible for the program to send the contents of a file to a remote client, where the name of the file read is determined, at least in part, by the remote client itself. This behavior is not necessarily incorrect: it is the normal operation of the two FTP daemons that we present in our results. We can use this error checker to make sure the behavior is not unintended (for example, in a finger daemon) or to validate the expected behavior of the FTP programs.

We use the Trust analysis defined above to determine when untrusted data is read from one stream to another. However, we need to know that one stream is associated with a file and the other with a remote socket. Figure 6.7 defines the flow value to track different kinds of sources and sinks of data. We can distinguish between different kinds of sockets, such as "Server" sockets, which have bound addresses for listening, and "Client" sockets, which are the result of accepting a connection.

Whenever a new file descriptor is opened, we mark it according to the kind. In addition, like the other analyses, we associate this kind with any data read from it. We check for FTP behavior in the `write()` family of routines, shown in Figure 6.7, by testing both the buffer and the file descriptor.

```
property FDKind : { File,
                    Client, Server,
                    Pipe, Command, StdIO }

procedure write(fd, buffer_ptr, size)
{
  on_entry { buffer_ptr --> buffer
             fd --> file_handle }
  error
    if ((FDKind : buffer could-be File) &&
        (Trust : buffer could-be Remote) &&
        (FDKind : file_handle could-be Client) &&
        (Trust : file_handle could-be Remote))
      "Error:␣possible␣FTP␣behavior";
}
```

Figure 6.7: Annotations to track kinds of data sources and sinks. In combination with Trust analysis, we can check whether a call to `write()` behaves like FTP.

6.3 Experiments

This section describes the experiments we use to evaluate our approach to error detection. We use a suite of 18 real C programs, and we check each program for all five error detection problems described above. In Section 6.4 we present the results of these experiments and discuss the effectiveness of our system at finding different kinds of errors. In Chapter 7 we describe the underlying client-driven analysis algorithm and present a study of the impact of analysis precision on the cost and accuracy of the results.

Program	Description	Priv	LOC	CFG nodes	Procedures
stunnel 3.8	Secure TCP wrapper	✓	2K / 13K	2264	42
pfingerd 0.7.8	Finger daemon	✓	5K / 30K	3638	47
muh 2.05c	IRC proxy	✓	5K / 25K	5191	84
muh 2.05d	IRC propy	✓	5K / 25K	5390	84
pure-ftpd 1.0.15	FTP server	✓	13K / 45K	11,239	116
crond (fcron-2.9.3)	cron daemon	✓	9K / 40K	11,310	100
apache 1.3.12 (core only)	Web server	✓	30K / 67K	16,755	313
make 3.75	make		21K / 50K	18,581	167
BlackHole 1.0.9	E-mail filter		12K / 244K	21,370	71
wu-ftpd 2.6.0	FTP server	✓	21K / 64K	22,196	183
openssh client 3.5p1	Secure shell client		38K / 210K	22,411	441
privoxy 3.0.0	Web server proxy	✓	27K / 48K	22,608	223
wu-ftpd 2.6.2	FTP server	✓	22K / 66K	23,107	205
named (BIND 4.9.4)	DNS server	✓	26K / 84K	25,452	210
openssh daemon 3.5p1	Secure shell server	✓	50K / 299K	29,799	601
cfengine 1.5.4	System admin tool	✓	34K / 350K	36,573	421
sqlite 2.7.6	SQL database		36K / 67K	43,333	387
nn 6.5.6	News reader		36K / 116K	46,336	494

Table 6.1: Properties of the input programs. Many of the programs run in a higher "privileged" mode, making them more security critical. Lines of code (LOC) is given both before and after preprocessing. CFG nodes measures the size of the program in the compiler internal representation—the table is sorted on this column.

6.3.1 Programs

Table 6.1 describes our input programs. We use these particular programs for our experiments for a number of reasons. First, they are all real programs, taken from open-source projects, with all of the nuances and complexities of production software. Second, many of them are system tools or daemons that have significant security implications because they interact with remote clients and provide privileged services. Finally, we use security advisories to find versions of programs that are known to contain format string vulnerabilities. In addition, we also obtain subsequent versions in which the bugs are fixed, so that we can confirm their absence.

We present several measures of program size, including number of lines of source code, number of lines of preprocessed code, and number of procedures. The number of procedures is an important measure to consider for context-sensitive analysis. CFG nodes measures the size of the program in the compiler internal representation and probably provides a better basis for comparing programs. The table is sorted by the number of CFG nodes, and we use this ordering in all subsequent tables.

6.3.2 Methodology

In this chapter, we are primarily interested in evaluating the quality of the error checking results produced by Broadway. Therefore, we focus on measuring the number of errors reported and determining whether or not they correspond to real errors in the input programs. In Chapter 7 we perform an in-depth study of the cost of error detection, which includes a comparison of the performance and accuracy of several different analysis algorithms. We obtain the results in this chapter using the best of these algorithms, which is our own client-driven analysis algorithm.

Ideally, we would evaluate our system by comparing the number of errors reported to the actual number of errors present in the program. Since our analysis is sound and conservative the errors that it detects are always a superset of the actual errors. Therefore, the difference between these two numbers is the number of false positives. In the case of format string vulnerabilities, we can compute this number precisely because the programs in our test suite have known vulnerabilities. However, for the other errors it is a considerable task to manually examine the programs and determine which errors are true errors. Instead, we look at overall trends in the error reports, and we select a few cases to explore in more depth.

An alternative approach is to add errors to the programs and then verify that our system finds them. The merit of this approach is that it gives us a precise measure of the false positive rate. The problem with this approach is that we are likely to add errors in specific ways, which could bias our results. Real programs might contain errors that occur in ways we would not have foreseen. Anecdotal evidence suggests that this is the case: we are often surprised at the actual causes of errors when we track them down.

6.3.3 Annotations

In the sections above we describe the error detection analysis problems and give representative examples of the annotations. As is the case in Chapter 5, we cannot include all of the annotations for the Standard C Library. The following summary, however, provides some sense of the contents of the annotation file.

- The Standard C Library annotation file contains about 3300 lines of annotations.
- We annotated 205 library routines – each routine averages about 15 lines of annotations. Notice that the optimizations presented in Chapter 5 require considerably more lines of annotations per routine.
- We used the four dataflow analysis properties described above: `Taint`, `FileState`, `FDKind`, and `Trust`.
- The annotations contain 156 error checks spread over 57 different routines. Note that even though the other 150 routines do not emit errors themselves, they are important to the error detection analyses because many of them generate or convey error states.

6.3.4 Platform

We run all experiments on a Dell OptiPlex GX-400, with a Pentium 4 processor running at 1.7 GHz, and 2 GB of main memory. The machine runs Linux with the 2.4.18 kernel. Our system is implemented entirely in C++ and compiled using the GNU g++ compiler version 3.0.3.

6.4 Results

We find that the Broadway compiler is an effective tool for detecting complex errors and security holes. Figure 6.2 shows number of errors found for each input program and each

Client: **Program**	File Access	FSV	Remote Access	Remote FSV	FTP Behavior
stunnel-3.8	5	1	0	1	0
pfinger-0.7.8	8	1	7	1	0
muh2.05c	6	1	0	1	1
muh2.05d	6	0	0	0	1
pure-ftpd-1.0.15	4	0	18	0	1
fcron-2.9.3	43	0	0	0	0
apache-1.3.12	8	1	6	1	3
make-3.75	2	0	0	0	0
BlackHole-1.0.9	72	0	0	0	5
wu-ftpd-2.6.0	25	7	15	2	4
openssh-3.5p1-client	10	2	1	2	0
privoxy-3.0.0-stable	8	0	0	0	0
wu-ftpd-2.6.2	25	3	26	0	6
bind-4.9.4-REL	419	1	4	1	6
openssh-3.5p1-server	13	0	5	0	0
cfengine-1.5.4	18	9	88	9	9
sqlite-2.7.6	3	0	0	0	0
nn-6.5.6	148	13	41	13	37

Table 6.2: This table summarizes the overall error detection results: the entries show the number of errors reported for each client and each program.

error problem. In particular, our system detects remotely exploitable format string vulnerabilities with high accuracy: Figure 6.3 shows that in most cases it detects all the known errors with no false positives. We generally find that the overall number of errors reported is low, which is important because this number determines how much additional work a programmer would have to perform, first to determine if the errors really exist and then to fix them. The exception to this finding is the file access error, which we discuss later in this section. We also find that the error messages themselves are helpful in locating and fixing bugs because they indicate the place in the source code where the error occurs, and they can provide domain-specific information about why code is erroneous.

6.4.1 File access results

The file access error detection problem stands out because it produces the largest number of error reports. One mitigating factor in these results is that programs contains hundreds or thousands of file accesses, so rate of error reporting is still relatively low. Inspection of the input programs, however, indicates that few of these are actual errors. Manual inspection of the programs and the analysis results indicates that the high false-positive rate is due to conditional branches: file manipulation is often subject to run-time conditions. We have identified two specific situations in which conditional behavior leads to inaccurate analysis of file accesses.

First, several of the programs are daemons that reconnect the standard input and standard output file streams in different ways depending on whether they are run as stand-alone daemons or as part of the centralized Unix Internet services daemon (inetd). As a result, our analyzer often cannot determine whether the standard streams are open or closed. Every subsequent use of these streams, which occurs frequently, reports a possible error. One potential solution to this problem is to indicate at compile time that we want to analyze the daemon in one specific mode.

```
while (not_done) {
    f = fopen(filename, "r");
    if (f) {
        /* -- Safe to read from the file -- */
        ...
    }
    ...
    if (f)
        fclose(f);
    /* -- File is always closed at this point -- */
}
```

Figure 6.8: Checking file accesses produces many false positives because of conditional branches.

Second, calls to `close()` and `fclose()` are often guarded by a null pointer test on the file handle. Because an attempt to open a file can fail, the standard library indicates this failure by returning a null pointer. A properly written C program will test the result of `fopen()` and only use this result when it is not null. Figure 6.8 shows a typical example of code that safely accesses a file. The key to this code fragment is that at the end of the loop body the file is guaranteed to be closed, regardless of whether the `fopen()` succeeded or not: if `fopen()` fails, the file is not open and there is nothing to do; if it succeeds, then `f` is non-null, which triggers the call to `fclose()`. Unfortunately, our analyzer cannot recognize this behavior. The first problem is that we handle control-flow conservatively, by analyzing both branches and merging the information. As a result, it appears to the analyzer that `fopen()` is always called, while `fclose()` is only called some of the time. The second problem is that even if the analyzer could handle the two branches separately, it needs to know that a null file pointer indicates that the file is not open. This information, however, is idiosyncratic to the behavior of `fopen()` and cannot be derived from the program source, no matter how precise the analysis.

```
procedure fopen(path, mode)
{
  on_exit { return -->
                   new file_stream -->
                             new file_handle }
  analyze FileState {
    (return != null) => file_handle <- Open
    (return == null) => file_handle <- Closed
  }
}
```

Figure 6.9: Proposed syntax for an annotation that captures the correlation between special return values and the behavior of the routine. In this example, the file is considered open when the return value is non-null, and it is considered closed when it is null.

We find this conditional behavior in many libraries: special return values indicate the success or failure of a routine and therefore indicate whether or not the analyzer should apply the prescribed state transitions to arguments. Any error detection system that hopes to accurately detect errors in the use of these routines must be provided with the information that correlates special return values with the states of the objects. Our proposed solution to this problem is to provide a modified `analyze` annotation that captures this information. Figure 6.9 shows some possible syntax for encoding the behavior of `fopen()`. The analyzer would carry both potential state transitions until one of guards is tested, in which case the transition takes effect. We refer to this type of dataflow information as a *conditional transfer function*. We leave the implementation and evaluation of these conditional transfer functions as future work.

6.4.2 Format string vulnerability results

Several of the example programs contain known format string vulnerabilities, which allows us to evaluate the accuracy of our system in a way that is not practical for the other errors. We combine our discussion of the general format string vulnerability error with the augmented form that determines remote exploitability.

Figure 6.3 contains a breakdown of the format string vulnerability results for each program. The first column shows the number of known format string vulnerabilities in each program. This information comes primarily from security advisories provided by the Computer Emergency Response Team. (CERT is a center for studying Internet security and is part of Carnegie Mellon's Software Engineering Institute.) The next two columns show the both numbers of FSVs reported by Broadway: the general FSV case and the exploitable FSV case. The fourth column indicates any new errors found by our system that were not previously known. Finally, the last column shows the number of false positives: the number of errors reported by our system that are not actually errors. Since the CERT security advisories report only exploitable vulnerabilities, we obtain the false positive rate by comparing known errors against the number of exploitable FSVs we found.

We find that with one exception, the Apache web server, we find all the known errors with no false positives. The reported error in Apache does appear to be a format string vulnerability, but it is not exploitable for algorithmic reasons that are beyond the scope of our analysis. We do not consider this error to be a false positive in the traditional sense because it is not caused by imprecision or conservative assumptions in the analyzer.

For two of the input programs, `muh` and `wu-ftp`, we obtained two versions of each program: one version known to contain format string vulnerabilities and a subsequent

Program	Known errors	Found		New errors	False positives
		All	Exploitable		
stunnel-3.8	1	1	1	0	0
pfinger-0.7.8	1	1	1	0	0
muh2.05c	1	1	1	0	0
muh2.05d	0	0	0	0	0
pure-ftpd-1.0.15	0	0	0	0	0
fcron-2.9.3	0	0	0	0	0
apache-1.3.12	0	1	1	0	1
make-3.75	0	0	0	0	0
BlackHole-1.0.9	0	0	0	0	0
wu-ftpd-2.6.0	2	7	2	0	0
openssh-3.5p1-client	2	2	2	0	0
privoxy-3.0.0-stable	0	0	0	0	0
wu-ftpd-2.6.2	0	3	0	0	0
bind-4.9.4-REL	1	1	1	0	0
openssh-3.5p1-server	0	0	0	0	0
cfengine-1.5.4	6	9	9	3	0
sqlite-2.7.6	0	0	0	0	0
nn-6.5.6	1	13	13	-	-

Table 6.3: Our results for the format string vulnerability show that our system generates very few, if any, false positives for many of the programs.

version with the bugs fixed. Our system accurately detects the known vulnerabilities in the old versions and certifies the new versions as bug-free. The two versions of `wu-ftp` also show the different between the basic FSV analysis and the exploitable FSV analysis. Of the seven FSVs found in the earlier version, only two are remotely exploitable. The subsequent version fixes the exploitable FSVs, but still contains three errors. While they do not represent security holes, they could cause the program to fail if the configuration file contains improper entries.

The case of `cfengine` shows the benefit of automatic error detection even for simple errors. The three new errors that we found are caused by typos in the program source. The code contains three cases where `sprintf()` is accidentally called without the first argument, which specifies the target buffer. As a result, the arguments are shifted down one place, causing the third argument to be incorrectly interpreted as the format string. This code would fail immediately upon execution, but probably handles a case that occurs rarely, if ever, during normal execution. This example shows how static analysis complements program testing by exploring all execution paths, regardless of their likelihood of use.

6.4.3 Remote access results

Our compiler finds a large number of remote access vulnerabilities in several of the programs. We believe that the main reason for these high numbers is not false positives, but rather the way we formulate the problem. In particular, we report a remote access vulnerability for any file operation in which the file name or handle is untrusted. The news reader `nn`, for example, reports 41 remote access vulnerabilities, but most of them occur in routines like `chdir()` and `unlink()`, which do not have significant security implications. We can reformulate this analysis to focus on more sensitive routines, such as the `exec()` family of system calls. In fact, when we reexamine the results of the analysis, we find that only four of the programs contain calls to `exec()` with untrusted commands: `cfengine`, the two versions of `wu-ftp`, and the `open-ssh-server`. None of these reports is a false positive, but they are not true errors either because they represent the intended behavior of the programs.

6.4.4 FTP behavior results

Detecting this behavior is the most complex problem because it depends on the states of multiple memory locations and multiple client analyses. Nevertheless, our analysis does properly detect exactly those program points in the two FTP daemons that perform the "get" or "put" file transfer functions. For an FTP daemon, however, reading and writing arbitrary files is the intended behavior of the program, so these reports do not represent false positives. The Apache web server reports three places where a remote client can specify files to be read. Like the FTP daemons, we would expect this capability in a web server, and the reports provide a validation of this behavior. Note that file access permissions play an important role in controlling which files an FTP daemon may read, but that modeling this feature of the file system is currently beyond the scope of our analysis.

6.5 Conclusions

In this chapter we demonstrate that the Broadway compiler is an effective tool for discovering a range of significant programming errors. We believe that as programming languages and run-time systems become more robust to low-level attacks, high-level errors, such as library-level errors, will become a more significant class of vulnerabilities. Our system provides both the flexibility to express a range of errors and the analysis power needed to detect them in real programs.

Even for more mundane errors, however, our approach provides a valuable service. A library writer can use the annotations to check for basic errors, such as improper arguments to a library functions, and to make sure that library routines are called in the correct sequences. Compile-time error checking helps eliminate the overhead of run-time checking, and it saves the programmer time by providing meaningful library-specific error messages early in the development lifecycle.

Our error detection results suggest several directions for future research. First, our annotation language is currently geared towards expressing the domain-specific properties of individual objects, such as object state. However, we have only begun to explore capturing more complex relationships between objects, such as relative size or lifetime. These properties would allow us to detect potential resource leaks and exploitable resource mismanagement, such as so-called "fork bombs". Second, it is clear that branch conditions play an important role in some error detection problems. In particular, we believe it is critical to recognize the domain-specific conditional behavior of library routines.

Chapter 7

Client-Driven Pointer Analysis

The analysis requirements for both library-level error detection and library-level optimization place a considerable strain on our dataflow analysis framework. In particular, analyzing the pointer behavior of library routines over entire programs presents a serious challenge for the scalability of our approach. In this chapter we present a new *client-driven* pointer analysis algorithm that automatically adjusts its precision in direct response to the needs of error detection and optimization problems. We evaluate our algorithm on the same C programs and using the same error detection problems from Chapter 6. We compare the accuracy and performance of our algorithm against several commonly-used fixed-precision algorithms. We find that the client-driven approach effectively balances cost and precision, often producing results as accurate as fixed-precision algorithms that are many times more costly. Our algorithm is effective because many client problems only need a small amount of extra precision applied to the right places in each input program.

7.1 Introduction

Pointer analysis is critical for effectively analyzing programs written in languages like C, C++, and Java, which make heavy use of pointers. In Chapters 5 and 6 we argue that many library-level compilation tasks require a precise model of how library routines manipulate pointer-based data structures. Pointer analysis attempts to disambiguate indirect memory references, so that the optimization and error detection passes have a more accurate view of program behavior. In this sense, pointer analysis is not a stand-alone task: its purpose is to provide pointer information to other *client* analyses.

Existing pointer analysis algorithms differ considerably in their precision. Previous research has generally agreed that more precise algorithms are more costly to compute—often significantly more costly—but has disagreed on whether more precise algorithms yield more accurate results, and whether these results are worth the additional cost [83, 77, 52, 38, 74]. Despite these differences, a recent survey found a common view among pointer analysis researchers that the choice of pointer analysis algorithm should be dictated by the needs of the client analyses [50].

```
p = safe_string_copy("Good");       char * safe_str_copy(char * s)
q = safe_string_copy("Bad");        {
r = safe_string_copy("Ugly");         if (s != 0) return strdup(s);
                                        else return 0;
                                    }
```

Figure 7.1: Context-insensitive pointer analysis hurts accuracy, but whether or not that matters depends on the client analysis.

In this chapter we present a new *client-driven* pointer analysis algorithm that addresses this viewpoint directly: it automatically adjusts its precision to match the needs of the client. The key idea is to discover where precision is needed by running a fast initial pass of the client, using a low-precision pointer analysis algorithm. The pointer and client analyses run together in an integrated framework, allowing the client to provide feedback about the quality of the pointer information it receives. Using these initial results, our algorithm constructs a customized precision policy tailored to the needs of the client and input program. This approach is related to demand-driven analysis [53, 49], but solves a different problem: while demand-driven algorithms determine which parts of the analysis need to be computed, client-driven analysis determines which parts need to be computed using more precision.

For example, consider how context-insensitive analysis treats the string copying routine in Figure 7.1: the pointer parameter `s` merges information from all the possible input strings and transfers it to the output string. For a client that associates dataflow facts with string buffers, this could severely hurt accuracy—the appropriate action is to make the routine context-sensitive. However, for a client that is not concerned with strings, the imprecision is irrelevant.

We evaluate our algorithm using the five security vulnerability and error detection problems from Chapter 6 as clients. These clients are demanding analysis problems that stress the capabilities of the pointer analyzer, but with adequate support they can detect significant and complex program defects. We compare our algorithm against five fixed-precision algorithms on the same suite of 18 real C programs. We measure the cost in terms of time and space, and we measure the accuracy simply as the number of errors reported: the analysis is conservative, so fewer error reports always indicates fewer false positives.

This chapter makes the following contributions:

- We present a client-driven pointer analysis algorithm that adapts its precision policy in response to the needs of client analyses. For our error detection clients, this algorithm effectively discovers where to apply more analysis effort in order to reduce the number of false positives, while applying minimal effort to the rest of the program in order to keep the overall cost low.

- We present empirical evidence that different analysis clients benefit from different kinds of precision—flow-sensitivity, context-sensitivity, or both. However, in most cases only a small part of each input program needs additional precision. Our algorithm is effective because it automatically identifies these parts.

- Our results show that whole-program dataflow analysis is an accurate and efficient tool for error detection when it has adequate pointer information.

7.2 Client-driven algorithm

Our client-driven pointer analysis is a two-pass algorithm. The key idea is to use a fast, low-precision pointer analysis in the first pass to discover which parts of the program need more precision. The algorithm uses this information to construct a fine-grained, customized precision policy for the second pass. This approach requires a tighter coupling between the pointer analyzer and the client analyses: in addition to providing memory access information to the client, the pointer analyzer receives feedback from the client about the accuracy of the client flow values. For example, the client analysis can report when a confluence point, such as a control-flow merge or context-insensitive procedure call, adversely affects the accuracy of its analysis. The interface between the pointer analyzer and the client is simple, but it is the core mechanism that allows the framework to tailor its precision for the particular client and target program.

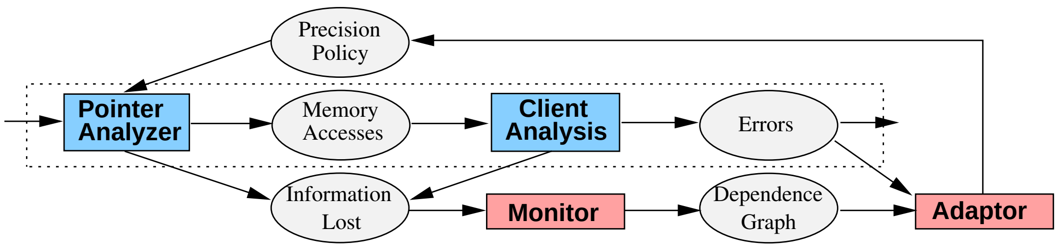

Figure 7.2: Our analysis framework allows client analyses to provide feedback, which drives corrective adjustments to the precision.

Figure 7.2 shows a diagram of our system. The components in the dashed-line box represent the traditional architecture of a program analysis framework: the pointer analyzer provides information about memory accesses to the client analysis. The client uses this information to perform its own analysis and compiler tasks, such as optimization or error detection. Information flows in only one direction, and the pointer analysis typically runs to completion before passing control to the client analysis.

The implementation of our algorithm adds two components to this traditional framework: a *monitor* that detects and tracks loss of information during program analysis, and an *adaptor* that uses the output of the monitor to determine corrective adjustments to the precision. During program analysis, the monitor identifies the places where information is lost, and it uses a dependence graph to track the memory locations that are subsequently affected. When analysis is complete the client takes over and performs its tasks—afterward it reports back to the adaptor with a set of memory locations that are not sufficiently accurate for its purposes. Borrowing terminology from demand-driven analysis, we refer to this set as the *query*. The adaptor starts with the locations in the query and tracks their values back through the dependence graph. The nodes and edges that make up this back-trace indicate which variables and procedures need more precision. The framework reruns the analysis with the customized precision policy.

Even though the algorithm detects information loss during analysis, it waits until the analysis is complete to change precision. One reason for this is pragmatic: our framework cannot change precision during analysis and recompute the results incrementally. There is a more fundamental reason, however: during analysis it is not readily apparent that imprecision detected in a particular pointer value will adversely affect the client later in the program. For example, a program may contain a pointer variable with numerous assignments, causing the points-to set to grow large. However, if the client analysis never needs the value of the pointer then it is not worth expending extra effort to disambiguate it. By waiting to see its impact, we significantly reduce the amount of precision added by the algorithm.

7.2.1 Polluting Assignments

The monitor runs along side the main pointer analysis and client analysis, detecting information loss and recording its effects. Loss of information occurs when conservative assumptions about program behavior force the analyzer to merge flow values. In particular, we are interested in the cases where accurate, but conflicting, information is merged, resulting in an inaccurate value—we refer to this as a *polluting assignment*.

For "may" pointer analysis smaller points-to sets indicate more accurate information—a points-to set of size one is the most accurate. In this case the pointer relationship is unambiguous, and assignments through the pointer allow strong updates [18]. Therefore, a pointer assignment is polluting if it combines one or more unambiguous pointers and produces an ambiguous pointer.

For the client analysis information loss is problem-specific, but we can define it generally in terms of dataflow lattice values. We take the compiler community's view of lattices: higher lattice values represent better analysis information. Lower lattice values are more conservative, with lattice bottom denoting the worst case. Therefore, a client update is polluting if it combines a set of lattice values to produces a lattice value that is lower than any of the individual members.

We classify polluting assignments according to their cause. In our framework there are three ways that conservative analysis can directly cause the loss of information [31]. We will refer to them as *directly polluting assignments*, and they can occur in both the pointer analysis and the client analysis:

- Context-insensitive procedure call: the parameter assignment merged conflicting information from different call sites.
- Flow-insensitive assignment: multiple assignments to a single memory location merge conflicting information.
- Control-flow merge: the SSA phi function merges conflicting information from different control-flow paths.

The current implementation of the algorithm is only concerned with the first two classes. It can detect loss of information at control-flow merges, but it currently has no corrective mechanism, such as node splitting or path sensitivity, to remedy it.

In addition to these classes, there are two kinds of polluting assignments that are caused specifically by ambiguous pointers. These assignments are critical to the client-driven algorithm because they capture the relationship between accuracy in the pointer analysis and accuracy in the client. We refer to them as *indirectly polluting assignments,* and they always refer to the offending pointer:

- Weak access: the right-hand side of the assignment dereferences an ambiguous pointer, which merges conflicting information from the pointer targets.
- Weak update: the left-hand side assigns through an ambiguous pointer, forcing a weak update that loses information.

7.2.2 Monitoring Analysis

During analysis, the monitor detects the five kinds of polluting assignments described above, both for the client analysis and the pointer analysis, and it records this information in a directed dependence graph. The goal of the dependence graph is to capture the effects of polluting assignments on subsequent parts of the program.

Each node in the graph represents a memory location whose analysis information, either points-to set or client flow value, is polluted. The graph contains a node for each location that is modified by a directly polluting assignment, and each node has a label that lists of all the directly polluting assignments to that memory location—for our experiments we only record the parameter passing or flow-insensitive assignment cases. The monitor builds this graph online by adding nodes to the graph and adding assignments to the labels as they are discovered during analysis. These nodes represent the sources of polluted information, and the labels indicate how to fix the imprecision.

The graph contains two types of directed edges. The first type of edge represents an assignment that passes polluted information from one location to another. We refer to this as a *complicit assignment*, and it occurs whenever the memory locations on the right-hand side are already represented in the dependence graph. The monitor creates nodes for the affected left-hand side locations, if necessary, and adds edges from those nodes back to the right-hand side nodes. Note that the direction of the edge is opposite the direction of assignment so that we can trace dependences backward in the program. The second type of edge represents indirectly polluting assignments. The monitor adds nodes for the left-hand side locations and it adds a directed edge from each of these nodes back to the offending pointer variable. This kind of edge is unique to our analysis because it allows our algorithm to distinguish between the following two situations: (1) an unambiguous pointer whose target is polluted, and (2) an ambiguous pointer whose targets have precise information.

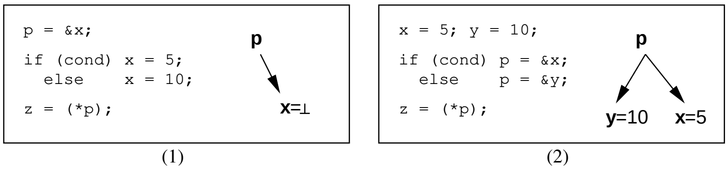

Figure 7.3: Both code fragments assign bottom to `z`: in (1) `x` is responsible, in (2) `p` is responsible.

Figure 7.3 illustrates this distinction using constant propagation as an example client. Both code fragments assign lattice bottom to `z`, but for different reasons. Case (1) is caused by the polluted value of `x`, so the monitor adds an edge in dependence graph from `z` back to `x`. Case (2), however, is caused by the polluted value of the pointer `p`, so the monitor adds an edge from `z` to `p`.

We store the program locations of all assignments, but for performance reasons the monitor dependence graph is fundamentally a flow-insensitive data structure. As a result, the algorithm cannot tell which specific assignments to an memory location affect other location. For example, a location might have multiple polluting assignments, some of which occur later in the program than complicit assignments that read its value. In most cases, this simplification does not noticeably hurt the algorithm, but occasionally it leads to overly aggressive precision, particularly when it involves global variables that are used in many different places and for different purposes.

7.2.3 Diagnosing Information Loss

When analysis is complete, the client has an opportunity to use the results for its purposes, such as checking for error states or applying an optimization. The client provides feedback to the adaptor, in the form of a query, indicating where it needs more accuracy. The adaptor uses the dependence graph to construct a precision policy specifically tailored to obtain the desired accuracy. The output of the adaptor is a set of memory locations that need flow-sensitivity and a set of procedures that need context-sensitivity. The new precision policy applies to both the pointer analysis and the client analysis.

The client query consists of a set of memory locations that have “unsatisfactory” flow values. For example, if the client tests a variable for a particular flow value, but finds lattice bottom, it could add that variable to the query. The goal of the adaptor is to improve the accuracy of the memory locations in the query. The corresponding nodes in the dependence graph serve as a starting point, and the set of nodes reachable from those nodes represents all the memory locations whose inaccuracy directly or indirectly affects the flow values of the query. The key to our algorithm is that this subgraph is typically much smaller than the whole graph—we rarely to need to fix *all* of the polluting assignments.

The adaptor starts at the query nodes in the graph and visits all of the reachable nodes in the graph. It inspects the list of directly polluting assignments labeling each node (if there are any) and determines the appropriate corrective measures: for polluting parameter assignments it adds the corresponding procedure to the set of procedures that need context-sensitivity; for flow-insensitive assignments it adds the corresponding memory location to the set of locations that need flow-sensitivity.

7.2.4 Chaining precision

In addition to addressing each polluting assignment, the adaptor increases precision along the whole path from each polluting assignment back to the original query nodes. When it finds a node that needs flow-sensitivity, it also applies this additional precision to all the nodes back along the path. When it makes a procedure context-sensitive, it also determines the set of procedures that contain all the complicit assignments back along the path, and it adds that set to the context-sensitive set. The motivation for this chaining is to ensure that intermediate locations preserve the additional accuracy provided by fixing polluting assignments.

By aggressively chaining the precision, we also avoid the need for additional analysis passes. The initial pass computes the least precise analysis information and therefore covers all the regions of the program for which more precision might be beneficial. Any polluting assignments detected in later passes would necessarily occur within these regions and thus would already be addressed in the customized precision policy. We validated this design decision empirically: subsequent passes typically discover only spurious instances of imprecision and do not improve the quality of the client analysis.

7.3 Experiments

In this section we present empirical measurements of the client-driven pointer analysis algorithm using five different error detection problems as clients. We compare both the cost and the accuracy of our algorithm against five fixed-precision algorithms. We find that the client-driven algorithm generally improves precision in the right places in each input program, producing results as accurate as the more costly fixed-precision algorithms at a fraction of the cost.

- For clients that require very little precision, the client-driven algorithm performs competitively with the fastest fixed-precision algorithm. In some of these cases, the client-driven algorithm takes more time, but only because it takes two passes. In absolute terms the difference is negligible—for example, six seconds instead of three.

- For clients that require more precision, the client-driven approach often substantially outperforms the comparably accurate fixed-precision algorithm.

We use the same error detection problems and input programs from Chapter 6. These problems represent realistic errors that actually occur in practice and can cause serious damage. Like many error detection problems, they involve data structures, such as buffers and file handles, that are allocated on the heap and manipulated through pointers. The lifetimes of these data structures often cross many procedures, requiring interprocedural analysis to properly model. Thus, they present a considerable challenge for the pointer analyzer.

The five errors we detect are:

- File access error: make sure that files are open when accessed.
- Format string vulnerability (FSV): make sure that format strings do not contain untrusted data.
- Remote access vulnerability: make sure that a remote hacker cannot control sensitive functions, such as execution of other programs.
- Remote FSV: an enhanced version of the format string check that determines when vulnerabilities are remotely exploitable.
- FTP behavior: make sure that the program cannot be tricked into reading and returning the contents of arbitrary local files.

We run all experiments on a Dell OptiPlex GX-400, with a Pentium 4 processor running at 1.7 GHz and 2 GB of main memory. The machine runs Linux with the 2.4.18 kernel. Our system is implemented entirely in C++ and compiled using the GNU g++ compiler version 3.0.3.

7.3.1 Methodology

Our suite of experiments consists of the same 18 C programs and five error detection problems presented in Chapter 6. In this chapter, we run the same experiments under six different pointer analysis algorithms—five fixed-precision pointer algorithms and our client-driven algorithm. For each combination of program, error problem, and pointer analysis algorithm, we run the analysis framework and collect a variety of measurements, including analysis time, memory consumption, and number of errors reported. For the client-driven algorithm, the query contains the memory locations that the error detection client found to be in the error state.

The number of errors reported is the most important of these metrics. The more false positives that an algorithm produces, the more time a programmer must spend sorting through them to find the real errors. Our experience is that this is an extremely tedious and time consuming task. Using a fast inaccurate error detection algorithm is false economy: it trades computer time, which is cheap and plentiful, for programmer time, which is valuable and limited. Our view is that it is preferable to use a more expensive algorithm that can reduce the number of false positives, even if it has to run overnight or over the weekend to do it.

On the other hand, we do not want to consume more computer resources than is necessary to produce accurate results. Therefore, when two algorithms report the same number of errors, we compare them in terms of analysis time and memory consumption.

In some cases, we know the actual number of errors present in the programs. This information comes from security advisories published by organizations such as CERT and SecurityFocus. We have also manually inspected some of the programs to validate the errors.

For the client-driven algorithm we also gather information about the specific precision adjustments that it makes. We record the number of procedures that it makes context-sensitive and the number of memory locations that it makes flow-sensitive.

Mode	Procedures	Pointers	Client
I-I-I	Context-Insensitive	Flow-Insensitive	Flow-Insensitive
I-I-S	Context-Insensitive	Flow-Insensitive	Flow-Sensitive
I-S-S	Context-Insensitive	Flow-Sensitive	Flow-Sensitive
S-I-I	Context-Sensitive	Flow-Insensitive	Flow-Insensitive
S-S-S	Context-Sensitive	Flow-Sensitive	Flow-Sensitive

Table 7.1: The five fixed-precision algorithms we use for comparison.

Unlike previous research on pointer analysis, we do not present data on the points-to set sizes because this metric is not relevant to our algorithm. We could compute the average points-to set size and see if it correlates with better accuracy in the client, but that is beyond the scope of this thesis.

7.3.2 Fixed-precision algorithms

The five fixed-precision algorithms include the four possible combinations of flow-sensitivity and context-sensitivity. In addition, we test a mixed precision algorithm in which the pointer analysis is flow-insensitive but the client analysis can store flow-sensitive information. The intent of this mode is to compare the relative benefit of flow-sensitivity on pointer analysis independent of the flow-sensitivity of the client. The fixed-precision algorithms are shown in Table 7.1.

In some cases we could not obtain results for the higher levels of precision because the analysis takes too long to run, or it runs out of memory.

7.4 Results

We measure the performance of all combinations of pointer analysis algorithms, error detection clients, and input programs—a total of over 500 experiments. We present the results in five graphs, one for each error detection client. Each bar on the graph shows the performance of the different analysis algorithms on the given program. To more easily compare different programs we normalize all execution times to the time of the fastest algorithm

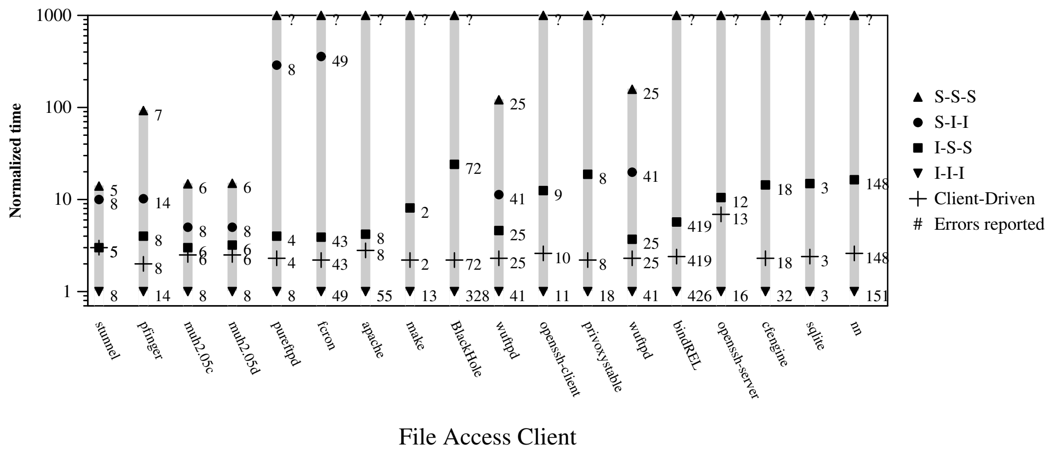

Figure 7.4: Checking file access requires flow-sensitivity but not context-sensitivity. The client-driven algorithm beats the other algorithms because it makes only the file-related objects flow-sensitive.

on that program, which in all cases is the context-insensitive, flow-insensitive algorithm. Therefore, each point on the graph represents a single combination of error detection client, input program, and analysis algorithm. We label each point with the number of errors reported in that combination.

For the 90 combinations of error detection clients and input programs, we find the following:

- **Accuracy:** In 87 out of 90 cases, the client-driven algorithm is as accurate as any of the fixed-precision policies. The other three cases appear to be anomalies, and we believe we can address them.

- **Performance:** In 64 of those 87 cases, the client-driven algorithm equals or beats the performance of the most accurate fixed-precision algorithm. In 29 of these cases the client-driven algorithm is both the fastest ***and*** the most accurate.

- In 19 of the remaining 23 cases the client-driven algorithm performs within a factor of two or three of the best fixed-precision algorithm. In many of these cases the best fixed-precision algorithm is the fastest fixed-precision algorithm, so in absolute terms

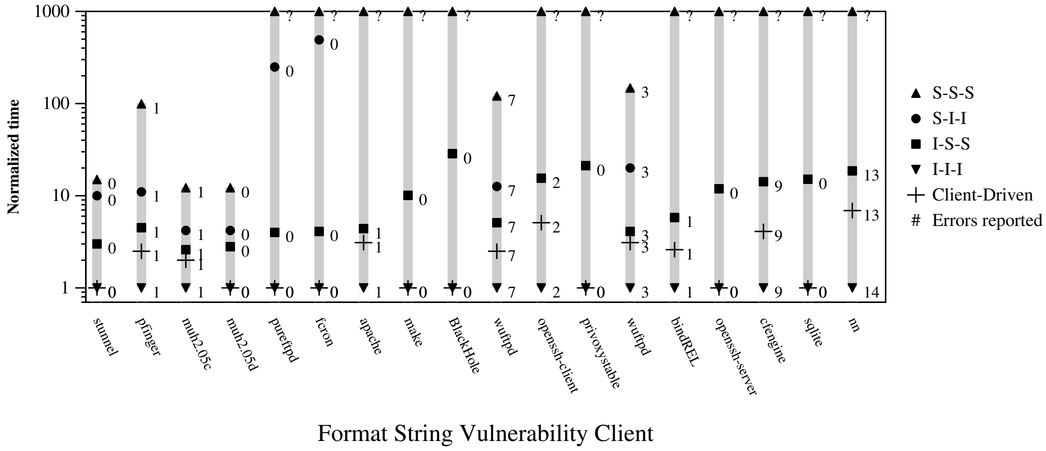

Figure 7.5: Detecting format string vulnerabilities rarely benefits from either flow-sensitivity or context-sensitivity. In many cases, the client-driven algorithm is only slower because it is a two-pass algorithm.

the execution times are all low.

These graphs have two notable omissions. First, for many of the larger programs the fully flow-sensitive and context-sensitive algorithm cannot complete. It either runs out of memory or requires an intolerable amount of time. In these cases we cannot measure the accuracy of this algorithm for comparison. However, we do find that for the smaller programs the client-driven algorithm matches the precision of the full-precision algorithm.

Second, we omit the results for the I-I-S algorithm (context-insensitive, flow-insensitive pointers, flow-sensitive client). This algorithm is neither the fastest nor the most accurate, and this data only serves to clutter the graphs. We interpret this result as compelling evidence that error detection systems cannot ignore pointer analysis.

In general, the only cases where a fixed-policy algorithm performs better than the client-driven algorithm are those in which the client requires little or no extra precision. In particular, the format string vulnerability problem rarely seems to benefit from higher levels of precision. In these cases, though, the analysis is usually so fast that the performance difference is practically irrelevant.

	Added precision:	Number of Procedures set Context-Sensitive				
Program	**Total procedures**	File Access	FSV	Remote Access	Remote FSV	FTP Behavior
stunnel-3.8	42	-	-	-	-	-
pfinger-0.7.8	47	-	-	1	-	-
muh2.05c	84	-	-	-	-	6
muh2.05d	84	-	-	-	-	6
pure-ftpd-1.0.15	116	-	-	2	-	9
fcron-2.9.3	100	-	-	-	-	-
apache-1.3.12	313	-	2	8	2	10
make-3.75	167	-	-	-	-	-
BlackHole-1.0.9	71	-	-	2	-	5
wu-ftpd-2.6.0	183	-	-	-	-	17
openssh-3.5p1-client	441	1	-	10	-	-
privoxy-3.0.0-stable	223	-	-	-	-	5
wu-ftpd-2.6.2	205	-	4	-	4	17
bind-4.9.4-REL	210	-	2	1	1	4
openssh-3.5p1-server	601	1	-	13	-	-
cfengine-1.5.4	421	-	1	4	3	31
sqlite-2.7.6	387	-	-	-	-	-
nn-6.5.6	494	-	1	2	1	30

Table 7.2: This table shows the number of procedures in each program that the client-driven algorithm chooses to analyze using context sensitivity.

Added Precision:	Percentage of Memory Locations set Flow-Sensitive				
Client: **Program**	File Access	FSV	Remote Access	Remote FSV	FTP Behavior
stunnel-3.8	0.20	–	–	–	0.19
pfinger-0.7.8	–	0.53	0.20	0.53	0.61
muh2.05c	0.10	–	–	0.07	0.31
muh2.05d	0.10	–	–	–	0.33
pure-ftpd-1.0.15	0.13	–	0.12	–	0.10
fcron-2.9.3	–	–	0.03	–	0.26
apache-1.3.12	0.18	0.91	0.89	1.07	0.83
make-3.75	0.02	–	–	–	2.19
BlackHole-1.0.9	0.04	–	0.24	–	0.32
wu-ftpd-2.6.0	0.09	0.22	0.34	0.24	0.08
openssh-3.5p1-client	0.06	0.55	0.35	0.56	0.96
privoxy-3.0.0-stable	0.01	–	–	–	0.10
wu-ftpd-2.6.2	0.09	0.51	0.63	0.53	0.23
bind-4.9.4-REL	0.01	0.23	0.14	0.20	0.42
openssh-3.5p1-server	0.59	–	0.49	–	1.19
cfengine-1.5.4	0.04	0.46	0.43	0.48	0.03
sqlite-2.7.6	0.01	–	1.47	–	1.43
nn-6.5.6	0.17	1.99	1.82	2.03	0.97

Table 7.3: This table shows the percent of all memory locations in each program that the client-driven algorithm chooses to analyze using flow sensitivity. We choose to show this value as a percentage because the overall numbers are large.

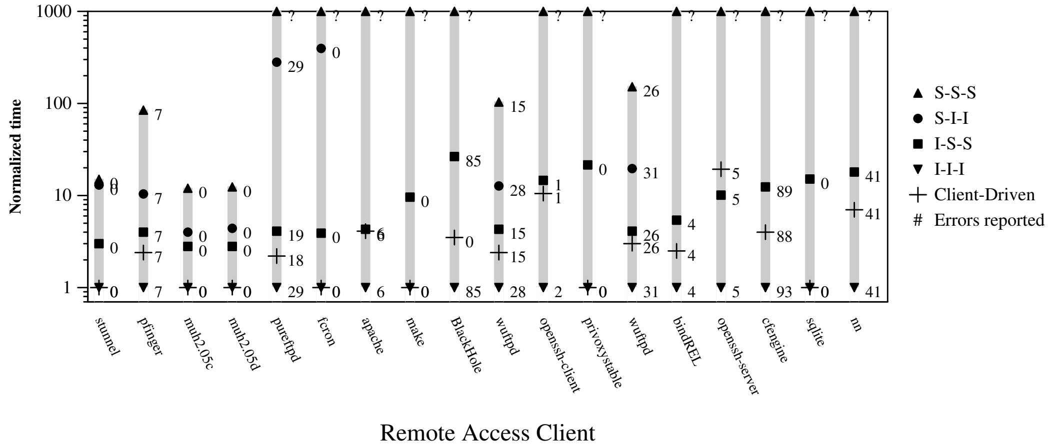

Figure 7.6: Detecting remote access vulnerabilities occasionally requires both flow-sensitivity and context-sensitivity. In these cases the client-driven algorithm is both the most accurate and the most efficient.

For the problems that do require more precision, the client-driven algorithm consistently outperforms the fixed-precision algorithms. Tables 7.2 and 7.3 provide some insight into this result. For each program and each client, we record the number of procedures that the algorithm makes context-sensitive and the percentage of memory locations that it makes flow-sensitive. Looking at the columns, we find that different clients have different precision requirements. The file access client, for example, benefits from some flow-sensitivity but not context-sensitivity; the FTP behavior client requires both. These statistics show that client analyses often need some extra precision, but only a very small amount. In particular, the clients that benefit from context-sensitivity only need a tiny fraction of their procedures analyzed in this way. This result suggests that while faster techniques may exist for implementing context-sensitivity, we can actually avoid it altogether in most cases.

7.4.1 Client-specific results

The client-driven algorithm reveals some significant differences between the precision requirements of the five error detection problems.

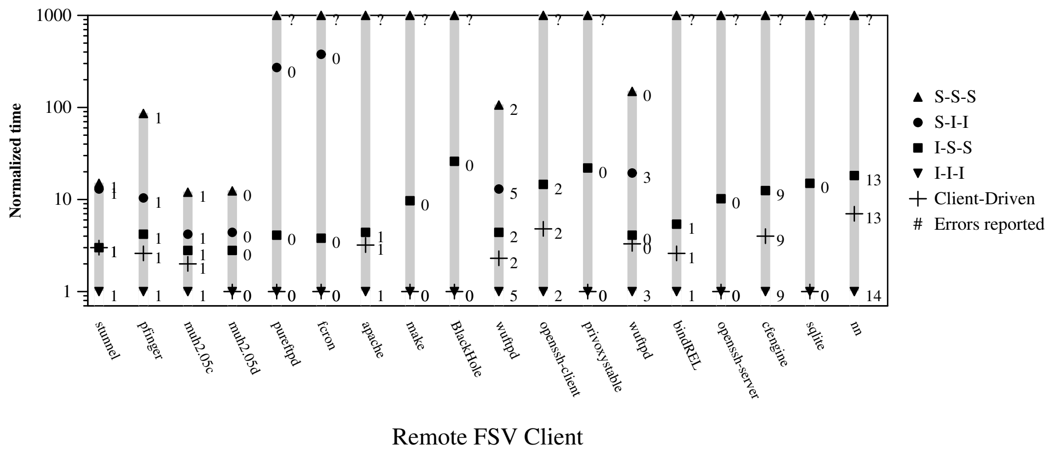

Figure 7.7: Determining when a format string vulnerability is remotely exploitable is a more difficult, and often fruitless, analysis. The client-driven algorithm is still competitive with the fastest fixed-precision algorithm, and it even beats the other algorithms in three of the cases.

File access results

Figure 7.4 shows the complete results for the file state client. File state benefits significantly from flow-sensitivity but not from context-sensitivity. This result makes sense: file state needs flow-sensitivity because it can change. On the other hand, procedures typically cannot be called with file handles either open or closed. We suspect that few of these error reports represent true errors, and we believe that many of the remaining false positives could be eliminated using path-sensitive analysis.

FSV results

Figure 7.5 show the results for detecting format string vulnerabilities. The taintedness analysis that we use to detect format string vulnerabilities generally require no extra precision. We might expect utility functions, such as string copying, to have unrealizable paths that cause spurious errors, but this does not happen in any of our example programs. The high false positive rates observed in previous work [76] is probably due to the use of equality-based analysis.

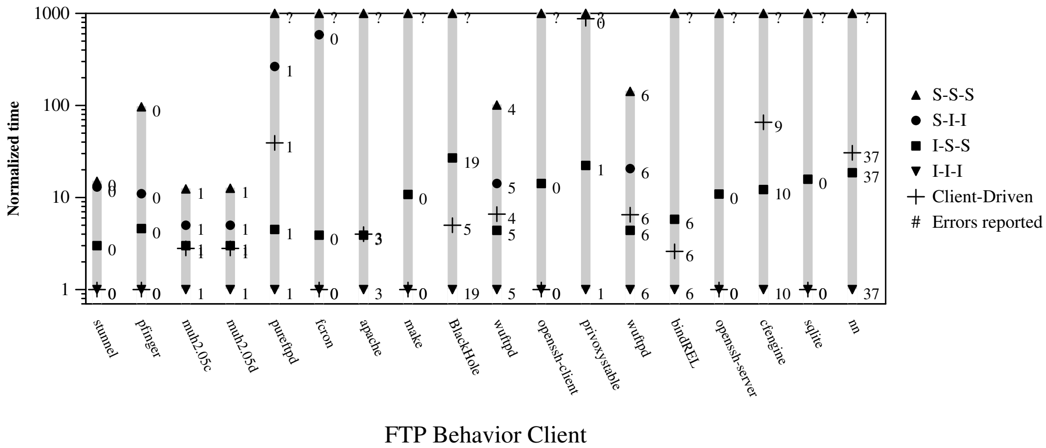

Figure 7.8: Detecting FTP-like behavior is the most challenging analysis. In three cases (WU-FTP, privoxy, and CFEngine) the client-driven algorithm achieves accuracy that we believe only the full-precision algorithm can match—if it were able to run to completion.

Remote access results

Figure 7.6 shows the results for remote access vulnerability detection. Accurate detection of remote access vulnerabilities requires both flow-sensitivity and context-sensitivity because the "domino effect" of the underlying Trust analysis causes information loss to propagate to many parts of the program.

For example, all of the false positives in BlackHole are due to unrealizable paths in a single function called `my_strlcpy()`, which implements string copying. The client-driven algorithm detects the problem and makes the routine context-sensitive, eliminating all of the false positives.

Remote FSV results

Figure 7.7 shows the results for determining the remote exploitability of format string vulnerabilities. FSV exploitability is a much harder problem than its basic counterpart because it distinguishes between different kinds of untrusted data. We have found this client particularly difficult for the client-driven analysis, which tends to add too much precision without lowering the false positive count. Interestingly, many spurious FSV errors are caused by typos in the program: for example, cfengine calls `sprintf()` in several places without providing the string buffer argument.

FTP behavior results

Figure 7.8 shows the results for detecting FTP-like behavior. Detecting this behavior is the most complex problem because it depends on the states of multiple memory locations and multiple client analyses. Context-sensitivity helps eliminate a false positive in one particularly interesting case: in `wu-ftp`, a data transfer function appears to contain an error because the source and target could either be files or sockets. However, when the calling contexts are separated, the combinations that actually occur are file-to-file and socket-to-socket.

7.4.2 Program-specific results

This section describes some of the significant challenges that the input programs present for static analysis.

Function tables

Despite being written in C, some of the programs use tables of function pointers in much the same way that C++ would use virtual function tables. Unfortunately, these tables are indexed by strings, making it practically impossible to reduce the number of possible call targets. As a result the dispatch procedures, which access the table and call through the function pointer, end up significantly polluting the call graph.

Library wrappers

Many of the programs put "wrappers" around standard library functions or provide their own implementations of these functions. A common example is for a program to put a wrapper around `strdup()` that handles a null pointer as input or that exits gracefully if memory is exhausted. The client-driven algorithm works well in these cases because it immediately makes the wrapper functions context-sensitive. However, occasionally there are so many calls to these functions that the cost of context-sensitivity explodes.

Custom memory allocators

A few of the programs use custom memory allocators. Apache is particularly problematic because it implements a region-like allocator with semantics unlike the conventional heap or stack allocation. Luckily, there is an option to compile it using the regular malloc interface. In general, though, many analysis tools rely on the semantics of malloc and free to build an accurate model of heap objects: since multiple calls to malloc always return distinct chunks of memory, there is no need to explicitly model the address space.

7.4.3 Average performance

Figures 7.9 and 7.10 show the performance of the different algorithms averaged over all five clients. In these two graphs we present the actual execution time in seconds and memory usage in megabytes. In most cases the client-driven algorithm performs almost as well as the fastest fixed-policy algorithm—the flow-insensitive context-insensitive algorithm. In the cases where it uses more resources, we often find that it produces a better result: it takes more time, but it eliminates false positives.

7.5 Conclusions and future work

This chapter presents a new client-driven approach to managing the tradeoff between cost and precision in pointer analysis. We show that such an approach is needed: no single fixed-precision analysis is appropriate for all client problems and all programs. The low-precision algorithms do not provide sufficient accuracy for the more challenging client analysis problems, while the high-precision algorithms waste time over-analyzing the easy problems.

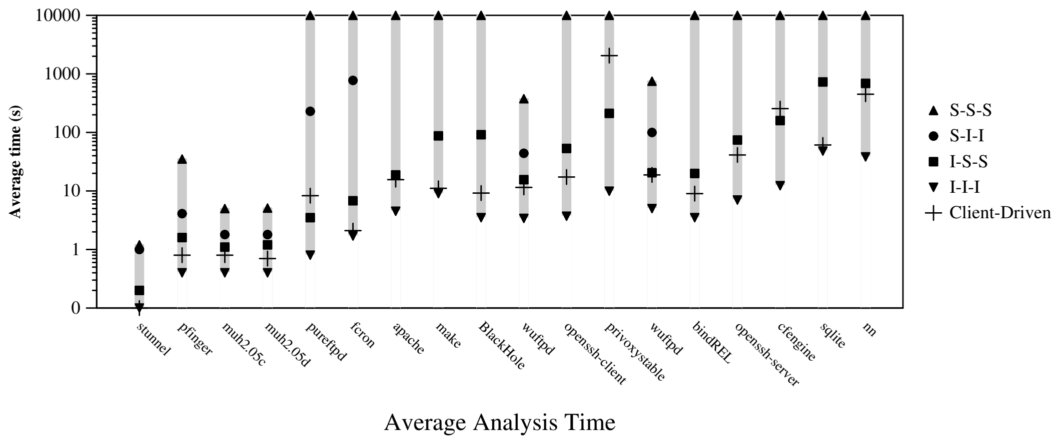

Figure 7.9: The client-driven algorithm performs competitively with the fastest fixed-precision algorithm.

Rather than choose any of these fixed-precision policies, we exploit the fact that many client analyses require only a small amount of extra precision applied to specific places in each input program. Our client-driven algorithm can effectively detect these places and automatically apply the necessary additional precision.

The current implementation of the client-driven algorithm manages two aspects of precision: flow-sensitivity and context-sensitivity. However, the approach could be extended to include other algorithm features. Combining the algorithm with others could yield further improvements in accuracy and scalability:

- More precise algorithms exist for handling control-flow and for modeling heap objects. While we can detect information loss in these situations, we currently have no mechanism to address them. For example, we could use path-sensitive techniques when the algorithm detects information loss at a control-flow merge. Similarly, we could employ shape analysis for heap objects that merge information.

- We can further improve scalability by starting with an even less precise initial pass. For example, equality-based pointer analysis can scale to programs with a million lines of code [81], but it produces significantly less accurate results [51]. We could

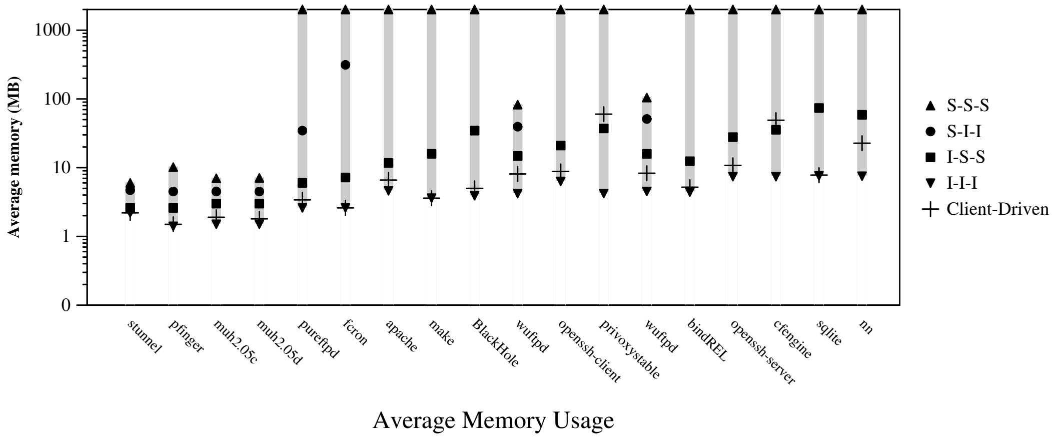

Figure 7.10: Memory usage is only a significant problem for the fully context-sensitivity algorithms. More efficient implementations exist, but we find that full context-sensitivity is not needed.

start our client-driven algorithm with this level of precision; when we detect a unification that causes of information loss, we force particular assignments to be treated as uni-directional.

Chapter 8

Related work

In this chapter we review previous research related to our work. The Broadway compiler represents contributions in a number of different areas of compiler research and at different levels of its design and implementation. Therefore, we divide the related work according to the following categories:

- **Configurable compilers.** Broadway belongs to a class of compilers that can be extended or configured to perform user-defined compilation tasks. The focus of this section is on the specific mechanisms for adding information into the compilation process. In our discussion of these systems, we compare the base facilities of the compiler, the method and ease of extension, and the degree of configurability. We argue that Broadway offers a better balance of power and usability than previous approaches: the annotation language provides a concise and user-friendly interface to the sophisticated compiler mechanisms in the implementation.

- **High-level optimization.** Research in both the compiler community and the high-performance computing community has recognized both the need and the opportunity to raise the level of abstraction of compiler optimizations. In this section we discuss the effectiveness of various existing approaches in supporting high-level optimizations, and in particular we focus on techniques for improving the performance of libraries. We find that previous work has found some success by targeting specific narrow domains, such as matrix computations or communication protocol layers. More general approaches, however, are largely speculative, with few fully functioning systems and hardly any automated results. We believe that Broadway is the first system to offer a complete working solution, effectively extending modern compiler support to a variety of domains.

- **Automatic error checking.** Research in error checking and program verification has a long history in the formal methods community. However, our approach is more closely related to recent work in partial program checking, which has achieved considerable success by using traditional compiler tools, such as type systems and dataflow analysis, to detect specific classes of errors. Broadway is unique among these in several respects: it is easier than most to configure for new kinds of errors, it uses a more powerful form of pointer analysis, and it supports multiple analysis precision policies.

- **Scalable program analysis.** Recent work in program analysis has focused on finding algorithms that scale to hundreds of thousands or millions of lines of code. The problem is particularly acute for interprocedural and whole-program analyses, such as pointer analysis and error detection. Previous work has explored many different approaches: using cheaper, but less precise algorithms, memoizing analysis results, and reducing the problem size through abstraction. We present a new approach that controls the cost of analysis by adaptively tuning precision at very fine grain. This algorithm is complementary to the memoization and abstraction techniques, and could be combined with them to further improve the scalability of analysis.

Our work also builds on a considerable body of previous research in both program analysis and optimization. The Broadway compiler employs many traditional compiler algorithms and optimizations [67, 3]. Our user-defined dataflow analyses are essentially abstract interpretations of the program using the augmented library semantics as defined in the annotations [23, 55]. The pointer analysis algorithm is based on the storage shape graph described by Chase et al. [18] and includes some extensions and improvements from Wilson et al. [96]. Our library-level optimization work is closely related to partial evaluation [10, 11, 70], which improves performance by specializing routines for specific inputs. Partial evaluation combines inlining, constant propagation and constant folding to evaluate as much of the program as possible at compile time. We also use techniques from hygienic macro expansion to implement the user-defined code substitutions [62].

8.1 Configurable compilers

Traditional compilers are typically monolithic programs, with a fixed set of capabilities. Previous research has explored a number of different ways of making compilers more flexible, and often this work is motivated by the same goals as Broadway: to support high-level and domain-specific compilation. These systems consist of a configurable compiler, compiler framework, or compiler generator, and a means for users to specify new compiler behavior. They generally differ from Broadway in two important respects.

First, existing approaches take a considerably different view of the tradeoff between usability of the system and power of expression. In particular, they often provide a more comprehensive set of tools for defining new compiler components, but using these tools is extremely difficult and error prone. For example, the Magik compiler system [36] provides a C interface directly to the compiler internals. A programmer can use this interface to manipulate the target program in practically any way. However, even simple transformations require a considerable amount of compiler coding. Our view is that such an approach is not viable for domain-specific compilation because it requires both domain expertise and compiler expertise.

Second, many existing approaches lack advanced program analysis features, such as pointer analysis or whole-program analysis, and they frequently provide no systematic program analysis at all. This design choice is particularly ironic for systems that allow direct access to the compiler internals: they support a panoply of complex code transformations, but have no information about when and where to apply them. Our view is that rigorous and comprehensive program analysis is a necessary foundation for high-level compilation.

8.1.1 Open and extensible compilers

Open and extensible compilers give the programmer complete access to the internal representation of the program [46, 36]. Our own C-Breeze C compiler infrastructure is an example of such a system. While these systems are quite general, they impose a considerable burden. To use them, the programmer needs to understand (1) general compiler implementation techniques, (2) how to configure the specific compiler they are using, and (3) how to express and execute their optimizations. These systems are ideal for compiler research or for implementing higher level compiler facilities. However, they are unsuitable for use by non-compiler experts.

8.1.2 Meta-programming

Meta-programming systems are similar to open and extensible compilers, except that they provide compile-time program manipulation capabilities from within the programming language itself. Meta-object protocols [20, 59] provide an object-oriented interface for programmers to customize the behavior of a programming language. In these systems, information about the program, such as data types and operators, is represented explicitly in the language itself. The downside of meta-object protocols is that, like open and extensible compilers, the meta-programming is often complex and error prone. For example, to generate an expression the programmer calls a series of methods that instantiate each variable and operator and then assembles them into an abstract syntax tree.

Expression templates and template meta-programming [88] support a limited form of meta-programming using C++ templates. The template instantiation mechanism allows compile-time computations, including a way to represent and manipulate abstract syntax trees. This mechanism is powerful, but can be awkward and overly verbose in practice.

Programmable syntax macros [93] provide a more limited form of program transformation based on compile-time macros. There are two advantages of this approach over meta-object protocols. First, syntax macros, as their name implies, can introduce new syntax into the language. This capability requires a more sophisticated configurable parser, but it allows the meta-programmer to present a more natural interface to language extensions. Second, the macro bodies consist of code fragment templates, which the compiler expands at each macro use. Code templates are more concise than the procedural construction used in meta-object protocols, and the compiler can check them to make sure they generate syntactically correct code.

While these systems can support some forms of library-level optimization, they differ from our approach in several important ways. First, we focus on optimizing existing programs, rather than requiring the programmer to rewrite their code in a form that is suitable for meta-compilation. Second, none of these meta-programming systems include advanced program analysis. Our system drives program transformations using information from dataflow analysis. Dataflow analysis captures deep semantic information about how programs work and operates on a larger scope than typical meta-programs. Finally, our system supports traditional optimizations that are difficult to express in meta-programming systems, such as dead-code elimination and other optimizations based on data dependence information.

8.1.3 Software generators

Software generators [79, 80, 25, 82] are systems for automatically generating programs from high-level specifications. These systems often use meta-programming techniques, like those described above, to assemble reusable software components into complete programs. The reusable components are often represented as parameterized types, which the application programmer composes into complex type expressions. The expansion of these type expressions provides several opportunities to customize and optimize the generated code. First, techniques from partial evaluation take advantage of fixed parameters, such as the sizes of objects, to inline and simplify code. Second, many systems provide a way to recognize and exploit particular patterns in the type expressions. Finally, some systems include features, such as attribute grammars, that support more general analysis of the type expressions.

The most significant difference between the software generator approach and the Broadway approach is the view presented to programmers. Software generators change the way programmers work by presenting a higher level view of software design and development. In these systems, domain-specific optimizations occur during the process of generating a concrete program from the high-level specification. Our approach starts with a concrete program and uses the library interface as a way to infer high-level properties, which in turn enable domain-specific optimizations. We believe that our approach is complementary to software generation. In particular, our dataflow analysis capabilities could be used to find optimization opportunities across software components that are not apparent until the type expressions are instantiated. In addition, our technique of extending traditional optimizations to non-primitives could help simplify many type expressions. For example, a set of components has to define a separate attribute for each computation that is potentially redundant, and each component must test the attribute to decide whether or not to generate code for that computation. In the Broadway approach, we can define a single notion of redundant computation that applies to all computations at all levels of abstraction.

Aspect-oriented programming [60] is a particular approach to software generators with the goal of capturing the independent properties (or "aspects") of a software system in separate specifications, even if those properties pervade the actual implementation of the program in a non-local way. For example, a programmer could provide a single specification of the locking characteristics of a program even if locking occurs in many places throughout the implementation. Our approach can be viewed as an instance of aspect-oriented programming: each annotation file captures the aspect represented by the library.

8.1.4 Optimizer generators and analyzer generators

Previous work includes a wide range of sophisticated systems for specifying program analysis passes and program optimizations. The goal of the Broadway compiler is not to provide an alternative to these systems or even to compete with them. These systems are comprehensive tools intended for use by compiler researchers and compiler implementers. Our focus is on providing a useful subset of these capabilities that can be expressed in a simple declarative manner for use by library experts.

The Genesis optimizer generator produces a compiler optimization pass from a declarative specification of the optimization [95]. Like Broadway, the specification uses patterns, conditions and actions. However, Genesis targets classical loop optimizations for parallelization, so it provides no way to define new program analyses. Conversely, the PAG system is a completely configurable program analyzer [66] that uses an ML-like language to specify the flow value lattices and transfer functions. While powerful, the specification is low-level and requires an intimate knowledge of the underlying mathematics. It does not include support for actual optimizations.

Sharlit [85] provides a unified system for defining dataflow analysis passes and optimizations. The Sharlit user specifies how the program is to be interpreted, the flow value and flow functions of the analysis, and the actions to take based on analysis information. From this specification it generates an efficient compiler pass implementation for the SUIF compiler system. For flexibility and efficiency, the specification is written in C++ and uses a direct interface to the compiler information. As a result, however, this approach imposes many of the same burdens on the programmer as the open and extensible compiler approach.

A number of systems have been developed using program rewriting as a means to specify program transformations [14, 5, 64]. Rewrite rules consist of a left-hand side code pattern, a right-hand side code replacement, and an optional condition that guards the transformation. Systems based on rewriting vary considerably in the power of the code patterns and the kinds of conditions that they support. Some approaches focus more on providing sophisticated code patterns but lack the program analysis capabilities to support complex conditions.

8.1.5 Compiler hints and pragmas

Compilers have long used hints and pragmas to guide optimizations such as register allocation and inlining, and to summarize procedure information such as whether a function has side effects. More recently, annotations have been used to guide dynamic compilation [41]. While annotations are not new, our use of them is new. First, our annotations describe function implementations, rather than call site-specific information. This means that application programs do not require annotations, so our annotations are hidden from the everyday user. Second, and more fundamentally, our advanced annotations can convey domain-specific information that other languages cannot. For example, annotators can define concepts, such as data distribution, that extend beyond those of the base language.

8.2 High-level compilation

The Broadway compiler raises the level of abstraction of compilation by recognizing and exploiting the semantics of libraries. It is useful to view Broadway as a specific instance of a system for supporting *active libraries* [90]. Active libraries represent a broad class of reusable software components that, unlike traditional libraries, are actively involved in the compilation process. In this section, we review previous work on high-level compilation that we view as alternative approaches to supporting active libraries.

Two overall features distinguish our system from previous systems. First, we focus on providing compiler support for existing software without requiring any changes to the application code. Our annotation language allows existing libraries to become active libraries without any change to the implementation of the library or to existing programs. Second, we emphasize the role of program analysis in determining how to customize and optimize software libraries. Dataflow analysis, in particular, provides several advantages over other kinds of information, such as types: it collects information over a larger scope of the program, and it enables many traditional dependence-based optimizations.

8.2.1 Template libraries

Several numeric libraries use the meta-programming capabilities of C++ templates to improve performance. For example, the Blitz++ library [89] uses template expressions to optimize sequences of array operations. The library uses template types to encode a number of common optimizations, such as tiling and loop unrolling. This approach yields significant performance improvements over static libraries. This approach, however, has several down sides. First, it requires the programmer to rewrite applications in C++ using the template library. Second, the implementation of these template libraries is a complex task that requires considerable expertise in the semantics of C++ templates. Developing a template library for a new domain could require a significant effort. Finally, the quality of the resulting code depends heavily on the underlying C++ compiler. Although the C++ standard specifies the basic capabilities of templates, some compilers may optimize the resulting expressions more aggressively then others.

8.2.2 Self-tuning libraries

Several libraries have adopted an approach that uses run-time exploration rather than compile-time analysis to guide optimizations. ATLAS [94] and PhiPAC [12] are two examples of self-tuning libraries for linear algebra. These systems abandon the notion of making performance decisions based on static analysis. Instead, these libraries consist of sets of configurable software components and a run-time test suite that measures performance to determine the specific parameters to these components. This approach differs from ours in several ways, but it may prove to be a complementary technique. First, we rely entirely on static information for optimization, which enables a different class of optimizations than dynamic information. It might be useful, however, to provide both options: when the compiler cannot determine a program property statically, because of conservative assumptions, it could instrument the program to collect and use the information at run-time. Second, these systems focus on customizing the library for particular hardware platforms. In the future, we would like to include machine-specific optimizations in Broadway. Finally, these systems do not provide a general solution for improving library performance. They are typically hard-coded for particular domains and particular performance models.

8.2.3 Telescoping languages

Telescoping languages [58] is an approach to active libraries that shares many goals and ideas with our research. It is currently in the early stages of development, and it is not clear at this time exactly what features it will support. A recent proposal [19], however, describes an annotation language similar to ours that captures properties of library interfaces. Many of the optimization goals appear similar, such as extending traditional optimizations to library interfaces. Unlike our compiler, the proposed system will also include a scripting language that guides the compilation process. The goal of the scripting language is to improve the performance of the compilation process itself, possibly for enabling efficient run-time specialization.

8.2.4 Ad hoc domain-specific compilers

Previous work includes a number of ad-hoc compilers for specific domains. Examples of these systems include Matrix++ [22] for optimizing matrix operations based on a specification of the matrix structure, the Falcon compiler for MatLab [29], and the FFTW [40] system for generating fast Fourier transform implementations. The advantage of building an ad-hoc compilers is that by focusing on a single domain, the compiler writer can employ the best representation and the most aggressive optimizations for that domain. As a result, these compilers often yield higher performance than more general approaches. The downside, however, is that developing a new compiler for each domain is a monumental task, even for a compiler expert. Furthermore, this approach offers no way to share common analysis and optimization facilities among compilers. Our approach is to offer some of the capabilities of an ad-hoc compiler, but at a significantly lower cost to the domain expert.

8.2.5 Formal approaches

There has been considerable work in formal semantics and formal specifications. In particular, Vandevoorde uses powerful analysis and inference capabilities to specialize procedure implementations [87]. However, complete axiomatic theories are difficult to write and do not exist for many domains. In addition, this approach depends on theorem provers, which are computationally intensive and only partially automated. Our work differs from these primarily in the scope and completeness of our annotations, which describe only specific implementation properties instead of complete behaviors.

8.3 Error checking

Compilers have always performed error checking of some sort. However, these checks have traditionally been limited to the semantics of the base programming language. Recent work has attempted to extend error checking to high-level semantics that are not built into the programming language. A wide range of approaches have been explored for automatically detecting programming errors. We find it useful to compare these approach in terms of four characteristics:

- *Kind of specification.* Most error checking approaches include some sort of specification that defines the errors or requirements of the program. The specification is probably the most important distinguishing feature of an error checker because it determines the kinds of detectable errors. For example, an approach that uses a full formal specification can check for complete program correctness. Other kinds of specifications include program annotations, type annotations, language extensions, finite state machines, and dataflow analysis.

- *Analysis engine.* The choice of specification often drives the kind of analysis engine needed. For example, using a complete formal specification typically requires a theorem prover for verification. However, there are often alternative implementations. For example, we can use a constraint solver to perform many of the same program analysis tasks as a dataflow analysis engine by translating the dataflow equations into a system of constraints. Other kinds of analysis engines include simple lexical scanners, parse tree checkers, and many variants of the dataflow analysis engine.

- *Analysis precision.* All static program analysis algorithms are approximate with respect to the actual behavior of programs. The reason is that in almost all cases it is not possible to enumerate every potential execution of a program. At a more practical level, we would like program analysis to complete in a reasonable amount of time. Therefore, we formulate analysis problems as approximations that elide some details of the program state or path of execution. In return for this loss of precision we get a faster analyzer. Previous work has explored a variety of design points on the precision spectrum. However, the most common characterization of a particular approach is its sensitivity to different kinds of control flow: flow sensitivity, context sensitivity, and path sensitivity.

- *Source language.* The choice of source language affects the kinds of software that a system can check. Many systems accept unmodified source code written in commonly used procedural or object-oriented languages, such as C, C++, Java. Other systems accept particular subsets of these languages. Some systems, however, use specialized languages or augment existing languages to include explicit support for error detection.

We can characterize our approach as follows: The Broadway compiler uses a simple declarative specification that focuses on capturing particular domain-specific properties rather than expressing complete correctness. Our analysis engine is an iterative whole-program dataflow analysis framework and we offer a variety of precision policies, including our own client-driven algorithm. Our compiler accepts programs written in unmodified ANSI C.

8.3.1 Formal verification

The goal of formal program verification is to guarantee program correctness by proving that an implementation conforms to its specification. Theorem proving [57] and model checking [34] are two approaches to formal verification. The problem with formal verification is that it currently requires an extreme level of effort by both the programmer, who has to formalize the expected behavior of the program, and by the verification tools. In the future, we hope that formal verification becomes a practical tool for software development. Until that time, however, our approach represents an intermediate solution that effectively verifies specific program properties at a reasonable cost.

8.3.2 Type systems

Two recent papers have focused on using type systems to check for programming errors. Shankar et al. present a system for detecting format string vulnerabilities using type inference [76]. In this approach, two new type qualifiers, "tainted" and "untainted," are introduced to the C language and added to the signatures of the standard C library functions. Type inference is performed by an extensible type qualifier framework, which derives a consistent assignment of these type qualifiers to string variables. Errors are reported as type conflicts. While this system is extremely efficient, it can produce a large number of false positives because the precision of the analysis is low. Initially, we believed that the inaccuracy was due to flow-insensitivity. However, our experiments show that flow-sensitivity is not necessary for the programs presented in their paper. Instead, the high false-positive rate is due to the equality-based constraint solver, which allows information to propagate the wrong way through assignments; in particular, from formal parameters back to actual parameters. In several cases the authors address this problem by manually adding context-sensitivity ("polymorphism" in the type-system terminology).

Foster et al. extend Shankar's work to flow-sensitive type qualifiers, which they use to check the state of file handles and detect double-locking bugs [39]. Unfortunately, imprecision in the store model and equality-based constraints continue to hamper this approach. The authors add two features to their system to help improve accuracy. First, they introduce a new keyword "restrict", which they manually add to the application code to disambiguate memory locations. Second, they add a very limited local form of path-sensitivity to handle the failure case of `fopen()` (when it returns a null pointer.) Our experiments show that these two features are as important, if not more important than the flow-sensitivity itself. In addition, the system generates many spurious errors, apparently due to the lack of parametric polymorphism for store locations. However, our experiments show that context-sensitivity has very little effect on these error checking problems: procedures are rarely called with open files in one context and closed files in another.

8.3.3 Dataflow analysis

While there has been considerable research on formal program verification, our work and much of the recent research on error checking builds on the notion of typestate analysis introduced by Strom and Yemini [84]. These systems differ primarily in the kind of program analysis they use to derive the typestate information, and research has focused primarily on improving the performance and accuracy of the typestate analysis engine. However, one of the major challenges in checking C programs is constructing a precise enough model of the store to support accurate error checking. Unfortunately, many of the techniques used to speed up typestate analysis do not work for pointer analysis. Previous work has generally settled for a low-cost, fixed-policy pointer analysis that provides minimal store information without overwhelming the error checking analysis. This analysis by itself is often inadequate, requiring manual intervention to disambiguate memory locations [27].

ESP

The ESP system implements a path-sensitive variation of typestate analysis that significantly improves precision [27]. In particular, it uses a theorem prover to detect correlations between different branches, and it eliminates many paths that cannot actually occur. The implementation runs in polynomial time by employing a dataflow analysis algorithm due to Reps et al. [72] that can efficiently summarize any dataflow problem that falls into a certain class of problems. However, this class does not include pointer analysis or constant propagation (with constant folding.) Therefore, the authors add a fast flow-insensitive, context-insensitive pointer analyzer as a front-end [26]. Unfortunately, the resulting store model is not precise enough to allow verification, and the authors must manually clone two procedures in order to disambiguate memory locations. Our algorithm, while not as powerful, detects these situations and automatically makes the procedures context-sensitive.

MC

The MC system checks for errors in operating system code using programmer-written checkers based on state machines [35]. A checker consists of a set of states and a set of syntax patterns that trigger transitions on the state machine. The compiler pushes the state machine down each path in the program and reports any error states that it encounters. While this approach has proven quite successful in finding errors, it has limitations. In particular, since the analysis is syntax-driven, the compiler lacks deep information about the program semantics, such as dataflow dependences and pointer relationships. In fact, the system is not sound: it can produce “false negatives” in which programs that have errors are reported as bug-free.

SLAM Toolkit

The SLAM Toolkit approach is similar to MC but is more rigorous and more powerful [8]. SLAM includes a pointer analyzer and can check programs interprocedurally. The toolkit first generates an abstraction of the program that represents its behavior only with respect to the properties of interest. It then uses a model checker to perform path-sensitive analysis on the abstracted program. However, this system still uses a fixed-policy pointer analysis to generate the initial program abstraction, including the store model. While our analysis engine is not as powerful, we allow the error checking problems themselves to dictate the precision of the store model. The SLAM approach could be combined with our technique to improve the initial abstraction or to control analysis precision during the iterative refinement process itself.

8.3.4 Bandera and FLAVERS

The Bandera Tool Set [47] and the FLAVERS system [21] represent a more aggressive approach to verifying finite state properties of programs, particularly for programs with concurrency. Like the SLAM Toolkit, these system specify safety properties using temporal logic or finite state automata, and they use model checking to verify these properties in programs. Research on this approach focuses on how to model the concurrency in programs and on techniques for improving the model checking. These systems can check deep and significant properties when they are provided with an adequate abstraction of a program’s behavior. As we have shown in this thesis, extracting this information from real programs proves to be a challenging task in itself. We believe that while our approach to error checking is less powerful than these systems, we have addressed important issues in the analysis of production software, such as pointer analysis and the use of libraries.

8.3.5 Languages

Previous work has explored the use of programming language support for explicitly expressing correctness constraints. Vault [28] is a new programming language that provides a way to express constraints on the use of domain-specific resources, such as files and locks. It uses *type guards* to control when a particular operation is valid for a resource. Cyclone [54] is a variant of C that includes special types to represent similar constraints.

These language-based approaches differ from ours in several ways. First, they often require the programmers to rewrite part or all of their existing programs in the new language. Second, these systems typically use type inference as the program analysis engine. Vault avoids some of the problems of type qualifiers by introducing *keys* that track flow-sensitive conditions. However, the system cannot reconcile conflicting conditions at control-flow merge points. Our approach avoids this problem by using lattices to represent these error conditions, which allows us to specify precisely how information should be combined at merge points. One of the advantages of Cyclone is that it inserts run-time checks in the places where static analysis fails.

In addition to full-fledged languages, there are several systems that use programmer-supplied annotations to assist in automatic error detection. Examples of such systems include extended static checking [65], role analysis [63], design-driven compilation [75], and LCLint [37]. The advantage of these systems is that they require relatively little extra information from the programmer. Our approach, however, avoids requiring any information from the application programmer, and instead it relies on annotations from the library writer.

8.4 Program analysis: cost and precision

Previous work in various kinds of program analysis, including pointer analysis, have explored ways to reduce the cost of analysis while still producing an accurate result. Our client-driven algorithm addresses this problem specifically for the precision of pointer analysis. It is closely related to demand-driven algorithms and mixed-precision analyses. We also describe recent related work in error detection, focusing on the role of pointer analysis.

8.4.1 Iterative flow analysis

Iterative flow analysis [69] is the only other algorithm that we are aware of that adjusts its precision automatically in response to the quality of the results. Plevyak and Chien use this algorithm to determine the concrete types of objects in programs written using the Concurrent Aggregates object-oriented language. When imprecision in the analysis causes a type conflict, the algorithm can perform *function splitting*, which provides context-sensitivity, or *data splitting*, which divides object creation sites so that a single site can generate objects of different types. The basic mechanism is similar to ours, but it differs in important ways. First, since the type of an object cannot change, iterative flow analysis does not require flow-sensitivity. By contrast, our approach supports a larger class of client analyses, known as *typestate* problems [84], which include flow-sensitive problems. More significantly, our algorithm manages the precision of both the client and the pointer analysis, allowing it to detect when pointer aliasing is the cause of information loss.

8.4.2 Demand-driven pointer analysis

Demand-driven pointer analysis [49] addresses the cost of pointer analysis by computing just enough information to determine the points-to sets for a specific subset of the program variables. Client-driven pointer analysis is similar in the sense that it is driven by a specific query into the results. However, the two algorithms use this information to manage different aspects of the algorithm. Client-driven analysis dynamically varies precision, but still computes an exhaustive solution. Demand-driven pointer analysis is a fixed-precision analysis that computes only the necessary part of the solution. The two ideas are complementary and could be combined to obtain the benefits of both.

8.4.3 Demand interprocedural dataflow analysis

Demand interprocedural dataflow analysis [53] also avoids the cost of exhaustive program analysis by focusing on computing specific dataflow facts. This algorithm produces precise results in polynomial time for a class of dataflow analyses problems called IFDS—interprocedural, finite, distributive, subset problems. However, this class does not include pointer analysis, particularly when it supports strong updates (which removes the distributive property). Program analysis systems based on this algorithm, such as ESP [27], rely on a separate pointer analysis phase [26].

8.4.4 Combined pointer analysis

Combined pointer analysis [97] uses different pointer algorithms on different parts of the program. This technique divides the assignments in a program into classes and uses a heuristic to choose different pointer analysis algorithms for the different classes. Zhang et al. evaluate this algorithm by measuring the number of possible objects accessed or modified at pointer dereferences. Our client-driven pointer analysis is more feedback-directed: instead of using a heuristic, it interacts directly with the client analyses.

8.4.5 Measuring the precision of pointer analysis

A number of previous papers have compared different pointer analysis algorithms, using both direct measurements (sizes of computed points-to sets) and indirect measurements (transitive effects on subsequent analyses) [74, 77, 38, 51, 52]. We find that the average points-to set size is not a good measure of the analysis because it treats all pointers as equal. For example, one algorithm might be more accurate than another by reducing the points-to set of a single variable by one pointer. While the overall metric is hardly affected, that one variable could be the critical distinction for the client. We also find that error detection is more demanding than the clients used in previous studies: the transitive benefits of higher precision are more apparent for our clients.

Chapter 9

Conclusions

Developing high-quality software that performs well and is free of bugs continues to be a significant challenge. While previous research has produced a wide array of automated approaches to improving software, few of these ideas have been incorporated into programming practice. Part of the problem is that programming tools, such as optimizing compilers and software checkers, need more information about what programs do in order to provide better optimization and error detection services. Existing systems have focused almost entirely on obtaining this information directly from application programmers by requiring them to use new mechanisms, such as extended type systems, programming language extensions, and specification languages. Our observation is that by using software libraries, programmers are already providing a wealth of domain-specific information. By capturing and codifying this information, we can significantly improve the quality of compilation without requiring any changes to existing programs or existing programming practices.

9.1 Contributions

In this thesis we have presented a new approach to compilation, called library-level compilation, which takes advantage of the domain-specific information associated with library interfaces. We have presented our library-level compiler, the Broadway compiler, which emphasizes a practical and beneficial separation of concerns:

- Our compiler leverages a significant body of proven compiler techniques, such as dataflow analysis, pointer analysis, and traditional compiler optimizations. It works on unmodified, industrial-strength software systems, with all the nuances and complexities of production code.

- We provide a separate annotation language that library experts can use to encode domain-specific knowledge and convey it to the compiler. The language provides access to powerful compiler capabilities but presents them in a form that is suitable for non-compiler experts. The annotations for a library serve as a common repository of library-specific expertise from which all users of the library can benefit.

We have demonstrated that our system is an effective unified framework for both reducing the number of errors in programs and for improving their performance. We have used the same system to obtain results in two diverse domains: improving the performance of parallel linear algebra programs, and checking system software for security vulnerabilities. The key to our approach is that we provide a user-friendly way for library annotators to define domain-specific dataflow analysis problems. This capability allows the compiler to gather information, expressed directly in terms of the library abstractions, that captures how a program uses library routines. We also show that these analysis problems can pose significant challenges for a program analysis framework. Therefore, we present a new *client-driven* analysis algorithm that provides scalability and precision by adapting to the specific needs of each dataflow analysis problem it solves.

9.2 Future work

Our work represents the first steps in exploiting library interfaces as a means to incorporate domain-specific information in the compilation process. Our results and experiences point to several directions for future research:

- There are many useful features that we could include in future versions of the annotation language. For example, several existing systems for verifying software use more powerful and more concise specifications, such as temporal logic. While we remain cautious about exposing annotators to complex formal specifications, recent work has made considerable headway on presenting these valuable tools in a user-friendly form.

- Our work on library-level optimization suggests ways that future libraries could be designed with library-level compiler support in mind. First, we find that the libraries that present clear and well-defined abstractions are both easier for programmers to use and easier for the compiler to analyze. Second, many libraries already divide routines roughly into "basic" and "advanced" categories. In the future, library designers might view these categories as representing the programmer's interface, which emphasizes clarity and ease-of-use, and the compiler's interface, which provides fine-grained control over the implementation. Finally, future libraries might consist of an interface with no explicit implementation at all; instead, the annotations generate a customized implementation based on the library interface usage in each application. This approach combines the best features of optimizing compilers and software generators.

- Our client-driven analysis algorithm points towards a more general strategy for achieving high precision and high scalability for any compiler analysis or optimization. We use a fast low-precision algorithm to handle the easy common cases. We apply more sophisticated but more expensive algorithms on the difficult cases, which the fast algorithm cannot handle. We maintain scalability because we apply increasingly expensive algorithms to successively smaller fragments of the program. The key is to be able to determine when and where the more expensive algorithms are needed.

Our technique is not strictly limited to libraries: we can exploit module boundaries, wherever they occur in software, to convey domain-specific information to the compiler. Our research is part of a wider trend in programming language research towards using software modularity to improve the capabilities and the performance of software engineering tools. We hope that by providing tools that are practical as well as powerful, we can help to move some of the valuable advances in compiler research into everyday programming practice.

Bibliography

[1] Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, 1993.

[2] Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[3] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Englewood Cliffs, NJ, 1986.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.

[5] Uwe Assmann. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems*, 22(4):583–637, 2000.

[6] Darren C. Atkinson and William G. Griswold. The design of whole-program analysis tools. In *International Conference on Software Engineering*, pages 16–27, 1996.

[7] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK: High performance through high-level abstraction. In *Proceedings of the International Conference on Parallel Processing*, pages 414–423, 1998.

[8] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[9] P. Benner and E.S. Quintana-Orti. Parallel distributed solvers for large stable generalized Lyapunov equations. In *Parallel Processing Letters*, 1998.

[10] A. Berlin. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, 1990.

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

[12] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI c coding methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[13] James Bowman. Format String Attacks 101. http://rr.sans.org/malicious/format_string.php, October 2000.

[14] James M. Boyle, Terence J. Harmer, and Victor L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhauser (Springer-Verlag), Boston, 1997.

[15] D. L. Brown, William D. Henshaw, and Daniel J. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *SIAM conference on Object Oriented Methods for Scientfic Computing*, pages 177–??, 1999.

[16] CERT Advisory CA-2000-22. LPRng can pass user-supplied input as a format string parameter to syslog() calls. December 2000.

[17] CERT Advisory CA-2001-02. ISC BIND 4 contains input validation error in nslookupComplain(). January 2001.

[18] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, 1990.

[19] Arun Chauhan. Telescoping matlab for dsp applications. Technical Report Thesis Proposal, Dept. of Computer Sciences, Rice University, June 2002.

[20] S. Chiba. A metaobject protocol for C++. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 285–299, October 1995.

[21] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–165, 2002.

[22] Timothy Scott Collins. *Efficient Matrix Computations through Hierarchical Type Specifications*. PhD thesis, The University of Texas at Austin, 1996.

[23] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[25] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries (extended abstract). In M. Jazayeri, D. Musser, and R. Loos, editors, *Generic Programming. Proceedings*,

volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000.

[26] Manuvir Das. Unification-based pointer analysis with directional assignments. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[27] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[28] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[29] Luiz A. DeRose. *Compiler techniques for MATLAB programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[30] Edsger W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 9:27–37, 1976.

[31] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23, 2001.

[32] J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–28, 1990.

[33] Carter Edwards, Po Geng, Abani Patra, and Robert van de Geijn. Parallel matrix distributions: Have we been doing it all wrong? Technical Report CS-TR-95-39, University of Texas, Austin, 1995.

[34] E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B — Formal Models and Semantics, pages 995–1072. North-Holland, 1990.

[35] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, 2000.

[36] Dawson R. Engler. Incorporating application semantics and control into compilation. In *USENIX Conference on Domain-Specific Languages*, pages 103–118, October 1997.

[37] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[38] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *International Static Analysis Symposium*, pages 175–198, 2000.

[39] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2002.

[40] Matteo Frigo. A fast Fourier transform compiler. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, Atlanta, Georgia, 1999.

[41] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. DyC: An expressive an notation-directed dynamic compiler for c. *Theoretical Computer Science*, to appear.

[42] Samuel Z. Guyer, Emery Berger, and Calvin Lin. Detecting errors with configurable

whole-program dataflow analysis. Technical Report TR 02-04, Dept. of Computer Sciences, University of Texas at Austin, February 2002.

[43] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *USENIX Conference on Domain-Specific Languages*, pages 39–52, October 1999.

[44] Samuel Z. Guyer and Calvin Lin. Optimizing the use of high performance software libraries. In *Workshop on Languages and Compilers for Parallel Computing*, pages 221–238, August 2000.

[45] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *International Static Analysis Symposium*, 2003.

[46] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.

[47] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent Java software. *Lecture Notes in Computer Science*, pages 39–??, 2001.

[48] Mark Hayden and Robbert van Renesse. Optimizing layered communication protocols. Technical Report 96-1613, Cornell University, 1996.

[49] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[50] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engeneering (PASTE-01)*, pages 54–61, 2001.

[51] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[52] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, January 2001.

[53] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand Interprocedural Dataflow Analysis. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, 1995.

[54] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In USENIX, editor, *2002 USENIX Annual Technical Conference*, pages ??–??, 2002.

[55] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[56] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[57] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[58] Ken Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *International Parallel and Distributed Processing Symposium*, 2000.

[59] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[60] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira

Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[61] Gary A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[62] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 151–181, 1986.

[63] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–32, 2002.

[64] David Lacey and Oege de Moor. Imperative program transformation by rewriting. *Lecture Notes in Computer Science*, 2027:52–67, 2001.

[65] K. Rustan M. Leino. Extended static checking: A ten-year perspective. *Lecture Notes in Computer Science*, 2000:157–??, 2001.

[66] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[67] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[68] Tim Newsham. Format String Attacks. http://www.guardent.com/docs/FormatString.pdf, September 2000.

[69] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–324, October 1994.

[70] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispen Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental spe-

cialization: Streamlining a commercial operating system. In *ACM Symposium on Operating Systems Principles*, pages 314–324, 1995.

[71] Enrique S. Quintana and Robert van de Geijn. Specialized parallel algorithms for solving linear matrix equations in control theory. *Journal of Parallel and Distributed Computing*, 61:1489–1504, 2001.

[72] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[73] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, M. Srikant, and Mary Dell Tholburn. POOMA: A framework for scientific simulation on parallel architectures.

[74] Erik Ruf. Context-insensitive alias analysis reconsidered. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

[75] Radu Rugina and Martin Rinard. Design-driven compilation. In *Proceedings of the International Conference on Compiler Construction*, pages 150–??, April 2001.

[76] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In USENIX, editor, *Proceedings of the Tenth USENIX Security Symposium*, pages 201–220, 2001.

[77] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *International Static Analysis Symposium*, pages 16–??, 1997.

[78] Charles Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., 1995.

[79] Y. Smaragdakis and D. Batory. Application generators. *Encyclopedia of Electrical and Electronics Engineering*, Supplement 1, 2000.

[80] Yannis Smaragdakis and Don Batory. DiSTiL: a transformation library for data structures. In *USENIX Conference on Domain-Specific Languages*, pages 257–270, 1997.

[81] B. Steensgaard. Points-to analysis in almost linear time. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[82] James M. Stichnoth and Thomas Gross. Code composition as an implementation language for compilers. In *USENIX Conference on Domain-Specific Languages*, pages 119–132, 1997.

[83] Phillip Stocks, Barbara G. Ryder, William Landi, and Sean Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, 1998.

[84] Rob Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliabiity. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[85] Steven W. K. Tjiang and John L. Hennessy. Sharlit—A tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 82–93, 1992.

[86] Robert van de Geijn. *Using PLAPACK – Parallel Linear Algebra Package*. The MIT Press, 1997.

[87] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, MIT, Department of Electrical Engineering and Computer Science (also MIT/LCS/TR-598), 1994.

[88] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[89] Todd L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, pages 223–?? Springer-Verlag, 1998.

[90] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[91] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, 3^{rd} Edition*. O'Reilly, July 2000.

[92] Mark Wegman. Personal communication, 2002.

[93] Daniel Weise and Roger Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.

[94] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software (ATLAS). In *SC'98: High Performance Networking and Computing Conference*, 1998.

[95] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. *ACM SIGPLAN Notices*, 26(6):120–129, June 1991.

[96] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[97] Sean Zhang, Barbara G. Ryder, and William A. Landi. Experiments with combined analysis for pointer aliasing. *ACM SIGPLAN Notices*, 33(7):11–18, July 1998.

Vita

Samuel Zev Guyer was born in Rochester, New York on May 25, 1970, the son of Bernard and Jane Guyer. After completing his work at Brookline High School, Brookline, Massachusetts in 1988, he entered Boston University in Boston, Massachusetts. He transferred to Harvard University, Cambridge, Massachusetts in 1990 and received a Bachelor of Arts in 1992. During the following three years he was employed as a software developer at Hamilton Technologies, Inc. In September 1995 he entered the Graduate School of The University of Texas at Austin to pursue a Ph.D. in Computer Sciences.

Permanent Address: 5708 Avenue D
Austin, TX 78752
USA

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.